

# The **NEW** PCLinuxOS Magazine

```
--hide=PATTERN          that points to a direc
do not list implied en
      (overridden by -a or
--indicator-style=WORD  append indicator with
      none (default), slas
      file-type (--file-ty
-i, --inode             print the index number
-I, --ignore=PATTERN   do not list implied en
-k                      like --block-size=1K
-l                      use a long listing for
-L, --dereference      when showing file info
link; show informati
references rather th
fill width with a com
like -l, but list nume
```

**Command Line Interface Intro**

**Special Edition**

**October, 2010**

# Table Of Contents

Welcome From The Chief Editor .....	3
Command Line Interface: Part 1 .....	4
Command Line Interface: Part 2 .....	8
Command Line Interface: Part 3 .....	16
Command Line Interface: Part 4 .....	24
Command Line Interface: Part 5 .....	32
Command Line Interface: Part 6 .....	40
Command Line Interface: Part 7 .....	49
Command Line Interface: Part 8 .....	58
Command Line Interface: Part 9 .....	68
Command Line Interface: Part 10 .....	77
Command Line Interface: Part 11 .....	85
Command Line Interface: Part 12 .....	95

## Disclaimer

1. All the contents of the NEW PCLinuxOS Magazine are only for general information and/or use. Such contents do not constitute advice and should not be relied upon in making (or refraining from making) any decision. Any specific advice or replies to queries in any part of the magazine is/are the person opinion of such experts/consultants/persons and are not subscribed to by the NEW PCLinuxOS Magazine.
2. The information in the NEW PCLinuxOS Magazine is provided on an "AS IS" basis, and all warranties, expressed or implied of any kind, regarding any matter pertaining to any information, advice or replies are disclaimed and excluded.
3. The NEW PCLinuxOS Magazine and its associates shall not be liable, at any time, for damages (including, but not limited to, without limitation, damages of any kind) arising in contract, rot or otherwise, from the use of or inability to use the magazine, or any of its contents, or from any action taken (or refrained from being taken) as a result of using the magazine or any such contents or for any failure of performance, error, omission, interruption, deletion, defect, delay in operation or transmission, computer virus, communications line failure, theft or destruction or unauthorized access to, alteration of, or use of information contained on the magazine.
4. No representations, warranties or guarantees whatsoever are made as to the accuracy, adequacy, reliability, completeness, suitability, or applicability of the information to a particular situation.
5. Certain links on the magazine lead to resources located on servers maintained by third parties over whom the NEW PCLinuxOS Magazine has no control or connection, business or otherwise. These sites are external to the NEW PCLinuxOS Magazine and by visiting these, you are doing so of your own accord and assume all responsibility and liability for such action.

### Material Submitted by Users

A majority of sections in the magazine contain materials submitted by users. The NEW PCLinuxOS Magazine accepts no responsibility for the content, accuracy, conformity to applicable laws of such material.

### Entire Agreement

These terms constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes and replaces all prior or contemporaneous understandings or agreements, written or oral, regarding such subject matter.

# Welcome From The Chief Editor

Last year, when we started the Command Line Interface Intro series, I had no idea that it would run for a full year. It also exceeded all and any expectations that I had at the time. Pete Kelly, a.k.a. critter, produced a top notch series, one that literally walks the reader, step-by-step, through learning the Linux command line. It's only fitting that we produce this special edition of The NEW PCLinuxOS Magazine. Pete not only did an outstanding job on the series, but the end result is a series that any user who wants to learn the Linux command line can use as a reference and tutorial. Put together into one comprehensive volume should make it easier to follow along as you go through the series, allowing you to refer back to previous parts of the series as needed.

So why the command line? Sure, the vast majority of modern, desktop Linux distributions have a very comprehensive graphical user interface (GUI). Despite that, there are a number of reasons to learn the Linux command line.

First of all, it helps you to better learn how Linux works. There is something very basic about working from the command line, and it somehow gives you greater insight into the inner workings of Linux. At least, that's how I feel whenever I work from the command line.

Secondly, it's good to know more than one way to do things in Linux. What are you going to do if, say some day, your GUI desktop doesn't load, and you are forced to use the command line to fix things so you can get your GUI desktop back? Without knowledge of the Linux command line, you won't know what to do, or how to do it.

Third, as difficult as it may be to comprehend, some things are not only done more easily from the command line, but they can also be accomplished a whole lot faster.

Fourth, if you have any interest in learning programming or scripting in Linux, it is imperative that you learn the Linux command line. Of course, there are many other reasons to learn the Linux command line.

There is no doubt that the GUI of modern desktop Linux distributions make using Linux easier and have opened up Linux to many more users who might not otherwise give Linux a try. There are even Linux users today who work almost exclusively in the GUI of the Linux desktop. But treat yourself to learning the Linux command line. You will undoubtedly thank yourself in the end. It's certainly not as hard as you might think it is.

Use this series of articles by critter to set yourself on a voyage of discovery. At the end of the 12th and final installment of this series, critter lists some additional resources you may be interested in checking out, should you decide to learn more about the Linux command line.

Paul Arnote, PCLinuxOS Magazine Chief Editor



The PCLinuxOS name, logo and colors are the trademark of Texstar.

The NEW PCLinuxOS Magazine is a monthly online publication containing PCLinuxOS-related materials. It is published primarily for members of the PCLinuxOS community. The Magazine staff is comprised of volunteers from the PCLinuxOS community.

Visit us online at <http://www.pclosmag.com>

This release was made possible by the following volunteers:

**Chief Editor:** Paul Arnote (parnote)  
**Assistant Editors:** Andrew Strick (Stricktoo), Meemaw  
**Consultants:** Archie Arevalo, Tim Robinson  
**Artwork:** Sproggy, Timeth

**Magazine Layout:** Paul Arnote, Meemaw, ms\_meme  
**HTML Layout:** Galen Seaman

**Staff:**

Neal Brooks	ms_meme
Galen Seaman	Mark Szorady
Patrick Horneker	Macedonio Fernandez
Guy Taylor	Meemaw
Andrew Huff	Gary L. Ratliff, Sr.
Peter Kelly	Darrel Johnston

The PCLinuxOS Magazine is released under the Creative Commons Attribution-NonCommercial-Share-Alike 3.0 Unported license. Some rights are reserved.  
Copyright © 2010.

# Command Line Interface Intro: Part 1

by Peter Kelly (critter)

## So what is this CLI thing anyway?

*These tutorials are based on a basic, fully updated installation of PCLinuxOS using the KDE desktop. There may be minor differences between different installations and desktop environments but the fundamental concepts are similar.*

Most people who are new to Linux are often confused when seeking help and are told:

“In a terminal type...”, or

“This can be done using the CLI...”.

Terminal? CLI? Console? What does it mean? Well in this context the three terms can be used interchangeably. **CLI** is short for **command line interface** and is another way of interacting with your computer, as opposed to the usual **GUI** or **graphical user interface** found in desktop environments like KDE or Windows. So why bother? Well for the most part, you don't have to bother. The GUI will do almost everything that anybody could possibly need on a day to day basis. The keyword here though is almost. There are times when you need to do something that there is no button to click or menu item to select to perform it.

Anything you do in a GUI can be done using the CLI, plus a whole lot more. The down side is that you have to learn what commands to type. Not all commands, there are after all hundreds of them, but a basic working knowledge of a few dozen of the

more popular ones will make you a proficient CLI user.

## Getting started

Want to dip a toe in the water? Follow me.

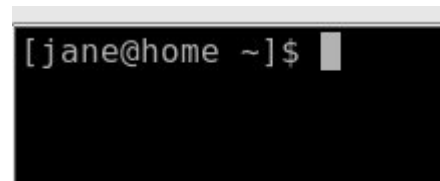
Open PCLinuxOS Control Center (PCC).



Under the System heading, click “Manage users on system” and create a new user. Call it whatever you like - my user is called jane. Log out and then log in as the new user. Now, if we manage to screw up, delete the user and re-create it. No damage done, and **your** account is safe.

Click the menu button and under System>Terminals select Konsole – Terminal Program. You could select anything under that menu, but this is the most suitable for what I have in mind.

A window opens with a black background and something like this:



This is called the prompt, because it is prompting you to type something. It is also giving some information: The current user is jane who is at a

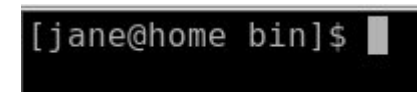
computer with the “hostname” home (If you haven't changed the hostname of your computer your prompt will say localhost). The ~ character (called tilde) is shorthand for “my home directory” which is where jane is in the filesystem. The \$ denotes jane is an ordinary user not the super-user root, whose prompt usually ends with a hash #. This is the default in PCLinuxOS but can be changed to include almost anything you want, for example, some people like to have the current date in the prompt.

## Looking around

So the prompt tells us that jane is in her home directory. Let's change that.

Type the following and press enter: **cd /usr/bin**. Make sure that you use only lower case letters, as Linux is case-sensitive.

The prompt now looks like this:



The command **cd** means change directory and the **/usr/bin** part is telling the command which directory to change to. But the directory shown in the prompt says **bin** not **/usr/bin**, and there are lots of directories named bin in the Linux file system.

It is very important that we know exactly where we are if we are going to start mucking about with things – no nice graphics here. To check, type in the



command **pwd**, short for “print working directory”, and press enter.

```
[jane@home bin]$ pwd
/usr/bin
[jane@home bin]$
```

This confirms where we are and supplies a new prompt ready for you to type the next command

Let's go back, type **cd** and press enter (you know to press enter by now). Typing the command **cd** on its own is a quick way of getting home from anywhere in the file system.

Nothing very impressive so far, and I said that you can do anything in here just like in a GUI. You can.

Type **firefox**. See, you still have access to all of your graphical applications while you are running a terminal in KDE. But now you can't type anything on the command line, so close firefox and you get the use of the command line again in your terminal (or console if you prefer). I'll show you later how to run a command and still have access to the command line.

Let's have a look around in this directory.

Type **ls** (it means list directory contents, in this example the contents of janes home directory).

```
[jane@home ~]$ ls
Desktop/ Documents/ Movies/ Music/ Pictures/ tmp@
[jane@home ~]$
```

This is a very brief and not very informative listing of the directory contents. If we want more information, then we can use a modified version of the **ls** command. For example, type **ls -l** (notice the space). Now type **ls -a**. See the difference? The **-a** and **-l** are known as options, and most Linux commands accept a host of options.

The **-a** (all files) gives you fewer details but more files. By default, any file that begins with a period is hidden. This has nothing to do with security but merely a convenience to reduce clutter. The **-l** option means provide a long listing. You can even combine options to get the results that you want.

Try **ls -al**.

How do you remember these options? You don't have to. For almost any command, typing the name of the command followed by **--help** will show you what is available. In practice, you will use only a few of the available options.

Type **ls --help**.

That went past too quickly to read so we need to slow it down.

Type **ls --help | less**.

The **less** command shows less information in one go. (The | is the vertical bar character above the backslash character. At least that's where it is on my keyboard). I'll explain later what this is all about. You can press enter or the space-bar for a new page and

the page up/page down keys work. Press 'q' to get out.

To summarize then, Linux commands take the following general form: **{cmd} {options} {argument}**

\* **cmd** is the name of the command, e.g. **ls**.

\* **options**, such as **-a** or **-al**, are ways of modifying the output or result of the command, and are usually prefixed with one or more dashes.

\* **argument** is information that you supply to the command, such as a directory to go to like **/home**.

Armed with the preceding information, you are ready to go exploring.

Logged in as a normal user you, can do no system damage with the following commands, which merely display information or move your current position within the filesystem. Some of the information displayed may look a bit cryptic at the moment, but I will explain it all when we have been through a few more commands.

<b>ls</b>	
<b>cd</b>	
<b>pwd</b>	
<b>free</b>	display memory usage
<b>df</b>	display disk usage
<b>date</b>	print the time and date to screen
<b>cal</b>	print a calendar on the screen

## Making changes

This is where we have to be a little more careful, as we will be making changes to files and directories and deleting some. But that is why we created this dummy account. (You did that, right?)

**cd** enter puts us back in the home directory from wherever we were in the file system.

Create a new directory named **mydir1** with the command **mkdir mydir1**. Check that it has been created with the **ls** command.

Now let's create a new file. One way to do this is to use the command **touch**.

### touch myfile1

This isn't what **touch** was designed for, but is what it is used for mostly these days. **ls** will show the new file, but it isn't in our new directory, so let's move it.

### mv myfile1 mydir1

**ls**           The file has gone.

**ls mydir1**    There it is in our new directory so move to our new directory.

### cd mydir1

**file myfile1** shows the file type to be empty

```
[jane@home mydir1]$ file myfile1
myfile1: empty
[jane@home mydir1]$ █
```

Remember the vertical bar character that we used with the less command? It took the output from the **ls --help** command and fed it to the less command so that we could read it in our own time. The **>** (greater than) character can be used in a 'similar' way.

**ls -l /** will give a listing of the root of the file system.

### ls -l / > myfile1

Here the output is captured before it is printed to screen, and then stuffed into our file. This is part of something called redirection, which we will go into in more depth in a later episode.

**file myfile1** now shows the file to be of type 'ASCII text'. To look at the contents we can use the command **cat** (short for concatenate).

### cat myfile1

### I hate typing in long commands

So do most people. You may have noticed that Linux commands tend to be short (e.g. **ls**, **cat**, **df**, **cd**, **rm**). Linux has its roots in Unix, an operating system dating back 40 years, to the days when there was no graphical interface and all work had to be done on the command line. Over the years, people have developed many ways to reduce the amount of

```
[jane@home mydir1]$ cat myfile1
total 88
drwxr-xr-x  2 root root  4096 Jun 29 16:18 bin/
drwxr-xr-x  3 root root  4096 Sep  5 10:01 boot/
drwxrwxrwt 12 root root  4200 Sep  6 04:02 dev/
drwxr-xr-x 114 root root 12288 Sep  5 12:20 etc/
drwxr-xr-x  5 root root  4096 Sep  5 12:20 home/
drwxr-xr-x  2 root root  4096 Sep  4 14:15 initrd/
drwxr-xr-x 17 root root  4096 Sep  4 14:02 lib/
drwx----- 2 root root 16384 Sep  4 13:40 lost+found/
drwxr-xr-x  2 root root  4096 Sep  5 12:20 media/
drwxr-xr-x  3 root root  4096 Sep  4 13:57 mnt/
drwxr-xr-x  5 root root  4096 Jun 29 01:32 opt/
dr-xr-xr-x 111 root root    0 Sep  5 05:00 proc/
drwxr-xr-x  34 root root  4096 Sep  6 13:11 root/
drwxr-xr-x  2 root root 12288 Sep  4 14:04 sbin/
drwxr-xr-x 11 root root    0 Sep  5 05:00 sys/
drwxrwxrwt 15 root root  4096 Sep  6 13:19 tmp/
drwxr-xr-x 13 root root  4096 Jul 14 00:24 usr/
drwxr-xr-x 15 root root  4096 Sep  4 08:37 var/
[jane@home mydir1]$ █
```

typing and speed up regularly performed tasks, as you will discover.

When you run a terminal session, such as Konsole, under Linux (notice how the terms terminal, console and CLI are blending together), you are actually running an application known as a **shell**. The shell takes what you type and interprets it into something that the computer can work with. There are lots of shell programs available for Unix/Linux. The original shell was known as the Thompson shell, or **sh** on the command line. This was developed by Stephen Bourne to include many more features, but was still known as **sh**. The most popular shell today is called **'bash'** (bourne again shell). The default shell in PCLinuxOS is **bash**.

You can check this by typing **echo \$SHELL** on the command line.

Bash is a very powerful program that can save you no end of typing.

Press the up arrow a few times and you will see the last few commands that you typed at the prompt, ready to be edited or executed. The down arrow takes you back down. Type **history** and you get a numbered list of all the commands that you have used in this session

type **!** (The number) enter, e.g. **!38**, and that command is echoed to the screen and then executed.

Another time saver is a feature known as 'command line completion'. It works like this:

Type **hi** and press the **tab** key.

You get prompted with a list of all the commands you can execute that begin with the letters "hi" .

Press the 's' key and tab and the command line completes the command history, waiting for you to add any options or arguments, then enter to execute the command. This not only saves on typing, but serves as a reference when you can't quite remember the command and cuts down on typing errors.

If you are part way through typing a command and notice a spelling mistake, you can use the following key combinations to get to it:

- ctrl + a** takes you to the beginning of the line
- ctrl + e** takes you to the end of the line
- alt + b** takes you to back a word
- alt + f** takes you forward a word

There are more, but these few are easy to remember and are usually sufficient.

If you have previously typed a long command and don't want to have to re-type it, then **ctrl-r** starts a reverse search. Just start typing the command, and as you type, the shell will match what you type to the most relevant previous command. When you have an exact match, just press enter.

Type a letter, any letter, e.g. 'a' then press tab and return.

```
[jane@home mydir1]$ a
Display all 134 possibilities? (y or n)
```

Answer 'y' and press return.

That gives you some idea of the number of commands we have to play with (almost 2000 in a basic installation of PCLinuxOS 2009.2).

Don't worry. You will only ever need a very small number of these commands but, if you need to do something, then there is a command or set of commands that will do it for you.

So you've dipped a toe in the water. How does it feel? Hopefully not too bad, because next time we are going to wade in waist deep, doing some things with root permissions. That is where the command line comes in to its own.



Visit.  
Contribute.  
Build.

The PCLinuxOS  
Wiki

*It Belongs To YOU!*



# Command Line Interface Intro: Part 2

by Peter Kelly (critter)

In last months tutorial, I presumed that anybody reading it had no experience whatsoever of using the command line. If you worked through that, then you should be ready for a more in depth look. There is nothing too taxing in here, but you may find more text per sub-heading. If you find an area where you come to a brick wall, just walk around it and carry on. Maybe come back to revisit it, or maybe wait for that forehead slapping moment "Dohh!" or even "Eureka!" when enlightenment arrives.

If you followed along with last months installment, you will now have a little experience of typing commands on a command line using the application konsole. So what? You could have done any of those things without having to do all that silly typing.

The application konsole is known as a terminal emulator. It allows you use the command line, without losing sight of your warm and cozy KDE GUI. But what happens if the X System, that is the windowing system that KDE runs on, crashed? Or some configuration file that the system depends upon got corrupted, and when you booted up, you were greeted only by some weird message and an almost blank screen?

## A leap in the dark

Press and hold **Ctrl + Alt** and press **F2**.

```
PCLinuxOS release 2009 (PCLinuxOS) for i586
kernel 2.6.26.8.tex3 on an i686 /tty2
home login: █
```

Now that is a terminal.

Don't panic! Your precious GUI is still around.

**Ctrl + Alt + F7** gets you back.

Actually, you could have pressed any of the function keys from F1 to F6 to get a raw, text only terminal.

I am currently logged into my KDE session as user jane, so if I now drop into a text terminal as before

**Ctrl + Alt + F2**

I am prompted to log in.

```
home login:
jane (enter)
password
```

Yes, in Linux I can be in two places at the same time. I am logged in as jane in my KDE session, and I have now logged in again as jane in this terminal. I now have access to all janes files and can edit them, delete them, move them, rename them and create new ones. I could have logged in as any user that I knew the password of and had access to all of their files. If I had logged in as root then I could have had access to all files on the system and have inadvertently caused chaos. For that reason you should avoid logging in as root at all costs, there are other ways to do things. There are times when it is necessary to log in as root but it is very rare and should only be done if you are absolutely sure about what you are doing.

## A change of direction

Type **cd ~** to make sure that you are in your home directory, and then create a new file by using the command.

**touch newfile (enter)** which creates a new, empty file called "newfile"

```
echo "this file was created in
terminal 2 on " > newfile
```

puts some text into the file.

Remember that the **>** symbol catches the output from the command and puts it into the file (replacing what was there originally so be careful when using it).

Type **date >> newfile**

Using the symbol twice **>>** catches the command output and appends it to the file.

**exit** logs me out and

**Ctrl + Alt + F7**

puts me back into KDE. Or, more correctly, into the terminal that is running the X System and the KDE environment.

Let's have a look at the contents of the file that we just created.

```
cat newfile
```



```
[jane@home ~]$ cat newfile
This file was created in terminal 2 on
Sat Sep 19 12:11:24 CDT 2009
[jane@home ~]$ █
```

What we are doing here is known as 'redirection,' and is a very important concept for working on the command line. Most Linux commands are 'stream oriented.' This means that data flows into and out of the command rather like a sausage machine – meat in one end, turn the handle and get sausages from the other end. The data is processed by the command as it flows through the command.

Let me try to explain what happens when you sit down and start to type a command at a terminal running a shell program such as bash. As you press a key ( or a combination of keys, like Shift + a), the shell program stores the value of that key press in a special area of memory known as a buffer and prints a copy of it to the screen (usually the screen anyway – see later). It then waits for another key-press to add that also to the buffer. When you press the enter key, it signals the end of that batch of input, and the entire contents of the 'keyboard buffer' are sent to be interpreted. This where bash works its' magic.

Bash takes all of the key presses that you have typed from the keyboard buffer, lays them out on the table into groups that you separated with spaces, looks for any group of characters that it recognizes as a command which it can execute, looks for certain special characters that have a special 'shell' meaning and then decides what to do with the rest of the groups on the table based upon what it has just found.

Usually, this just means that when you enter a command line such as

```
ls /home/jane
```

bash has two things on the table, `ls` and `/home/jane`.

Now bash recognizes `ls` as a command and so it looks for input information (how to use the command), any options that may modify the default way that this command performs its' function, and for what to do with the results. (This a very simplified overview but is sufficient for our present needs).

Previously we defined the command format to be

```
{cmd} {options} {argument}
```

Now we know a little more about bash we can expand this to

```
{cmd} {options} {input argument}
{output argument}
```

`{cmd}` is the name of the command to execute.

`{options}` such as `-a` or `-al` are ways of modifying the output or result of the command.

`{input argument}` is anything that you want to send to the command to work with.

`{output argument}` is where you want the results of the command to go.

Now that bash has found a command `ls`, it looks for

a group of key presses on the table that qualifies as a suitable `{input argument}` for the command and finds `/home/jane`. If nothing qualifies, then the programmer who wrote the command will hopefully have provided for a 'no input' default condition. `ls` with no input defaults to the value 'wherever I am now.' There are no options to tell the command to modify its' output, therefore the output will be the default for the `ls` command – a simple listing.

With no `{output argument}`, most commands default to 'print it to the screen'.

So this command prints a simple listing of the directory `/home/jane` to the screen.

These default values for where the input comes from, and to where the output is directed, can be changed by redirection. This is where you tell bash to temporarily change its habits, and to take instructions from the command line.

The shells default input and output are known as `stdin` and `stdout` (the standard input device – usually the keyboard, and the standard output device, which is normally the screen). (There is a third data stream known as `stderr` 'standard error,' but let's learn to walk first, eh?)

You can however, redirect data from other sources, or to other destinations such as files. In the previous examples, we have redirected the output to a file, instead of displaying it on the screen by using the `>` operator. To redirect the input from somewhere other than `stdin`, we use the `<` symbol. Try this.

`ls /etc > newfile2` lists the contents of the directory /etc to the file newfile2.

`sort -r < newfile2` sorts the contents of the file in reverse order.

Here, the `ls` command takes its input from `stdin` (/etc) which was typed into the keyboard buffer and the output is redirected to the file newfile2 instead of being printed to the screen. In the next line, the `sort` command uses contents of the file newfile2 as its input and, as we haven't specified otherwise, sends the output to `stdout`, the screen.

There is a better way to do this using a mechanism which you have seen before. It is called a pipe, and looks like this |.

`ls /etc | sort -r` gives the same result as the two lines above and cuts out the middleman i.e newfile2. The output from the command `ls /etc` is pushed through a pipe | into the command `sort -r`

So what's the difference between `ls > sort` and `ls | sort`?

This is often a source of confusion. `ls > sort` takes the output from the command `ls` and redirects it to the file 'sort', which it creates if necessary, rather than to the screen (`stdout`). Probably not what was intended.

`ls | sort` takes the output and pipes it through the command `sort`, which in turn sends its' output to the screen (`stdout`), as this output has not been redirected. In this manner, fairly complex commands can be built up.

Create a small file with a random list of names using some of the bash editing features described last month.

```
touch contacts
echo john > contacts
Use the up-arrow to bring back the
previous line then alt + b and the
delete key to edit the line.
echo amy > contacts (don't forget to
use the > (append) operator here.)
echo gustav > contacts
echo bob > contacts
echo glenn > contacts
echo simon > contacts
Look at the file contents
type
cat contacts
```

```
john
amy
gustav
bob
glen
simon
george
```

```
cat contacts | sort | tr [a-z] [A-z] >
contacts2
```

What did that do?

```
cat contacts2
```

```
AMY
BOB
GEORGE
GLEN
GUSTAV
JOHN
SIMON
```

In the above compound command, the contents of the file contacts is piped to the `sort` command which, with no options supplied to modify the output, sorts the contents alphabetically from a to z, which is the commands default action. This in turn, is fed to the `tr` (translate) command which converts any character in the range [a-z] to its uppercase equivalent [A-Z]. Finally, the results from the translation are written to a file called contacts2, which will be created if necessary, or overwritten if it already exists. Don't worry if you don't understand how these new commands work. I just want you to get an idea of how we can 'flow' data from files, through commands and filters, and then write that data to a file or to the screen.

### Editing on the command line

When we made the file contacts, we did it line by line, which is obviously unsatisfactory for all but the simplest files. What is really needed is a text editor. We could call up a graphical one that we are familiar with, such as `kwrite`, but not when we are in a text terminal, as we were when we typed `Ctrl + Alt + F2`. If, for example, the X windowing system

won't start, then you may well find yourself in just that position.

The basic editor which you will find in almost every distribution is vi, which we will visit later, as it may well be that one day it is all you have at your disposal. It is however, very powerful editor, though difficult and not very intuitive to use for new users.

Fortunately, PCLinuxOS comes with a very nice, simple editor for command line work. Meet nano!

Typing **nano** on the command line opens the editor with a blank page. If you specify a file-name after the command **nano**, then it will open that file, if it exists. If no file of that name exists, then no file is created at this stage but you will be prompted to save your work with this file-name when you exit. Only then is the file created.

Typing **nano contacts2** opens the editor with our sorted file, and the cursor is on the first character of the first line.

The screen is divided into four areas:

- \* The top line of the screen is known as the header bar. This shows the version of nano and the name of the current file being edited. If you didn't specify a file name, then this will read 'new buffer.' If the file has been modified since the last save, then 'Modified' will be shown on the right hand side of the header bar.

- \* The bottom two lines show a list of command shortcuts. Nano commands are defined by either the control key being held down while the shortcut key is pressed, or by the shortcut key being preceded by the escape key. The caret symbol ^ represents the control key, so for example, **Ctrl + x** exits the program. The escape key is represented by M. These few commands are usually enough for most purposes. **Esc - a** marks text, and **Esc - m** enables/disables limited mouse support. If you want more then there is more, **Ctrl + g** will show you a brief introduction.

With the file contacts2 loaded, use the arrow keys to position the cursor on the letter L of the name GLENN. Press **Esc - a** to start marking text. Press the right arrow key 3 times to mark the 3 letters LEN press **Ctrl + k**. This removes the three letters and places them in the cut buffer, a temporary storage area. Press **Ctrl + u**, and this inserts the contents of the cut buffer at the current cursor position restoring the name GLENN. Use the arrow keys to position the cursor at the end of the file, and press **Ctrl + u** again to add the new contact LEN. This is cut & paste, nano style.

**Ctrl + o** prompts you to write out the file with name contacts 2. Pressing **enter** saves the changes and puts you back in the editor. If you change the file name to save the file as, you will be prompted to confirm this, and be returned to the editor with the new file.

Add a few more names then press **Ctrl + x**. Answer y and press enter to leave the editor saving your changes.

This type of simple editor is ideal for beginners to edit Linux configuration files, as it produces only text with no fancy formatting that might be misinterpreted. If you want to write a novel use something else.

## Sitting in the bosses chair

For some things you do need to have the special privileges of the root user and the safest way to do this is to use the command **su**. This command

```

GNU nano 2.0.9 File: contacts2
AMY
BOB
GEORGE
GLENN
GUSTAV
JOHN
SIMON

[ Read 7 lines ]

^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell
    
```

- \* The third line from the bottom, just above the list of the commands, is the status line which shows that nano read in 7 lines from the file contacts 2.

- \* The rest of the screen is the editing area.

allows you to 'switch user' identity to that of any user you know the password of.

`su john` will prompt you for the password of john. If there is a user account for john, and if the password is successfully entered, then the shell will allow you full access to all of johns files and directories. This is why you should keep your password safe. `su` is a very powerful command.

Typing `su` without a user name will assume that you want to have root access to all files and directories, and will prompt you for the root password.



**When you have root privileges you are able to make your system completely unusable!**

Let's do some root stuff.

Notice that the prompt symbol has changed from \$ to #?

```
[jane@home ~]$ su
Password:
[root@home jane]#
```

Maybe you didn't, but I'm pretty sure that you noticed the prompt is now bright red. This is not always the case, but the developers of PCLinuxOS believe that you really should be aware that you are now in a position to do some real damage and have modified the prompt to reflect that. Notice also that the prompt shows I am working now as root not jane but I am

still in janes' home directory. Be aware that the commands `cd` and `cd ~` will now take you to the directory `/root`, and not to janes home directory, `/home/jane!`

One thing that root can do that mere mortals cannot is to add and delete users on the system. To add a new user named john to the system, the command `useradd john` creates the user account, and sets up the user environment by copying the files that the system administrator or the distribution developers have placed in the `/etc/skel` directory.

It does not add the user to any groups other than the users default group. This can be done here with the `-G` option, followed by a list of groups, or later with the command `usermod`. We'll cover groups later when we get to file permissions.

You should follow the account creation with

```
passwd john
```

to create an initial user password for the new john account and then pass this password to the user who, once he has logged in with it, may change it using the same command.

```
userdel john
```

deletes the user. If you specify the `-r` option here then the users home directory and any files it contains will be deleted.

There is also a command called `adduser`, which is similar to `useradd`.

Having done our work, we should renounce our special root privileges with `Ctrl + d` or the `exit` command.

```
[root@home jane]# exit
exit
[jane@home ~]$
```

Now check that the account has been successfully created.

```
[root@home jane]# exit
exit
[jane@home ~]$ su john
Password:
[john@home jane]$ cd ~
[john@home ~]$ pwd
/home/john
[john@home ~]$ exit
exit
[jane@home ~]$
```

`su john` logs me in to johns account, but the prompt tells me that I am still in janes' home directory

`cd ~` as I am now logged in as john, this takes me to johns home directory, which I verify with the command

```
pwd
```

The `exit` command logs me out of johns account and puts me back into janes account, and also back



into whatever directory jane was in when she issued the `su` command.

We'll return to the root terminal later when we have a few more commands to use.

## Customizing our environment

After using the command line for a while, you will find that many times you type in the same commands and options over and over. Surely somebody can think of a better way?.

They did. It is called an **alias**, and is a way of giving a command that you regularly use its' own name. You already have some aliases in PCLinuxOS. Type the command `alias` to show them.

```
[jane@home ~]$ alias
alias cd.='cd ..'
alias cp='cp -i'
alias d='ls'
alias df='df -h -x supermount'
alias du='du -h'
alias grep='grep --color'
alias kde='xinit /usr/bin/startkde'
alias l='ls'
alias la='ls -a'
alias ll='ls -l'
alias ls='ls -F --show-control-chars --color=auto'
alias lsd='ls -d */'
alias mc='./usr/share/mc/bin/mc-wrapper.sh'
alias md='mkdir'
alias mv='mv -i'
alias p='cd -'
alias rd='rmdir'
alias rm='rm -i'
alias s='cd ..'
[jane@home ~]$
```

Look at one about halfway down the list: `alias ll='ls -l'`

If you type

```
ll
```

on the command line bash will interpret this as `ls -l`, and execute it accordingly.

Let's make our own new alias. Suppose that I often want a hard copy of a directory listing with the contents sorted by file size and with these sizes in a format that is easily understood.

To print out files on the printer in Linux we use the command

`lpr`

The command `lpr myfile1` will send the contents of the file `myfile1` to the default printer without the need for any redirection by the user but it is also common practice to pipe the input to `lpr` from another command.

I want to create an alias that will print out my listing easily and I would like to use the name `lspr`, but I don't want to conflict with any existing system command. So enter `ls` and then press **tab** to show a list of all commands that start with the characters `ls`.

```
[jane@home mydir1]$ ls
ls          lshal      lskatproc  lsof       lspcmcia   lsusb
lsattr     lsinitrd  lsmod      lspci      lspgpot
lsd        lskat     lsnetdrake lspcldrake lss16toppm
[jane@home mydir1]$ ls
```

From this, I can see that there is no command named `lspr` that I have access to and so I am safe to choose this as the name of my alias.

Press **Ctrl + c** to cancel the command.

To create the alias, I use the command

```
alias lspr="ls -lhSr | lpr"
```

This tells bash "whenever I type the key combination `lspr` execute the command `ls -lhSr | lpr`."

This creates a long (option `l`) directory listing in human readable form (option `h`), Sorted by file size (option `S`) in reverse order (option `r`) and pipes the output to the printer.

Make sure that your printer is switched on and connected, then type

```
lspr enter.
```

This way, I don't have to remember how to format the command, just the alias `lspr`.

Unfortunately, as soon as you end this session of bash by logging out or by closing the konsole window, this new alias is lost. To make it permanent, we need to edit one of those hidden files, the ones whose names begin with a period, in your home directory, `.bashrc`.

This is the bash resource configuration file and is read every time a new instance of bash is invoked.

```

GNU nano 2.0.9      File: .bashrc
# .bashrc
# User specific aliases and functions

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

[ Read 8 lines ]
^G Get Help  ^O WriteOut  ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is   ^V Next Page  ^U UnCut Text ^T To Spell
    
```

nano ~/.bashrc

will open the `.bashrc` file that is in your home (~) directory, ready to be edited.

Press the down arrow until you reach the end of the file and then add the alias and press enter.

While we are here, copy and paste the following

```
export PS1='\[\033[01;32m\]\u@\h > \W
\$\[\033[37m\] '
```

Make sure that you include the final quote mark ('), then press enter.

Always press enter at the end of a system configuration file to make sure that it ends with a new blank line.

It should now look like the image at the bottom of this column.

Press **Ctrl + x** and answer **y** to the prompt then press enter to save the modified file. Close the konsole window to end the bash session, and then

restart it. This is necessary to enable the new instance of bash to read the modified configuration file.

If all went well, you should be able to type `lspr` to get your printout, and you should have a nice green prompt to identify you as "not root." If you don't like green, then you can change it by altering the `01;32` part of this line.

```

# .bashrc
# User specific aliases and functions

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi
alias lspr="ls -lhSr | lpr"
export PS1='\[\033[01;32m\]\u@\h > \W \$\[\033[37m\] '
    
```

```
export PS1='\[\033[01;32m\]\u@\h > \W
\$\[\033[37m\] '
```

Change 32 to a value between 30 and 37 to change the basic color.

Where have 01, we may put several different values:

- 00 for normal colors
- 01 for bright colors
- 04 for underlined text
- 05 for blinking text
- 07 for reverse video text

These can be combined e.g. `01;04;05` for bright, underlined, blinking text.

Adding a value between 40 and 47 changes the background color e.g. `1;34;47`

To try out the colors on the command line use

```
echo -e '\033[01;37;44mPCLinuxOS -
Radically Simple\033[0m'
```


and substitute `1;37;44` for the above values separated by a semicolon `;`. The `-e` option added to the `echo` command tells it to interpret certain sequences of characters, known as escape sequences, rather than just blindly printing them on the screen, which is why we don't see all that gobbledygook on the screen.

```

jane@home > - $ echo -e '\033[01;37;44mPCLinuxOS - Radically Simple\033[0m'
PCLinuxOS - Radically Simple
jane@home > - $
    
```

Experiment with different color combinations. Maybe login or `su` to johns account and change his prompt to blue. As long as you stay away from the root account you can do no real harm. These are, after all, only dummy accounts.

Do you feel more at home now?

Want to keep up on the latest that's going on with PCLinuxOS?

Follow PCLinuxOS on Twitter!

<http://twitter.com/iluvpclinuxos>

Managing My Data Center.

- Flexible
- Reliable
- Secure
- On-Demand Computing

...delivered in a win/win relationship



**EN\*KI** The Computing Utility™ [www.enkiconsulting.net](http://www.enkiconsulting.net)

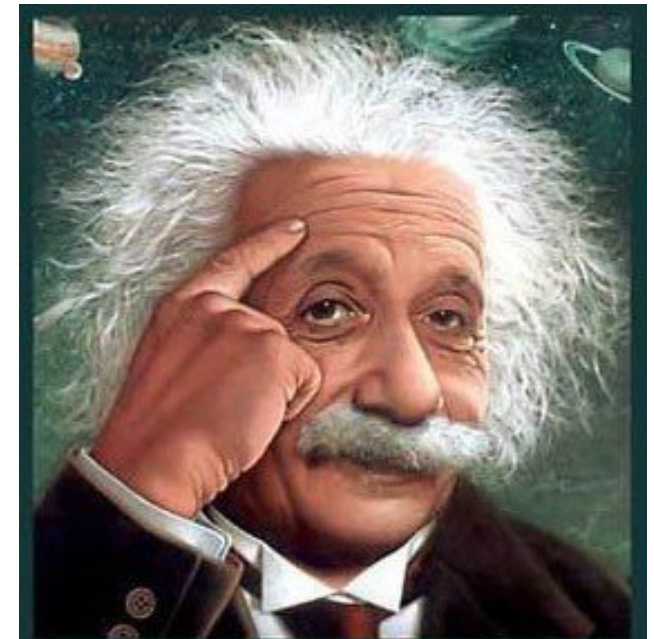


A magazine just isn't a magazine without articles to fill the pages.

If you have article ideas, or if you would like to contribute articles to the PCLinuxOS Magazine, send an email to:

[pclinuxos.mag@gmail.com](mailto:pclinuxos.mag@gmail.com)

We are interested in general articles about Linux, and (of course), articles specific to PCLinuxOS.



**It's easier than  $E=mc^2$**   
**It's elemental**  
**It's light years ahead**  
**It's a wise choice**  
**It's Radically Simple**  
**It's ...**



# Command Line Interface Intro: Part 3

by Peter Kelly (critter)

In the first two installments of this introduction, we learned how to get around the file system, create and edit files, how to use some of the more common commands in Linux, and how to cut down on some of that tedious typing. Hopefully, the command line environment is not so intimidating now. There is a bit more theory to cover but it is nothing too difficult, and it really doesn't matter if you don't understand it all. When you come to a point where you need that information, you can recall "hey! I read about that," and then come back to look it up, or search the internet for it. Importantly, you will have been introduced to the fact that it exists, which gives you a way forward. If you have followed so far and get through this episode, then you will have a good grasp of what Linux is about. Then we can start using the real power of the Linux command line that you can never fully achieve by using only the GUI.

## A New Name

When Jane got her computer, she chose to name it 'home.' It is, after all, the computer that she uses at home. This has a couple of disadvantages. What if the computer is to be networked, and her brother, John, has also named his computer home? There would be two computers on the network, both named home, and obviously this is not good. Also, when Jane looks at her prompt, she sometimes mistakes home in the prompt to be telling her that she is in the home directory. Let's change things. Jane decides that she wants her computer to be known as 'daisy'.

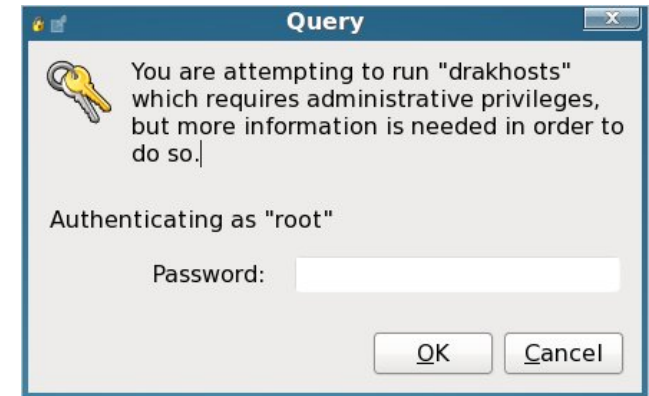
The name of a computer on the network is known as the 'hostname'. If no name has been assigned, then the name 'localhost' (meaning "this computer" ) will be shown. Any change needs to be done in two places in PCLinuxOS, and it needs to be done with root privileges. The name localhost is how the computer refers to itself internally, a bit like saying 'me'. In order for the computer to recognize that the new hostname daisy refers to this computer, we have to create an alias. This is like saying daisy, a.k.a. localhost.

The first file that needs to be edited is named hosts, and can be found in the /etc directory. This can be done in two ways: by editing the file directly with an editor, or by using the PCLinuxOS Control Center. The PCLinuxOS Control Center, also known as PCC, is a front end for several smaller, graphical utilities whose names usually contain the word 'drak' and do a little bit of command line work for you. To prove a point:

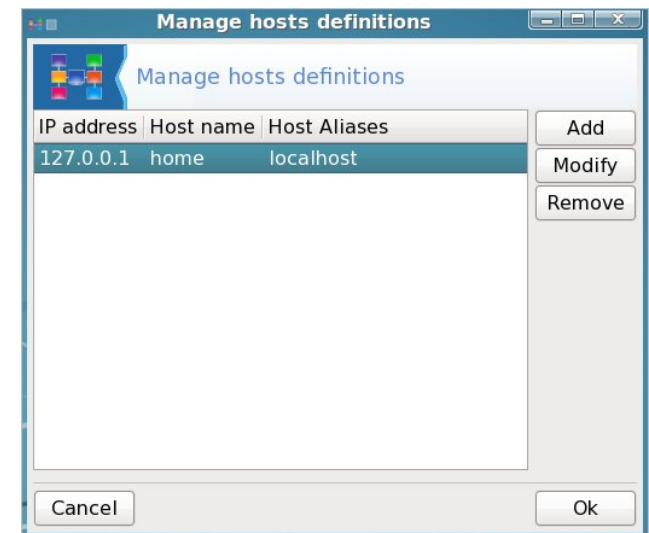
cat /etc/hosts

```
jane@home > host $ cat /etc/hosts
# generated by drakhosts
127.0.0.1 home localhost
jane@home > host $
```

If you have never changed it, your hosts file will look slightly different. Type drak and press Tab to see the available commands. The command we want is drakhosts (the Control Center 'front page' is drakconf). Type 'ho' and press Tab to complete the command and press enter. If you did this as a normal user, then you will be prompted for the root password.

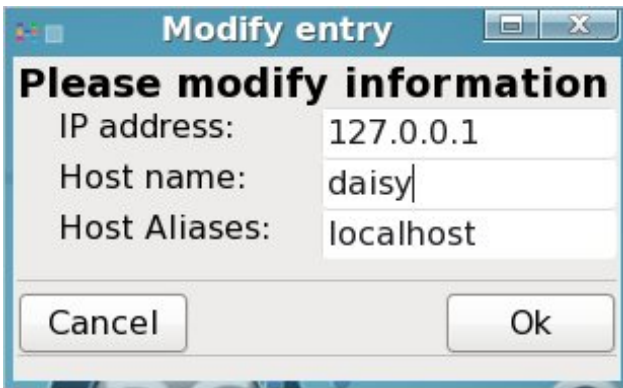


Then you will get this screen.



Notice that there are three columns: IP address, hostname and host aliases. Select localhost and then click modify.





In the host name box, enter the name you want for your computer (jane has chosen the name daisy), and in the host aliases box enter localhost (lower case no spaces), Then click OK in the modify dialog box, and click on OK in the drakhosts box to get back to the command line.

#### cat /etc/hosts

```
jane@home > host $ cat /etc/hosts
# generated by drakhosts
127.0.0.1 daisy localhost
jane@home > host $
```

Notice that the contents have changed. Most of the PCC utilities are just fancy-pants ways of editing system configuration files.

Now that this computer knows that we are referring to itself when we use the name daisy, we need to make sure that all other computers on the network also know.

Notice the IP address. That's the sequence of four numbers at the beginning. Localhost always has this sequence 127.0.0.1 – it's how the computer spells 'me.' Computers speak in numbers. An IP address is how computers refer to each other. On a network, a computer is known by the IP address assigned to its network adapter, usually something like 192.168.0.1, which is not easy for humans to remember, so we give the computer a 'proper' name, like daisy, so that it is more easily recognized and referred to on the network.

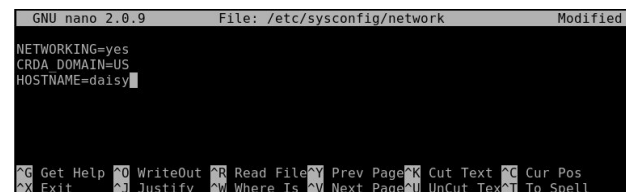
The second file that needs to be changed is called network, and is in the sysconfig directory, a sub-directory of /etc.

We'll use the terminal to change this file.

#### su (enter)

#### Enter the root password.

nano /etc/sysconfig/network will open the file for editing. Add or change the line HOSTNAME=daisy (or whatever name you chose). Notice uppercase and no spaces.



Press enter and then ctrl-x to save the file and exit. Job done! - command line style.

That's it, but you will need to reboot the apply the changes.

```
jane@daisy > ~ $
```

## Finding Things

To be able to work with files, we have to know where they are or how to find them. There many ways in Linux to get this information, so let's run through some of them.

**locate** – this command uses a database of files known on the system to look up their whereabouts. This database is updated on a daily basis automatically by the system using the cron utility, which we will look at in due course. The database may be updated manually at any time using the command updatedb. The updatedb command needs root privileges, while locate doesn't.

```
jane@daisy > ~ $ locate .bashrc
/etc/skel/.bashrc
/home/jane/.bashrc
/home/john/.bashrc
jane@daisy > ~ $
```

**.bashrc** is the file that we edited to change the color of our prompt. Locate has found 3 instances of the file: Jane's, John's, and the one that is used when a new user account is created. Locate has the advantage of being extremely fast, but the disadvantage of only knowing about the files that it has been told to store in it's database.

```
jane@daisy > ~ $ whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.bz2
jane@daisy > ~ $
```

**whereis** - this command looks only for Linux commands and their associated source and documentation files.

Here, the binary file that is the actual ls command, and its compressed manual documentation file, has been reported.

**find** – an extremely powerful command with a slightly more complex syntax than most other Linux commands. Most people (that's us) will only need a very small amount of the power available in this command, so we shan't look too deeply at all the bells and whistles – yet.

By default, find uses the current directory for its input, STDOUT (the screen) for its output and 'all files' as the option, which results in the output from using find being the same as from the command ls -aR, albeit in a slightly different format. Try it. All that output is because find, unchecked, looks at all files, including hidden ones, in its start directory, and then recursively in all sub-directories, outputting all those files that match its search criteria, in this case 'all files'.

To make use of the find command, we have to control it. The most common way of using find to locate a particular file or set of files looks like this. Find {where to start looking} {search criteria} {filename(s)}

There are lots of things that could go in the {search criteria} position, but usually we want to search for a file by name. So, to find a file named 'network' that is believed to be somewhere in the /etc directory structure, we would format our command thus:

```
jane@daisy > ~ $ find /etc -name network
find: `/etc/portreserve': Permission denied
/etc/rc.d/init.d/network
find: `/etc/skel/.kde': Permission denied
find: `/etc/cups/ssl': Permission denied
/etc/netprofile/profiles/default/services/network
/etc/sysconfig/network
jane@daisy > ~ $
```

This has found three files named network and given us their locations. Unfortunately, it has also thrown out some errors. The /etc directory is a system directory, and as such is not owned by Jane. Jane can only see the files in those directories to which the system has granted her permission to read the contents. When we covered redirection, I talked about **STDIN** and **STDOUT**, and mentioned a third data stream named **STDERR**. These three data streams may also be referred to by numbers, 0,1 and 2 respectively. **STDERR**, number 2, is to where programs write their error messages. Depending on the program, this may be a file such as a log file or to **STDOUT**, as was the case here. To hide these messages, we can redirect **STDERR** to somewhere else. Linux treats devices as files, and lets us read and write to them just as we would to a file. Devices have names like /dev/cdrom or /dev/hda1, but there is a special device known as /dev/null. /dev/null is like a black hole, swallowing anything written to it never to be seen again. Similarly, reading from /dev/null you get nothing, or a stream of nothings. Not the number 0, but the character that has the value 0 and is known as null. To redirect the error messages but keep the data output, we leave data stream 1 (**STDOUT**) alone and intercept number 2 (**STDERR**) with the directive 2>/dev/null.

```
jane@daisy > ~ $ find /etc -name network 2>/dev/null
/etc/rc.d/init.d/network
/etc/netprofile/profiles/default/services/network
/etc/sysconfig/network
jane@daisy > ~ $
```

Nice clean output.

When you don't know the exact name of the file that you are looking for, then you can use 'wild cards.' These are special characters that the shell interprets differently. The most common ones are:

- '\*' means 'substitute here zero or more unknown characters',
- '?' means 'substitute exactly one unknown character'

There are a lot more but we will cover these later when we get to 'regular expressions', These two suffice for most of our present needs.

## Linux – The Basics

Linux really only understands two things: files and processes. If that seems to be a rather bold and sweeping statement, then consider this: Whatever we do on a computer involves manipulating files. We create, delete, edit and rename them. We cut them up, join them together and search them for a particular piece of information. In short, we do all manner of things to files to get our desired result. We do this using processes.

There are several types of files.

- Regular files. These can be split into:
  - Data files such as text files, pictures, or music files
  - Executable scripts – lists of instructions, in human-readable language, to be executed sequentially (but often with some clever route planning that makes it difficult to follow).
  - Binaries – executable files in computer-readable form. These are the applications that we run and libraries of functions that the applications refer to. Their contents are mostly unintelligible to humans.
- Directories. Really just lists of files that may be scattered over one or more hard drives grouped together for human convenience.
- Links. Pointers to actual files. There may be many links to any one file.
- Special Files. These are used by Linux to communicate information to the system, and to interact with the hardware. These are mostly found in the /dev directory. In Linux, even your mouse is treated as a file.
- Sockets. We can ignore these for now.
- Named pipes. As above. These will also keep until later.

When we click on an executable file, or type its name on the command line, the kernel starts a process which will hopefully run until terminated, either on completion, or prematurely by the user. Each process is given an identity in the form of a number, known as the 'process id'. The process may, in turn, start any number of sub-processes.

When the initial process is terminated, then any unused processes associated with that process are also terminated, and any system resources such as allocated memory and the process id, are released, and any open files are closed. Occasionally, things don't quite go according to plan and the system resources start to be in short supply, having the effect of slowing down the system, unless somebody intervenes. A system reboot would fix things, but that is not always convenient. This was designed as a multi-user system, and shutting down a large system would cause too much disruption. There are other ways to do things.

This is why we need to understand about files and processes.

## Links

Perhaps now is a good time to discuss links. There are two kinds of links: soft links, also known as symlinks (symbolic links), and hard links. A soft link is similar to a shortcut in Windows and is a pointer to a filename that may be in the same directory or, more commonly, buried deep in some other directory structure. This is a convenient way to access files without having to enter (or even know) the entire fully qualified address of the file. For example,

suppose that we have a file named contacts that resides several directories deep in your home directory, but you need to be able to access it easily from your home directory.

Let's set this up.

```
cd ~
mkdir -p mydir/personal/mycontacts
```

Here the -p option tells the mkdir command to make any parent directories as required.

**mv contacts mydir/personal/mycontacts** moves the contacts file we created previously into the new directory

```
jane@daisy > ~ $ cat contacts
cat: contacts: No such file or directory
jane@daisy > ~ $ █
```

The file cannot be read because we moved it to our new directory

**ln -s mydir/personal/mycontacts/contacts link-to-contacts** will create a soft (-s) link to that file and then it can be accessed through link-to-contacts.

**cat link-to-contacts** will display the contents of /home/jane/mydir1/personal/mycontacts/contacts.

```
jane@daisy > ~ $ cat link-to-contacts
john
amy
gustav
bob
glenn
simon
george
jane@daisy > ~ $ █
```

The syntax for the command `ln` is `ln {-s if a soft link} {what you want to link to} {name of the link}`

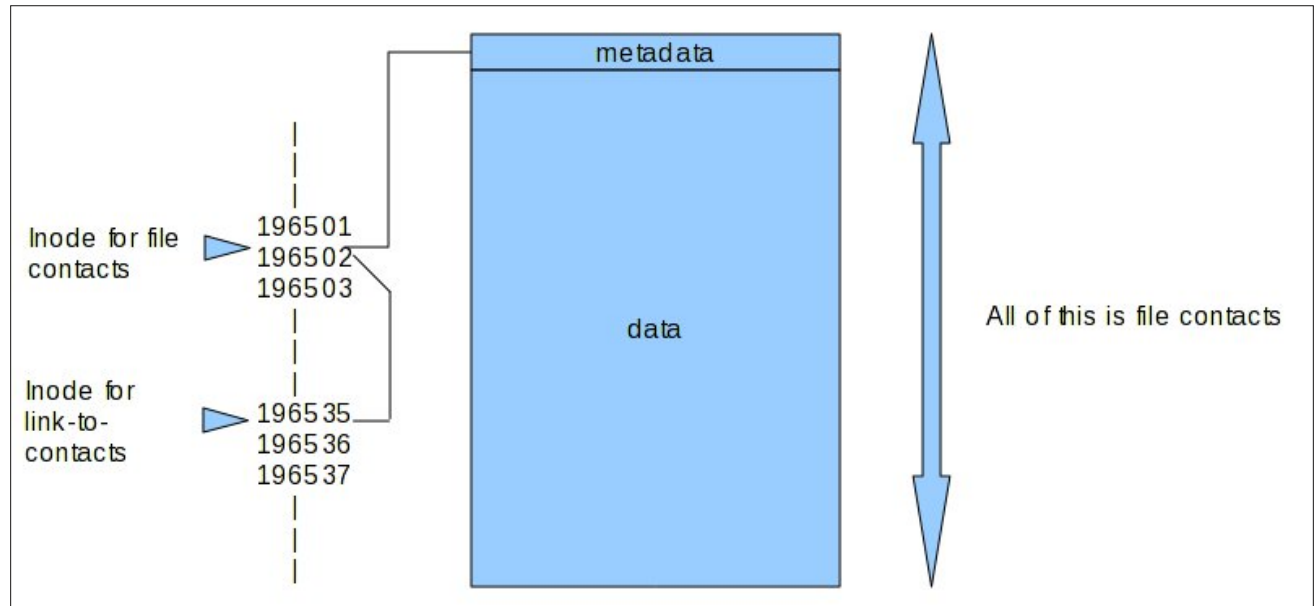
The system makes extensive use of symlinks, and any file may be linked to many times. If the original file is deleted from the directory `mycontacts`, then the link remains in my `/home` directory, but `cat contacts` now gives the message 'No such file or directory.' This is known as a broken link, and if we issue the command `ls -l`, we will see the output for that link listed in flashing red/white text (other distributions may use different colors).

Hard links don't point to the file name, but rather contain a reference to something known as an 'inode'. When a file is created, the file system allocates a number to it, an inode. This number points the file system to a set of meta data, or information about the file, it's permissions, it's name, and where the data is stored on the disk etc. Every inode on a partition is unique and knows only about one file, but the same inode number on a different partition or drive will reference a different set of meta data, and hence file. Think of a file as having two sections: the meta data part that is referenced by the inode and the data part that is referenced by the meta data. Normally, you don't need to know about inodes, as the filesystem does all that for you.

You can see these numbers if you issue the command

`ls -li`

```
jane@daisy > ~ $ ls -li
196503 contacts2          196494 Movies/         196500 newfile
196463 Desktop/         196458 Music/          196501 newfile2
196461 Documents/      196690 mydir/          196487 Pictures/
196536 link-to-contacts@ 196497 mydir1/         196486 tmp@
jane@daisy > ~ $
```



`cd mydir/personal/mycontacts`

```
jane@daisy > mycontacts $ ls -li
196502 contacts
jane@daisy > mycontacts $
```

A hard link is a bit like another name for the file, but it inherits the files DNA, as it were. Creating a hard link is the same as for a soft link, but without the `-s`. When a file is created, the number of links to the inode is set to one, and when a hard link is created, the count for the number of links to the inode is increased by one. When a hard link or 'the original file' is deleted, then the count is decreased by one. When, and only when, the link count reaches zero, the inode and storage for the meta data and data parts are released i.e. the file is deleted.

`ln ~/mydir/personal/mycontacts/contacts`

`contacts-link` creates a hard link named `contacts-link`

`ls -li`

```
jane@daisy > ~ $ ls -li
196503 contacts2          196690 mydir/
196502 contacts-link     196497 mydir1/
196463 Desktop/         196500 newfile
196461 Documents/      196501 newfile2
196536 link-to-contacts@ 196487 Pictures/
196494 Movies/         196486 tmp@
196458 Music/
jane@daisy > ~ $
```

Note that the inodes are the same for both the link and the file, i.e. 965021



```
jane@daisy > ~ $ cat contacts-link
john
amy
gustav
bob
glenn
simon
george
jane@daisy > ~ $ █
```

### cat contacts-link

**rm mydir/personal/mycontacts/contacts** deletes the original file.

```
jane@daisy > ~ $ rm mydir/personal/mycontacts/contacts
rm: remove regular file `mydir/personal/mycontacts/contacts'?
y
jane@daisy > ~ $ █
```

### cat contacts-link

```
jane@daisy > ~ $ cat contacts-link
john
amy
gustav
bob
glenn
simon
george
jane@daisy > ~ $ █
```

Although the file has been deleted, the link still points to the inode, and can thus access the data which has not yet been deleted. Now while this may be seen as a security nightmare, it does have the advantage of allowing important files to be accessed by unsafe hands, while being comforted by the

knowledge that the link to the file data can be easily reconstructed and no actual data loss need occur.

## Permissions & Groups

Unix, from which Linux was developed, was designed as a multi-user system, and a method was needed to determine who had access to which files. There are some files that are private, some that other users need access to, and some that may be made public. Also, the level of access needs to be considered: are users allowed to modify or delete the file, or if the file is executable, who may execute it?

By default, when a user creates a file, they are known as the 'owner' of that file. It belongs to the users primary group, but this can be changed. Some users create files that they need to allow a group of other users to access, but deny that access to others. Permissions were defined in three levels:

- Read permission
- Write permission
- Execute permission (In the case of a directory you may change to it.)

Each of these permissions are applied or removed for:

- The owner
- The group
- Everybody else

For regular files, the permissions are fairly obvious. For a directory, read permission means that you may

```
jane@daisy > ~ $ ll
total 48
-rw-r--r-- 1 jane jane 44 Nov 10 12:37 contacts2
-rw-r--r-- 2 jane jane 39 Nov 10 12:33 contacts-link
drwxr-xr-x 3 jane jane 4096 Nov 16 08:34 Desktop/
drwxr-xr-x 2 jane jane 4096 Nov 10 22:46 Documents/
drwxr-xr-x 2 jane jane 4096 Feb 26 2007 Movies/
drwxr-xr-x 2 jane jane 4096 Feb 25 2007 Music/
drwxr-xr-x 3 jane jane 4096 Nov 10 20:45 mydir/
drwxr-xr-x 2 jane jane 4096 Nov 10 12:06 mydir1/
-rw-r--r-- 1 jane jane 69 Nov 10 12:10 newfile
-rw-r--r-- 1 jane jane 2174 Nov 10 12:12 newfile2
drwxr-xr-x 2 jane jane 4096 Nov 11 00:38 Pictures/
-rw-r--r-- 1 root root 3084 Nov 13 06:56 test.txt
lrwxrwxrwx 1 jane jane 5 Nov 12 12:39 tmp -> /tmp/
```

list the contents (file or directory names), write permission means that you may create, delete or rename files in the directory, and execute permission means that you may 'cd' – change to that directory.

If we look at a directory listing using the command **ls -l**:

The first position on the left indicates the file type and this can be any of the following:

- - regular file or hard link
- d directory
- l symbolic link
- p named pipe
- s socket
- c character device
- b block device

The next nine positions indicate the files permissions. The first three are the user permissions, the second three the group permissions, and the last three are the other, or world, permissions. The read, write and execute permissions are displayed like this:

user	group	other
r w x	r - -	r - -

Here the user may read, modify and execute the file while others may look at the contents only. The read and write permissions are represented by 'r' and 'w', but the execute permission may be 'x', 's' or 't'.

- x the normal execute permission
- s suid – set user id
- t The 'sticky' bit

The last two are rarely needed by regular users so we can skip them for now. Permissions may be changed by root or by the owner of the file and the command to do this is

### **chmod {option} {permissions} {file or directory name}**

What you put in the permissions part of the above statement may be done in two ways.

You can use 'u', 'g' 'o' and 'a' (all) to specify which set of permissions to change, 'r', 'w' and 'x' for the permission and '+', '-' or '=' to set or unset the permissions. Additionally, you may combine these to change more than one. Omitting u, g or o sets or unsets the specified permission in all positions.

Examples: If myfile has permissions rw- r-- r--

**chmod g+w** adds write permission for the group, i.e. rw- rw- r--

**chmod o+w** adds write permission for others, i.e. rw- rw- rw-

**chmod +x** adds execute permission for everybody, i.e. rwx rwx r-x

**chmod ug-w** removes write permission for the user and group, i.e. r-x r-x r-x

**chmod ugo=rw** sets permission to read write for everyone, i.e. rw-rw-rw-

The other way to specify permissions is with numbers. Numerically permissions are set like this:

read=4, write=2 and execute=1.

This is expressed in something known as 'octal' (counting in eights instead of tens), but we don't need to understand that here to use it. Instead of 'r', 'w', 'x' we use '4', '2' and '1'. To combine them, we add them up so that rw = 4 + 2 = 6.

To apply this to the three groups, we use three of the sums so that 'rwx rw- r--' becomes (4+2+1) (4+2+0) (4+0+0) = 764.

This is also sometimes expressed as 0764 – don't worry about the leading zero, it's an 'octal' thing and, for our purposes, can be included or not.

So to set permissions to rw- rw- r-- we would use the command **chmod 664 myfile1**.

Looking again at the directory listing, after the permissions, is a number that is the number of links

or references to that file. After that, we have the name of the owner of the file, and then the name of the group that the file belongs to.

To get a list of all groups, look at /etc/group

### **cat /etc/group**

To find out which groups somebody belongs to, use the command

### **groups username**

To add a new group, you need root privileges and the command 'groupadd'

### **su (enter root password) groupadd friends**

adds the new group 'friends' to the /etc/group file.

To add users to groups use the command usermod the -a option means append.

```
jane@daisy > $ groups john
john : john lp floppy cdrom cdwriter audio video users lpadmin polkituser dialout
```

### **usermod -aG friends jane (Note uppercase G) usermod -aG friends john**

Adds jane and john to the group friends. Becomes effective at the next login.

To make a file accessible to members of a group we can use the command 'chown' – change ownerships.

### **chown jane:friends contacts2**

This keeps jane as the owner of the file but changes group membership.

```
jane@daisy > ~ $ su
Password:
[root@daisy jane]# groupadd friends
[root@daisy jane]# usermod -aG friends jane
[root@daisy jane]# usermod -aG friends john
```

As we wanted to change only the group could have also done

**chown :friends contacts2**

Groups are deleted with the command

**groupdel groupname**

```
[root@daisy jane]# groupdel friends
[root@daisy jane]# ls -l contacts2
-rw-rw---- 1 jane 502 44 Nov 10 12:37 contacts2
```

This leaves all files that belonged to the deleted group 'orphaned'. Cleaning up this mess is up to you.

```
jane@daisy > ~ $ groups
jane lp floppy cdrom polkituser cdwriter audio video
users lpadmin dialout friends
jane@daisy > ~ $ ls -l contacts2
-rw-r--r-- 1 jane jane 44 Nov 10 12:37 contacts2
jane@daisy > ~ $ chown :friends contacts2
jane@daisy > ~ $ ls -l contacts2
-rw-r--r-- 1 jane friends 44 Nov 10 12:37 contacts2
jane@daisy > ~ $ chmod 660 contacts2
jane@daisy > ~ $ ls -l contacts2
-rw-rw---- 1 jane friends 44 Nov 10 12:37 contacts2
jane@daisy > ~ $ su john
Password:
john@daisy > jane $ groups
john lp floppy cdrom polkituser cdwriter audio video
users lpadmin dialout friends
```

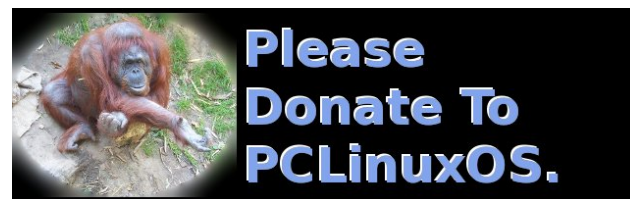
The group has been deleted from the /etc/group file, and now the file shows as belonging to the indeterminate group 502. This known as the group id or guid and we can use this to tidy things up. Use the find command to locate all files that have a guid of 502.

```
jane@daisy > ~ $ su
Password:
[root@daisy jane]# find /home -gid 502
/home/jane/contacts2
```

We found, as expected, only the one file that we changed. but even if there were many files it would be a simple matter to

**chown :jane filename**

If however, there were hundreds of files, then it would be rather daunting to manually change them all. To do this we might use a loop. We'll discuss this shortly.



**International Community PCLinuxOS Sites**



# Command Line Interface Intro: Part 4

by Peter Kelly (critter)

## Processes

When you start an executable file, either by clicking on its icon or by entering its name in a terminal, you are actually starting what Linux knows as a process. This may, in turn, start other processes, or series of processes. Other processes are started during the boot up sequence, or automatically as required by the system.

A process is what acts on files and the data that those files contain. As Linux is a multitasking, multi-user operating system, you and other users on the system may start many applications that have their own sets of processes. Obviously, Linux has to have a method of controlling and keeping track of all these processes.

Each process is given a unique identity when it is started, known as the process id number, or PID, is allocated space in memory and assigned certain other system resources that needn't concern us here. When booting the system, the first process to be started is always `init`, and is given the PID number 1. To get a graphical representation of the processes currently active on your system type in a console:

### `ps tree -p`

The `-p` option shows the PIDs.

The initial process started by an application is known as the **parent**, and any processes spawned from that process are known as **child processes**. Modern systems use threads, or Light Weight Processes (LWP), as child processes where practical, which share some of the resources of the parent. Communication between the system and processes is done by signals, and the whole show is orchestrated by the scheduler.

Usually all this happens under the hood, and we don't have to bother about it. But just occasionally, something goes wrong, the system starts to misbehave, and user intervention is required. This is where familiarity with the command line comes in.



Open a terminal and type:

```
glxgears > /dev/null &
```

This a fairly resource intensive application (usually used for benchmarking systems) that we don't want to see the output from in this case, so

we dump it to nowhere. The `&` character puts this process running in the background so that we get control of our terminal back. More about background tasks in a moment.

Now type:

```
top
```

```
top - 07:19:05 up 25 min, 1 user, load average: 1.00, 1.01, 0.72
Tasks: 90 total, 3 running, 87 sleeping, 0 stopped, 0 zombie
Cpu(s): 95.0%us, 5.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%st
Mem: 1035628k total, 332676k used, 702952k free, 63696k buffers
Swap: 533192k total, 0k used, 533192k free, 169172k cached

  PID USER      PR  NI  VIRT  RES  SHR  S %CPU  %MEM    TIME+  COMMAND
 2766 root        20   0 49168  23m 4412 R 96.6   2.3  12:56.16 X
 8534 jane        20   0 3780  1320 1040 S  1.3   0.1   0:13.66 glxgears
    1 root         0   0  1572   536  472 S  0.0   0.1   0:01.60 init
```

That's a lot of information! Don't worry, you don't need all of this. Look at the line representing the `glxgears` application (the next to last line in the screen shot). The first column is labeled PID and has a value of 8534, that's what we need.

Suppose this application refused to respond and you couldn't close it. In `top` type `k` and you will be prompted "PID to kill:" Enter 8534. This gives the prompt **Kill PID 8534 with signal [15]:**

If you enter `n` here, the command is canceled. Enter `y`, and the application is sent the default signal 15. Signal 15 is known as SIGTERM, and is the normal termination signal. On receipt of this signal, the application will close gracefully, handing back any resources to the system, kind of like asking somebody to 'please leave'. Occasionally, you meet a troublemaker of a process that just will not go. Then, you can use the strongest signal of all, signal 9, known as SIGKILL. This is more of a bullet in the head approach – sure it works, but it might just make more trouble in the long run, and is best avoided if possible.

There are lots of signals available, but those are the only two that you are likely to have to issue manually. `top` is a very comprehensive program that uses single letter commands. `h` brings up the help screen, `A` for the alternate display using field groups, `z` toggles the color of the current group, and `W`



saves your settings. There is so much more to this utility that you really do need to use the 'h' (help) command to get the most out of it. Remember that Linux is very literal, and uppercase or lowercase commands give very different results.

```

jane@home ~$ ps ux | grep glxgears
jane  11730  1.6  0.1  3780 1316 pts/1    S    07:10   0:02 glxgears
jane  11931  0.0  0.0   2996   704 pts/1    S+   07:13   0:00 grep --color glxgears
jane@home ~$
    
```

```

jane@home ~$ ps ux | grep glxgears
jane  11730  1.6  0.1  3780 1316 pts/1    S    07:10   0:02 glxgears
jane  11931  0.0  0.0   2996   704 pts/1    S+   07:13   0:00 grep --color glxgears
jane@home ~$
    
```

This gives us two results. The first one, with the PID of 11730, is the one that we are looking for. The second one is the grep command we used to filter the results. Why grep `---color`? Where did that come from? PCLinuxOS provides an alias for the grep command so that matches are highlighted in color (or, if you live in the UK, they will be in full glorious colour). Type `alias | grep grep` on the command line to see it.

```

jane@home ~$ alias | grep grep
alias grep='grep --color'
jane@home ~$
    
```

The grep command has also filtered out the header line, as that did not contain the expression 'glxgears'.

To get rid of the errant process, we use the kill command:

### kill 11730

We could specify a signal, as in `kill -s 9 11730`, but I think that the default, unstated signal 15 is powerful enough magic.

A signal may be sent by name or by number. If you want the complete list of signal names and numbers type

### kill -l

You may only terminate any process that you own,

so be especially careful when working with root privileges.

When working in a GUI, if you want to start another application, then clicking on its icon will open it in a new window. For command line work, you can open another terminal emulator. But what if you are locked into a single terminal, as you may be if the system has crashed? You may occasionally start a process which takes a long time to complete and need to execute another command.

One way around this is to follow the command with an ampersand (&), as we did with the glxgears application. This puts the process into the background and returns control of the terminal to the user. If the process is already running, it can be put into the background using the command **control + z**, which suspends the process running in the foreground, and then use the command **bg**, which causes the process to resume execution in the background.

To list the processes running in the current shell, use the command **jobs**.

```

jane@home ~$ glxgears > /dev/null &
[1] 8319
jane@home ~$ jobs
[1]+  Running                  glxgears >/dev/null &
jane@home ~$
    
```

Each job started in the shell is given a number, which is displayed in square brackets, along with the PID. Here, job 1 has PID 8319. The jobs command displays the job number, its status (running, Stopped or Terminated), and the command that initiated the process. To kill a job, we use the **kill** command, like this:

**kill %N && fg** where N is the job number.

The **&&** part is used to tell the shell to wait until the kill command has finished, and then to move to the foreground. If we don't add that bit, the the job will be terminated, but not removed from the job list. Just a bit of tidying up.

I know this all sounds rather complicated but it can all be summarized like this.

**Command &:** Start a job in the background

**Control + z:** Suspend the job currently running in the background

**bg N:** Continue suspended job N in the background

**fg N:** Move suspended job N to the foreground

**jobs:** List all jobs

**kill %N && fg:** Kill job N

For the glxgears example, we dumped the output to /dev/null. When a job is running in the background, it will still produce output, which it will happily spew out to the screen as you are trying to work on another command. So, it is usually a good idea to redirect the output of a back-grounded job to a log file, or similar, and to dump any error messages.

e.g., **find /usr -iname \*.ogg > musicfiles 2>/dev/null &**

This will put the names of any files found into the file 'musicfiles', ignoring the case of the file name, and discard any error messages, such as trying to enter directories for which you don't have access. If you don't want to keep any output at all from the application, then there is a special construct that will grab everything, and send it to where ever you like, usually /dev/null.

**command 2>&1 /dev/null**

All this means is append **STDERR** to **STDOUT**, and send them both to the same place. Don't worry if this doesn't seem very intuitive. It is a very commonly used expression, but you would be amazed at how many experienced users who use it don't understand it!

## Backing up and Archiving

The problem with data backup is that most people don't do it. They may mean to, they may forget, they may find it too complicated, or they may not do it regularly enough. Even those who doggedly back up their data regularly, rarely bother to check that the data can actually be restored, which rather defeats the object. If you are one of the very small majority who don't fall into this category, then you can skip this next part.

For the rest of us, there is some good news. There is some software available that will do all our backups for us, will never forget, and will check the integrity of the backup automatically. And best of all, it is free! It's called Linux.

Before starting a backup strategy, you should consider what you want to backup, how often, and to where. There are various types of backup.

**Full** - What it says

**Differential** - backup only what has changed since the last full backup

**Incremental** - backup only what has changed since the last backup of any type

**Clone** - Usually used for the backup of entire file systems, mindlessly copying everything block by block

**Synchronized** - An exact copy of all data in a location, optionally deleting data in the target destination that no longer exists in the source location.

Each has it's pros and cons, and there are many dedicated applications that will do backups in any way that you want. The Linux/Unix ethos is to use multiple small applications to perform more complex tasks, and using this, we can produce a tailor made backup system and automate it. First off then, let's take a look at some of the applications that are available. All of the following are either installed by default, or available in the PCLinuxOS repositories.

**cp:** Yes, the simple copy command, when used with the -a (archive) option, is a simple (but not very efficient) backup utility.

**tar**: One of the oldest utilities, its name means 'tape archive', and it simply takes a list of files and lumps them all together. For this reason, the resulting file is often referred to as a 'tarball'. It is often used in conjunction with a compression utility, like gzip or bzip2, and it can do this automatically. To create a compressed archive of all files in the current directory (here I assume that jane is in her Pictures directory) and write it to a folder in janes home directory named junk, I might use the command

**tar -czf ~/junk/mypics.tar.gz \***

**-c** create an archive

**-z** filter the output through gzip to compress it

**-f** use the following file name for the archive

Although Linux doesn't need dos-like file name extensions such as .exe or .zip to determine the file type (the information is in the meta data pointed to by the files inode), it is helpful and traditional to do so with the tar command. If I had substituted **-j** for **-z**, then tar would have used the bzip2 utility. Then, the extension .tar.bz is usually used.

To list the files in an archive use:

**tar -tvf mypics.tar.gz**

To extract the files use:

**tar -xf mypics.tar.gz {list of file names}**

If no file names are given, then all files are extracted. Tar is really only suitable for backing up a small number of files at a time. For large or full backups, there are better tools available. When using compression utilities, be aware that they are not very fault tolerant, and the loss of even one byte can render the entire archive unreadable. This is a very real danger when using media such as floppies or CDs.

```
jane@daisy > ~ $ cd Pictures/
jane@daisy > Pictures $ tar -czf ~/junk/mypics.tar.gz *
jane@daisy > Pictures $ cd ~/junk/
jane@daisy > junk $ ls -l
total 44
-rw-r--r-- 1 jane jane 42933 Dec 6 11:35 mypics.tar.gz
jane@daisy > junk $ tar -tvf mypics.tar.gz
-rw-rw-r-- jane/jane 13165 2009-12-01 13:23 snapshot1.png
-rw-rw-r-- jane/jane 7460 2009-12-01 13:22 snapshot2.png
-rw-rw-r-- jane/jane 2511 2009-09-05 18:39 snapshot3.png
-rw-rw-r-- jane/jane 2481 2009-09-05 18:58 snapshot4.png
-rw-rw-r-- jane/jane 4059 2009-09-05 19:08 snapshot5.png
-rw-rw-r-- jane/jane 8435 2009-09-05 19:39 snapshot6.png
-rw-rw-r-- jane/jane 7138 2009-12-01 12:33 snapshot7.png
jane@daisy > junk $ tar -xf mypics.tar.gz snapshot5.png
jane@daisy > junk $ ls -l
total 48
-rw-r--r-- 1 jane jane 42933 Dec 6 11:35 mypics.tar.gz
-rw-r--r-- 1 jane jane 4059 Sep 5 19:08 snapshot5.png
jane@daisy > junk $
```

**dd** falls into the 'clone' category, and is an extremely useful command to know how to use.

There are lots of options for this command, but most of the time you can get by with just these four:

**if={name of the input file}**

**of={name of the output file}**

**bs={block size}**

**count={number of blocks to transfer}**

Notice the = sign in the options. This is mandatory. The important thing to remember when using dd is not to mix up the input file and the output file, as you will get no prompt from dd – just blind obedience. So, if you wanted to copy a partition to an empty partition and you mixed them up, then you would overwrite the first partition with blank data from the empty partition losing all data, no questions asked.

To copy a single file:

**dd if=/home/jane/contacts2 of=/home/jane/junk/contacts**

```
jane@daisy > ~ $ dd if=/home/jane/contacts2 of=/home/jane/junk/contacts
0+1 records in
0+1 records out
60 bytes (60 B) copied, 0.000218269 seconds, 275 kB/s
jane@daisy > ~ $ ls -l junk/
total 52
-rw-r--r-- 1 jane jane 60 Dec 6 11:43 contacts
-rw-r--r-- 1 jane jane 42933 Dec 6 11:35 mypics.tar.gz
-rw-r--r-- 1 jane jane 4059 Sep 5 19:08 snapshot5.png
jane@daisy > ~ $
```

To copy an entire partition (make sure that the destination is large enough):

**dd if=/dev/hda1 of=/dev/hdf1** to do a partition to partition copy

or

**dd if=/dev/hda1 of=/backup/hda1.img** will make an image file of the partition.

This way, entire file systems can be cloned and then restored with

**dd if=/backup/hda1.img of=/dev/hda1**

Here, /dev/hda1 should not be mounted, and not be part of the currently running operating system.

If you have a 1TB drive, then be prepared for a long wait.

The first 512 bytes of a hard disk contains the MBR, or Master Boot Record, and the Partition table. If this gets corrupted, then your system may not know how to boot up, or how your drive is partitioned. This can be fixed, if you are patient and have some in depth knowledge of file systems (or some understanding friends), but life can be so much easier if you have a backup of that data. Here dd is ideal. su to root, and type:

```
dd if=/dev/hda of=/backup/hda_mbr.img bs=512 count=1
```

```
jane@daisy > ~ $ su
Password:
[root@daisy jane]# dd if=/dev/hda of=/backup/hda_mbr.img bs=512 count=1
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000261765 seconds, 2.0 MB/s
[root@daisy jane]# cd /backup
[root@daisy backup]# ls -l
total 64
-rw-r--r-- 1 root root 512 Dec 6 12:07 hda_mbr.img
drwx----- 24 jane jane 4096 Dec 5 10:49 jane/
-rw-r--r-- 1 jane jane 52622 Dec 5 12:03 jane-log
[root@daisy backup]#
```

This means copy one block of 512 bytes from /dev/hda to the image file /backup/hda\_mbr.img.

(Note: no partition number is entered as the MBR relates to the whole drive, not to any particular partition).

Now if you do have problems, you can simply restore those 512 bytes. To restore the MBR, the drive should not be running the current operating system.

Boot up from the Live CD.

In a terminal, su to get root privileges.

Make a directory, e.g **mkdir /mnt/hda**,

Mount the drive there, **mount /dev/hda /mnt/hda**

Make sure that the image file is where you think that it is, i.e **/backup/hda\_mbr.img**, as running from the Live CD, the directory /backup may not exist.

```
dd if=/backup/hda_mbr.img of=/mnt/hda bs=512 count=1
```



Stop and re-read this command at least twice before pressing enter.

You should now be able to reboot the original system.

If you want to see what a MBR looks like, (it won't mean much) type:

```
cat /backup/hda_mbr.img
```

Then you will get a load of gibberish on the terminal, and the chances are that your prompt and anything you type on the command line are now gibberish. If that is the case then don't panic. Just type:

```
reset
```

and all will be well.

Mondo archive is an excellent and extremely reliable full system backup tool which is run from the command line, and includes a graphical interface in the terminal. There is an excellent step by step tutorial in the PCLinuxOS forums by catlord17 (<http://www.pclinuxos.com/forum/index.php/topic,59705.0.html>).

Rsync is one of my all time favorites. It is used to synchronize two sets of files, either locally or between remote machines, even if they are thousands of miles apart. It includes options for compression, can delete files on the destination that no longer exist in the source location, and can be used over ssh (secure shell). Best of all, it only transfers the differences between files on the source and destination, and if interrupted, will resume from the break point when restarted. To give you an idea of the power of this command, I maintain a copy of the PCLinuxOS repositories on a local hard drive, and automatically update it in the early hours, when network traffic is lightest. The full repository is currently around 17GB, but I download only what has changed and delete obsolete packages. The command to do this is quite long, but as it is automated, I don't have to remember it. If you're interested it looks like this:

```
rsync -aPvz --stats --delete --log-
file=/home/pete/update.log --exclude=SRPM*
ftp.heanet.ie::pub/pclinuxos/apt/pclinuxos/2007/
/data/repo/2007/
```

**partimage** is another command line utility that includes a graphical interface and is quite easy to use. It will make an image of an unmounted partition, using compression if required, and will only



copy used blocks. It allows you to include comments that can be useful when restoring. For example, you could include the size and type of file system of the original partition. It will also allow you to split up the image so that it may be saved to multiple CDs or DVDs.

## Creating a backup

Now that we have a few tools to play with, it is time to decide what to back up and where to save it. The size of the backup may determine where to back up to, but ideally, you want to back up to a different device than the one that holds the original data. If a hard drive dies, then you will be relieved that your back up didn't go with it. If you are doing a full system backup for disaster recovery, then mondo archive or partimage are probably the way to go. If you want to incrementally backup the files from a particular location that have changed over a set time period, then first you have to get a list of those files. The find utility includes an option to locate files based on the time they were last modified. To find all files in a directory that have changed in the last week

```
find /home/jane/ -mtime -7 ! -type d
```

**-mtime** looks for the file modification time.

Here I have excluded files of type directory with **!** **type d**.

The **!** means 'not', and must be preceded with a back slash to prevent the shell from interpreting it as a special character. This is known as 'escaping' the character.

For files changed in the last week use:

```
find /home/jane/ -mtime -1 ! -type d
```

If we redirect the output of this command to a file, then we can use that file to feed tar:

```
find /home/jane/ -mtime -1 ! -type d > /tmp/newfiles  
tar -cv -T /tmp/newfiles -f /backup/newfiles.tar
```

The **-T** option tells the utility to get its list of file names from the following file.

Reinstalling a Linux system these days is a relatively quick and trivial task, but getting everything back exactly as you like it can be more of a headache. All of your personal settings are stored in your `/home/{user_name}` folder. This is convenient, but has the disadvantage that as you own those files, then you can do whatever you like (or dislike) to them, and this is what new users usually do and experienced users, who should know better, still often do. This directory then is a prime candidate to have a backed up copy somewhere safe that is fully updated automatically and regularly. Sounds like a job for rsync.

First, we need to collect a bit of information.

It is necessary to preserve ownership, groups, links and modification times.

We want to back up `/home/jane` but not all of it.

We don't want to copy the contents of `junk/`, a folder jane uses to store temporary stuff.

There is a second hard drive mounted at `/backup`, and this is where we want our backup.

As this is going to be an automatic process, we should save the output of the program to a log.

No compression – data integrity is paramount, and the destination has several hundred GB free.

Obsolete files should be deleted.

The backup should be run daily, 1:30 AM, and also weekly at 3:00AM Sunday under a different name. This gives a week to have a change of mind about some settings. The odd times are chosen to be when the system is likely to be under a low load, and to avoid any conflicts due to daylight saving time changes.

Recovery should be a trivial task.

The command then should look like this:

```
rsync --archive --exclude=junk/* --delete --log-file=/backup/daily/jane-log /home/jane /backup/daily/
```

This must be all on one line. If your command includes more options and your editor can't cope then you will need to use the line continuation sequence

### Enter.

It works like this. When you reach the end of a line and press the enter key bash inserts an invisible character known as a 'newline'. If we type \ before pressing the enter key, then this newline character is 'escaped' or in effect ignored. The effect of this is that you can continue typing extremely long commands and bash will see it as one long line. I think the limit is 2048 characters, but I have never reached it, and I have typed in some pretty long commands!

Here, I have used the long version for all the options, as this is going to be saved to run automatically, and this format makes the command easier to follow in the future.

The trailing slash on directory names is important.

**/home/jane** backs up the directory jane and all the files and directories under it into the directory /backup/daily.

**/home/jane/** would not create the directory jane but would copy all of its contents into /backup/daily.

Similarly we want the directory junk/ to be created so that it is still available if we do a restore, but we don't want its contents. So, we exclude only the contents with junk/\*.

When you have built a long command, you don't want to have to retype it every time. You could create an alias but a better, and much more powerful method, is to include it in a script. Scripts can be very long and complicated, or short and to the point like this one.

Make two directories in the destination – here the destination is /backup:

```
mkdir /backup/{daily,weekly}
```

The curly braces include a comma separated list if directories to create.

Open your text editor of choice, and type in the following (copy and paste from the magazine doesn't always work as it carries over some of those 'invisible' characters that are used to typeset the article).

```
#!/bin/bash
#Script to back up my home directory
rsync --archive --exclude=junk/* --delete --log-
file=/backup/daily/jane-log /home/jane
/backup/daily/
```

Don't forget that the long rsync command must all be on one line.

I'll Explain all the weird stuff at the beginning when we get to more involved scripts.

Save it to your home directory under whatever file name takes your fancy (e.g. home-bkup). Having it

in your home directory has the advantage that it also gets backed up – self preservation.

Change the files permissions to make it executable.

```
jane@daisy > ~ $ chmod +x home-bkup
jane@daisy > ~ $ ls -l home-bkup
-rwxr-xr-x 1 jane jane 151 Dec  6 12:57 home-bkup*
jane@daisy > ~ $
```

Type ./home-bkup and the script should execute – check it out. ./ is needed to tell bash to look first here for the file, which is not where it would normally look for commands.

```
ls -al /backup/daily
```

```
jane@daisy > ~ $ ls -al /backup/daily
total 140
drwxr-xr-x  3 jane jane  4096 Dec  6 12:57 ./
drwxrwxrwx  4 root root  4096 Dec  6 12:53 /
drwx----- 26 jane jane  4096 Dec  6 12:57 jane/
-rw-r--r--  1 jane jane 124517 Dec  6 12:57 jane-log
jane@daisy > ~ $
```

Make a copy of the file with a new name for the weekly backup.

```
cp home-bkup home-bkup-weekly
```

Open this new file in an editor, and change the destinations in the command to read:

```
rsync --archive --exclude=junk/* --delete --log-
file=/backup/weekly/jane-log /home/jane
/backup/weekly/
```

Save the file.

Check that this one also worked. You should have a copy of your home directory and a log in each of the

two new directories that we created in **/backup/ – daily/ and weekly/**.

Now, to backup your home directory, all you need to do is to run that script. This, however, is where a lot of backup strategies fail, as you have to remember to run it. So let's automate it. To do this, we use a utility called cron. This utility is used by the system to run certain things at a particular time, like updating the database that is used by the locate command.

To set up cron, **su** to root. We need to use:

**crontab -e**

The **-e** puts us in edit mode in the vi editor.

The top line of the file shows the required format for the file. The first five columns are for when we want the command executed. An asterisk in any column means 'from the first to the last'. In the day of week column, **0** and **7** both represent Sunday.

So to execute our two scripts at 1:30AM every morning and 3AM Sunday respectively we add lines as shown.

Type **o** to open a new line below the top one and type in these two lines.

**30 1 \* \* \* /home/jane/home-bkup**

**0 3 \* \* 7 /home/jane/home-bkup-weekly**

Press the escape key, followed by a colon, and you

will get a prompt at the bottom of the screen. Type **wq** (write then quit) to save the file and then quit.

```
# min(0-59) hours(0-23) day(1-31) month(1-12) dow(0-7) command
30 1 * * * /home/jane/home-bkup
0 3 * * 7 /home/jane/home-bkup-weekly
:
:
:wq
```

Type **crontab -l** to see a listing of the file and check your typing.

Tomorrow you can check to see if it worked!



**Visit Us On IRC**

- Launch your favorite IRC Chat Client software (xchat, pidgin, kopete, etc.)
- Go to freenode.net
- Type `"/join #pclinuxos-mag"` (without the quotes)



# Command Line Interface Intro: Part 5

by Peter Kelly (critter)

## The Linux File System

The Linux file system is built like an upside down tree starting with the root directory `/`. Don't confuse this with the user name root. This is the root of the file system. From this root, grow branches, and from there grow other branches – ad infinitum. The first level of branches is mostly standard, although other distributions may add special directories, and you may also add some yourself. Where a file or directory is located within the file system is known as the path. The path of these first level directories is always `{directory name}` e.g. `/home`. As you make your way through the file system, every time you reach a new branch, like a fork in the road, you add another `/` so that you would refer to a particular file as

`/home/jane/mydir/myfile1`.

This file is three levels deep, and this way of referring to it is known as the absolute path. When jane is in her home directory `/home/jane`, she would refer to the same file as `mydir/myfile1`. This is known as the relative path, relative to where you are in the file system.

The standard PCLinuxOS file system looks like this:

<code>/</code>	The root of the file system
<code>  bin</code>	Commands for use by the users
<code>  boot</code>	Files used by the boot loader (grub)
<code>  dev</code>	Hardware is treated as files and here is where you can access them

<code>  etc</code>	Miscellaneous scripts and system configuration files
<code>  home</code>	Users home directories are in here
<code>    jane</code>	Janes home directory
<code>    john</code>	Johns home directory
<code>  initrd</code>	A special directory used by PCLinuxOS at boot time
<code>  lib</code>	Common libraries of routines used by various applications
<code>  media</code>	Where the system mounts external media (e.g. thumb drives)
<code>  mnt</code>	Other files systems are often mounted here
<code>  opt</code>	Additional 3rd party application software
<code>  proc</code>	memory resident file system that tracks the system state
<code>  root</code>	The super user's home folder
<code> /sbin</code>	System administration and privileged commands
<code>  sys</code>	A virtual file system akin to <code>/proc</code> storing dynamic kernel data
<code>  tmp</code>	Temporary system-wide storage area
<code>  usr</code>	Other installed applications, documentation and shared resources such as icons
<code>  var</code>	Various log files, mail and print spoolers etc.

Any distribution that you encounter will not deviate too much from this, although there may be some small changes

All of this takes no account of which hard drive or storage device any of these files are actually on. That depends upon where on the tree the file system that is resident on the device is mounted. Note that `/proc` & `/sys` do not exist on any hard drive. They are memory resident only. Try `du -sh /sys` to display the disk usage of the `/sys` directory.

## Mounting and unmounting file systems

When you boot the system, the device nominated as the root of the file system is given the path `/`. The installation process will have created the necessary folders at the first level, as shown above. If you elected to have a separate `/home` partition, then the `/home` directory will point to that partition. The partition is then said to be mounted at `/home`. You may add or remove additional devices as you see fit to anywhere on the file system. To mount a device on the file system, you need to provide certain information:

- The type of file system used by the device
- The mount point – where on the file system the device is to be mounted
- The device name or id
- Any options that control how the device is accessed and mounted

The type of the file system could be one of very many recognized by the system but the ones that you are most likely to encounter are

- **ext2** The second extended file system also known as the linux native file system. There was a first, `ext`, but it is no longer supported and shouldn't be used.
- **ext3** This is `ext2` with the addition of a journal. I'll explain journals in a moment.
- **ext4** The next stage in the development of the `ext` file system. This is still in the testing stage but usable if you want to experiment.



It should not however be used in 'mission critical' situations such as servers or for the boot partition of a system until your distribution approves it.

- **reiserfs** A popular journaled Linux file system
- **iso9660** Used on CDROM and DVD disks
- **vfat** Used to access Microsoft fat file systems.
- **ntfs-3g** The open source driver to access ntfs file systems used by Microsoft Windows.
- **nfs** Networking file system, not to be confused with ntfs.
- **swap** The linux swap file system type

The device name may be given in various ways,

**/dev/xxx** This is the traditional way

**LABEL={the partition label}** This can be used to simplify recognizing what is on a particular partition.

**UUID={Universally Unique Identifier}** This is the system that is currently used by PCLinuxOS in **/etc/fstab**

While UUID may be confusing to look at, it has advantages in multiple partitioned systems. If you are happy to let the system look after things, then this is best left alone. If you want to take control, then either of the other two methods might be a better bet.

The default options are usually ok but in certain cases you may need to specify others, the most usual ones being:

- **auto** or **noauto** to mount or not when the command **mount -a** (all) is issued or when the

system is booting.

- **user** to allow an ordinary user to mount the file system, only that user or root may unmount the file system.
- **users** allow all users to mount or unmount the file system
- **ro** or **rw** to mount the file system read only (e.g. a cdrom) or read write.

To use the mount command to manually mount devices, you usually need root permissions, and you would issue commands like these:

**mount -t auto /dev/cdrom /media/cdrom -o ro**

**-t** specifies the file system type, here we are requesting that the system recognizes the file system type automatically. If this option is omitted then the mount command will attempt to guess the file system type.

**-o** is the start of a list of options, separated by commas, that control the method of mounting the device.

**ro** read only is the only option used in this example.

**mount -t ntfs-3g /dev/sda1 /mnt/windows** to mount an ntfs formatted windows partition

```
[root@daisy jane]# fdisk -l
Disk /dev/hda: 8589 MB, 8589934592 bytes
16 heads, 63 sectors/track, 16644 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/hda1 *          1         8591     4329832+  83  Linux
/dev/hda2            8592     16644     4058712    5  Extended
/dev/hda5            8592     9649     533200+   82  Linux swap / Solaris
/dev/hda6            9650     16644     3525448+  83  Linux

Disk /dev/hdb: 8589 MB, 8589934592 bytes
16 heads, 63 sectors/track, 16644 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/hdb1 *          1     16644     8388544+  83  Linux
```

Results of **fdisk -l** command.

To discover which file system devices are attached to your system use the command **fdisk -l**.

To then find the label, uuid and file system type of a device you wish to mount issue the command **blkid {device name from fdisk -l command}**

```
[root@daisy jane]# blkid /dev/hdb1
/dev/hdb1: UUID="f8413aeb-25d1-41bd-8c7e-61eaad5d6e1a" SEC_TYPE="ext2" TYPE="ext3" LABEL="data"
```

If you need to use labels or uuid, then use **-L {label}** or **-U {uuid}**

```
[root@daisy jane]# mount -t ext3 -L data /data
[root@daisy jane]# df
Filesystem      Size  Used Avail Use% Mounted on
/dev/hda1       4.1G  2.4G  1.6G  61% /
/dev/hda6       3.4G   82M  3.3G   3% /home
data            451G  171G  281G  38% /mnt/host
/dev/hdb1       7.9G  1.6G  6.0G  21% /data
```

To create a label for an ext2, ext3 or ext4 file system use the command **tune2fs -L {label}**

**mount** on its own gives a list of all mounted file systems and their types, Adding **-l** will also show the labels. This information is actually the contents of the file **/etc/mtab**, which is one of the places the system keeps a list of mounted file systems. The other is **/proc/mounts**. Try **cat /proc/mounts**.

**mount -a** mounts all devices listed in **/etc/fstab**, except those with the option **noauto**

If a device is listed in **/etc/fstab**, then the mount command will take information from there and require you to supply only the device name or the mount point.

To remove a device from the file system, the command is **umount**. Notice the missing **n**.

**umount /dev/cdrom**

**umount -a** unmounts all file systems. A file system cannot be unmounted if a file or directory in it is being accessed. The root file system **/** of a running file system cannot be unmounted

Why bother with unmounting a file system? Well, this is all to do with keeping things in sync. Linux is a multi-user, multi-tasking system, and as such, has to use system resources like memory and processor time wisely. Unless instructed to the contrary, data is written to file systems asynchronously, i.e. not when the command is issued, but when the system deems it prudent to do so. Even if you are the only user on the system, you may have several applications that

periodically write to a file, like perhaps an autosave feature or a scheduled event like a backgrounded backup task. All of the data cannot be written at the same time, and so it is sidelined until it can be. Removing a floppy disk without first unmounting it, for example, might cause data loss and will confuse the device management system. The **umount** command synchronizes the file system before releasing it.

Journalled file systems were introduced to go some way towards protecting file systems against corruption when asynchronous data writing is used and the system suffers a catastrophic event, such as a power failure. Writing data is a multi-part operation, known as a transaction which involves the data, inodes, the directory entries and other metadata. If any part of this transaction is not completed when the system is brought down, then file system corruption occurs. On a large file system, it can take a while to rectify, walking through inode by inode, block by block. In a journalled file system, when a transaction is authorized, the processes involved in the transaction are recorded in a central area before any data writing is committed. If the transaction is incomplete when the crash occurs, then on reboot the journal is replayed, and the transaction is then completed. This makes recovery much quicker. An ext2 file system can be converted to ext3 without data loss by using the command **tune2fs -j /dev/xxxx**.

For a device like a CD drive, once the drive is unmounted, you can get the disk out with the following command:

**eject -T** (-T toggles the state of the drive tray

between open and closed, repeat the command to close the tray)

You could, of course, just push the eject button on the device. But if the PC is under the desk, you may find it useful to put a shortcut on your desktop to this command. **eject** should automatically unmount the volume if you have not already unmounted the volume. If you have an external floppy drive or an iPod (such as a second generation Nano), you must issue the **eject** command before disconnecting the device, or removing the floppy from the external drive. **eject** ensures that any data left in the buffers is flushed to the device, preventing corruption of the data on the diskette or device.

For those devices that need to be mounted at boot time, or are required to be regularly mounted or unmounted, the system keeps a look up table that provides this information. It is found in the **/etc** directory and is called **fstab**.

```
jane@daisy > ~ $ cat /etc/fstab
/dev/hda1 / ext3 defaults 1 1
/dev/hda6 /home ext3 defaults 1 2
none /proc proc defaults 0 0
/dev/hda5 swap swap defaults 0 0
none /dev/pts devpts mode=0620 0 0
```

The above is a fairly standard **fstab**. Look at the first line. It tells the system that when mounting **/dev/hda1**, it should be mounted to **/**, it uses the ext3 file system and the default options should be used to mount it. The last two numbers aren't relative to mounting partitions, but for completeness. The first of the two numbers is known as the dump number. Dump is a back up program, and checks

this number to decide whether or not to include this file system in a backup. Zero means no. The second of the two numbers is used to decide the order in which file systems should be checked using the fsck utilities. It is usual for the root file system to have a value of 1 here, and other file systems that are to be checked to have a value of 2. If a file system is to be skipped in a 'full' file system check, then it must have a value of 0 here.

## Working with partitions and file systems

One thing that causes a lot of people problems is the management of partitions. Not that there is anything inherently wrong with partitions, or that they are difficult to understand. Just sometimes they, or the configuration files used to access them, get screwed up.

A hard drive can be divided up into smaller chunks to separate data, or to house different operating systems. Initially, when the partitioning system was proposed and hard drives were small (just a few Megabytes. My first hdd was 40 MB Wow! All that space.), four partitions were deemed sufficient, but as hard drive sizes increased, a work around was found to allow more than four partitions. The original partitions were designated primary partitions, and if one of those was sacrificed and created as an extended partition, then this could be used as a container to house logical partitions. There can be only one extended partition in the partition table.

Linux has a lot of utilities for dealing with partitions,

from small cli-only utilities, to full blown graphical applications. PCLinuxOS Control Center uses the excellent diskdrake to provide a 'radically simple' graphical utility suitable for even the newest to Linux. We are going to use a cli-only utility called fdisk.

There several reasons to use fdisk. Messing about with partitions can be very dangerous, and it is all too easy in a graphical environment to get 'click happy' and wipe out a full system (although if a new file system hasn't been written to the device, it is still possible to recover). In a console, things tend to be more focused on the task at hand. A small utility like fdisk has fewer commands, and here simple is good. You will always find fdisk or something similar on any distribution.

We've already used **fdisk -l** to get a list of devices but if we now type

### fdisk {device}

Here, device is the device name without a partition number, as **fdisk** works with entire devices. So use **/dev/hda**, not **/dev/hda1**.

```
[root@daisy host]# fdisk /dev/hda
The number of cylinders for this disk is set to 16644.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
 1) software that runs at boot time (e.g., old versions of LILO)
 2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)
Command (m for help):
```

Typing **m (enter)** at any time will give us a list of available commands.

```
Command (m for help): m
Command action
 a  toggle a bootable flag
 b  edit bsd disklabel
 c  toggle the dos compatibility flag
 d  delete a partition
 l  list known partition types
 m  print this menu
 n  add a new partition
 o  create a new empty DOS partition table
 p  print the partition table
 q  quit without saving changes
 s  create a new empty Sun disklabel
 t  change a partition's system id
 u  change display/entry units
 v  verify the partition table
 w  write table to disk and exit
 x  extra functionality (experts only)

Command (m for help):
```

There are only sixteen commands available, including 'm' for the list of commands.

Linux uses DOS partition tables, so it is unlikely that you will ever need the 'b' and 's' commands, and for now I think that option 'x' should be avoided. That leaves us twelve commands with which we can destroy our system, but only one command will do that: the 'w' command. Until that command is issued, none of the changes that you have made are permanent. The 'q' command is our 'get out of jail free card.' Whenever we issue the 'q' command, we are returned to the system without committing any changes. Instantly and without fuss.

As for the remaining commands.

- **a** Some versions of the MSDOS/Windows boot loader would only expect to see one, and only

one, primary partition marked as boot-able, usually the first partition on the disk. Linux boot loaders grub and lilo just ignore this if set. If you are dual booting Windows and Linux, it does no harm to leave this as is.

- **c** Normally this flag is not set under Linux and if set can cause problems with overlapping partitions – best left alone.
- **d** Delete a partition
- **l** Print a list of all partition types known by fdisk
- **n** Create a new partition
- **o** Create a new partition table replacing the existing one – this removes all partitions on the device. Use with care.
- **p** Print the partition table to screen
- **t** Change the partition system id – this hexadecimal code tells the system what kind of file system to expect on the partition, e.g. 83 for ext2/ext3 file system
- **u** Switches the display units between cylinders and sectors
- **v** Verify the integrity of the partition table

You can experiment with these few commands until you feel comfortable, just don't use the 'w' command unless you mean it. 'q' will get you out.

```
Command (m for help): p
Disk /dev/hda: 8589 MB, 8589934592 bytes
16 heads, 63 sectors/track, 16644 cylinders
Units = cylinders of 1088 * 512 = 516096 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/hda1 *          1         8591    4329832+   83  Linux
/dev/hda2            8592     16644    4058712    5  Extended
/dev/hda5            8592      9649     533200+   82  Linux swap / Solaris
/dev/hda6            9650     16644    3525448+   83  Linux
```

If you look at the output of the 'p' command in the screen-shot, we can get a picture of the layout of the entire device – IDE hard drive no 1 in this case.

(NOTE: IDE is now called PATA (for Parallel ATA) to distinguish it from SATA, or Serial ATA. PATA and IDE are interchangeable terms when referring to the technology.) The drive is organized into a total of 16644 cylinders, and so all of the partitions must be created on cylinders 1 to 16644. If this is not the case, then you have a problem. Partition numbers 1 to 4 are reserved for primary partitions, whether or not they exist. Logical partitions are numbered from 5 upwards.

Partition 1 occupies cylinders 1 to 8591 and is a primary partition.

Partition 2 is also a primary partition created as an extended partition and occupies the rest of the disk so there can be no more primary partitions. This partition runs from cylinder 8592, the next available cylinder, to cylinder 16644 which is the end of the disk.

Partition 5 also starts on cylinder 8592 running up to cylinder 9649. This is the first logical partition.

Partition 6 occupies the remaining cylinders and is the second logical partition.

This all works very nicely, using all of the available space on the disk. Er! well no, not really. Let's

```
Command (m for help): u
Changing display/entry units to sectors
Command (m for help): p
Disk /dev/hda: 8589 MB, 8589934592 bytes
16 heads, 63 sectors/track, 16644 cylinders, total 16777216 sectors
Units = sectors of 1 * 512 = 512 bytes
Disk identifier: 0x00000000

   Device Boot      Start         End      Blocks   Id  System
/dev/hda1 *          3         8659727    4329832+   83  Linux
/dev/hda2            8659728    16777151    4058712    5  Extended
/dev/hda5            8659791     9726191     533200+   82  Linux swap / Solaris
/dev/hda6            9726255    16777151    3525448+   83  Linux
```

change the display units from cylinders to sectors with the 'u' command.

Then, we can see some ragged edges between starting and ending numbers. This is perfectly alright. Partitions must start on a cylinder boundary with the exception of partition 1.

Resizing partitions is a risky business. A partition is a container for a file system, so you could use fdisk to delete the partition from the partition table and then create a new partition. You would then need to resize the file system using a tool like **resize2fs**. Or, you could use a tool like gparted or diskdrake to do the job for you. In any event, it is a dangerous process, and not one one that I would be prepared to undertake without having a trusted, recent backup of all the data on that partition. If such a backup is available, then a better strategy would be to delete the partition, create a new partition, create a new file system on the new partition, and re-populate the partition from the backup. (NOTE: this latter method is also, typically, faster.)

Before a partition can be mounted it must contain a valid file system.

```
Device Boot      Start         End      Blocks   Id  System
/dev/hdb1 *          1         12190    6143728+   83  Linux
/dev/hdb2            12191     16644    2244816    5  Extended
/dev/hdb5            12191     16644    2244784+   83  Linux
[root@daisy host]# mount /dev/hdb5 /mnt/hdb5 -t ext3
mount: wrong fs type, bad option, bad superblock on /dev/hdb5,
missing codepage or helper program, or other error
In some cases useful info is found in syslog - try
dmesg | tail or so
```

The partition has been created with an id of 83, which tells the system to expect a file system of type 'Linux Native,' so we need to create one. The command for this is **mkfs**, and initializes or formats



the file system, laying out stuff like the directory tables and inode tables and setting up block sizes etc. Unlike partition table creation, this process overwrites existing data.

If you type **mkfs** into a terminal and then press tab you will see that there are many variations of the command. You can use the one that fits the file system type you want to create, or you can use the **mkfs** command with the **-t** option and supply a file system type.

**WARNING! Just make sure that the target partition is correct, as you will get no warning and all data on the partition will be overwritten!**

```
[root@daisy host]# mkfs.ext3 /dev/hdb5
mke2fs 1.41.6 (30-May-2009)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
140544 inodes, 561196 blocks
28059 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=578813952
18 block groups
32768 blocks per group, 32768 fragments per group
7808 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912

Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 38 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
```

Of course, there are a multitude of options that you can pass to the command to control exactly how the

file system is created, but in matters as fundamental as file systems, I prefer to accept the defaults. You can see from the output that automatic file system checking will take place at predetermined intervals, but if you want to do it yourself, then the drive should be unmounted. To check the root file system, it is easiest to reboot from a Live CD, and check the file system while running from the Live CD.

To check a file system, use the command **fsck**. Each file system has its own specific checking utility, and **fsck** is a 'front end' for these utilities. If the file system type is known, then it can be specified with the **-t** option, or **fsck** will look in **/etc/fstab** for it. Of course if you know the name of the correct utility you can use that directly.

**fsck -t ext3 /dev/hdb5** or

**fsck.ext3 /dev/hdb5**

Whether or not a disk is to be automatically checked is determined by the value in the sixth column of **/etc/fstab**. The frequency of checking ext file systems is determined when the file system is initialized, but this can be overridden with the utility **tune2fs -c** to change the total number of mounts.

Use **tune2fs -i** to change the interval in days. Add **w** or **m** to the number for weeks or months.

```
[root@daisy host]# fsck.ext3 /dev/hdb5
e2fsck 1.41.6 (30-May-2009)
/dev/hdb5: clean, 11/140544 files, 26061/561196 blocks
```

## Swapspace

This is a file system that always provokes a lot of discussion. It has its own set of tools, and is used by the system, not the users. One question that always crops up is "How much should I have?" The answer is always the same: "That depends."

Swap space was introduced in the days when memory was very expensive, and therefore very limited. When system load was high, it was possible to run extremely low on memory, which meant that the kernel had a busy time trying to juggle things around just keep the system running. This resulted in a slow and unresponsive system. With swap space, the system administrator could give the system some storage space to use as temporary memory. Because the data in memory had to be 'swapped' in and out of memory to the much slower hard drive space, it was always only a temporary solution, and if swap space was being frequently used, it was a sure sign that the installed memory was insufficient for the demands placed on the system.

Today, things have changed and modern systems have large amounts of memory. The rule of thumb used to be to have twice as much swap space as RAM. If you are limited to a small amount of memory, say 256MB, then 512MB of swap would be reasonable. If however your machine is fitted with 2GB or more, then twice that would be rather ridiculous, and could even slow down the machine. If

you do a lot of memory intensive tasks, like video editing, then you may benefit from more swap space. If you use a laptop that utilizes hibernation, then you will need a little more swap than RAM, as all the contents of RAM are copied to swap space when hibernation is entered, and copied back to RAM on resume. The thing to do is to monitor memory usage, either with a graphical system monitor, or with the command `free`. You can always add more.

Swap space can be in the form of a partition or a file. Partitions are the preferred method, as they tend to be somewhat faster, but a swap file can be a great temporary measure if the system suddenly finds itself low on ram and swap. To create a swap partition, you can use `fdisk` and create a partition of type id 82 (Linux swap).

To create a swap file is slightly more involved. You will need to create a file of the desired size, and then write data to it. To create a swap file of 128Mb (131072 blocks of 1024 bytes) called `myswap` in the root directory, and to fill it with null characters, use the command `dd if=/dev/zero of=/myswap bs=1024 count=131072`.

It is advisable at this stage to make sure that the system has actually written all the data to the file. The command to do this is simply `sync`.

This file then needs to be formatted/initialized to the swap file system.

**`mkswap /myswap 131072`**

Now, the file permissions need to be changed so that only the system has read/write access.

**`chmod 0600 /myswap`**

The file is now ready to be used, and can be added or removed with the commands `swapon /myswap` and `swapoff /myswap`

Do not delete the file without first removing it from swapspace with the `swapoff` command.

```
[root@daisy host]# dd if=/dev/zero of=/myswap bs=1024 count=131072
131072+0 records in
131072+0 records out
134217728 bytes (134 MB) copied, 3.68923 seconds, 36.4 MB/s
[root@daisy host]# mkswap /myswap 131072
Setting up swapspace version 1, size = 134213 kB
no label, UUID=de084bf7-71ca-4502-8742-9b1f9781d89a
[root@daisy host]# chmod 0600 /myswap
[root@daisy host]# ls -l /myswap
-rw----- 1 root root 134217728 Dec 30 10:39 /myswap
[root@daisy host]# free -m
              total        used         free      shared    buffers     cached
Mem:           1011          415          595           0          38         222
-/+ buffers/cache:    154          856
Swap:           520           0          520
[root@daisy host]# swapon /myswap
[root@daisy host]# free -m
              total        used         free      shared    buffers     cached
Mem:           1011          416          595           0          38         222
-/+ buffers/cache:    154          856
Swap:           648           0          648
```

If you want to make it permanent, then add an entry to the `/etc/fstab` file.

**`/myswap swap swap defaults 0 0`**

There are a couple of system directories that need a little more explanation, `/dev` and `/proc`.

## `/dev`

This is a strange directory, unlike most of the others that are full of files that are recognizable. Actually, this directory is full of files, device files. As already stated, Linux treats everything as a file, so device files are how we communicate with the systems hardware. They come in two distinct types: block devices that store data, and character devices that transfer data. A hard drive is a block device, and a keyboard is a character device.

In a long directory listing, `ls -l`, the first character of every line denotes the file type. `-` for a normal file, `d` for a directory and `l` for a link to another file. If we type `ls -l /dev`, we notice that apart from a few lines starting with `d` or `l`, the majority start with a `b` or `c` to denote a block or character device file. Every device on the system, and some that aren't, will be represented here, along with a few strange ones. We've already met `/dev/null` and `/dev/zero`.

If you look at the listing for the block and character devices, you'll notice something strange about the file size. It is given as two numbers, separated by a comma. This is not a file size, but the major and minor device numbers. The major number is specific to the type of device, and is used by the kernel to determine which driver to use to communicate with the device, while the minor number refers to a particular device. By example:

The first IDE/PATA channel on a PC can support up to 2 devices (master/slave), and is allocated block major 3. The first full device is then hda – block 3, 0. Partitions on the device are then numbered sequentially.

```
hda1      3, 1
hda2      3, 2
```

(and so forth).

The second device is hdb – block 3,64

```
hdb1      3,65
hdb2      3,66 etc.
```

The second ide channel (hdc & hdd) get major number 22

SCSI devices (including SATA drives and USB mass storage devices) start at major number 8, but partitions minor numbers repeat every 16, giving a maximum of 15 partitions per device.

Partitions on SCSI devices are numbered similar to the IDE/PATA devices, but are named as follows:

```
sda1      8, 1
sda2      8, 2
```

This will always be the case for systems equipped with SATA drives. In the case of USB mass storage drives, including external hard drives and memory card readers (SD/MMC, MemoryStick, CompactFlash, etc.), the device name will be **sdx**, where **x** is determined by the order in which the USB

mass storage devices are detected. For example, if a laptop has a SATA drive and a built-in card reader that reads and writes SD/MMC and MemoryStick cards, then these are named **sda** for the hard drive, and **sdb** for the card reader. Plug in an external hard drive, and that hard drive is named **sdc**.

### /proc

There is a wealth of information in here, and it reflects the entire state of the system. Unfortunately, it is not easy to find what you want, and when you do, you will probably be overwhelmed by the amount of detail. If you look inside this directory, you will see a lot of sub-directories with numbers for names. Each one of those contains all the details about a running process. The numbers are the process id, and the first one of these folders has the name 1, and contains information about process 1 - init, the first process to be run on boot up. If you poke about in these directories, you can find out things like the full command line that was used to invoke the process, it's current working directory and a whole bunch of other stuff that the kernel finds really cool, but is of little use to mortals. Get past these numbered directories, and things start to make more sense. If you want to know all about your processor, try `cat /proc/cpuinfo`. Want a list of modules? Use `cat /proc/modules`. Most of the files in here are read only, but some of them, notably many in the `/proc/sys` directory, are writable. Even though the effects of any changes only last until the next reboot, you should be careful in here.

For example, to temporarily change the machine's hostname you could type as root **echo newhostname > /proc/sys/kernel/hostname**. Start a new terminal to see the effect.



### Want To Help?

Would you like to help with the PCLinuxOS Magazine? Opportunities abound. So get involved!

You can write articles, help edit articles, serve as a "technical advisor" to insure articles are correct, create artwork, or help with the magazine's layout.

Join us on our [Google Group mailing list](#).

# Command Line Interface Intro: Part 6

by Peter Kelly (critter)

## Globbering

What? It's an unusual word that has its roots in the way command line interpreters used to handle things in the early days of Unix.

The bash shell recognizes certain characters as 'wild cards' that can be used to specify unknown or multiple occurrences of characters when specifying file names. We've already met some of these 'wild card' characters back in chapter 3, '\*' and '?'. These are the most common ones, but groups and classes of characters can also be used. When the shell encounters these 'wild cards' in a file name, it substitutes the *meaning* of the wild card at that position, a process known variously as “**file name expansion**”, “**path name expansion**”, “**shell expansion**” or “**globbing**”, depending on how pedantic or geeky you want to be.

The following can be used:

\* means match at this position *any one or more* characters

? means match at this position *exactly one* character

Individual characters can be grouped in square brackets:

[a,f] matches either **a** or **f**

[a-m,w-z] matches only a character in the ranges **a-m** or **w-z**

[a-z,0-9] matches any lowercase letter or any digit

[!a-c] matches any character that is **not** in the range **a-c**

[^a-c] same as above

```
jane@daisy > ~ $ ls -d [mM]*
Movies/ Music/ musicfiles mydir/ mydir1/
```

Another way of specifying groups of characters is to use **pre-defined classes**. These are groups of characters defined in the POSIX standard and the syntax is **[:class:]**.

The defined classes include:

[ :**alnum:** ] any alphanumeric character [0-9,a-z,A-Z]

[ :**alpha:** ] alphabetical characters [a-z,A-Z]

[ :**blank:** ] characters that don't print anything like spaces and tabs (also known as whitespace).

[ :**digit:** ] numeric digits [0-9]

[ :**punct:** ] punctuation characters

[ :**lower:** ] lowercase characters [a-z]

[ :**upper:** ] uppercase characters [A-Z]

[ :**xdigit:** ] any character that may form a part of a hexadecimal number [0-9,a-f,A-F]

So, **ls -d [[:upper:]]\*** will find all files and directories that start with an uppercase letter.

```
jane@daisy > ~ $ ls -d *[[[:digit:]]*
contacts2 mydir1/ newfile2
```

Note that two pairs of square braces are required here, one pair to define the start and end of a range and one pair to contain the class.

All of these methods may be combined to provide the file set that exactly matches your requirements. This method of file name expansion may reduce or increase the number of files to which your commands are applied.

The dot character '.' is not included in these expansions. Type **ls -al** in your home directory and you will see quite a few files that begin with this character. These are so called hidden files which is why we needed the **-a** option to the **ls** command to display them. The first two file names (in any directory) are the names of directories, '.' and '..', and these refer respectively to this directory and to the parent of this directory. Why do we need these? One reason is to be able to refer to a directory without specifying its name.

**cd ..** takes you up a level

**cd ../..** takes you up two levels and so on.

If you write a script and want to execute it, then it has to be on your **PATH** (a list of directories to be searched for executable files), or you have to supply the full absolute address for the file. Typing **./myscript** {this directory/myscript} is easier than **/home/jane/myscripts/myscript**.

For security reasons, it is inadvisable to add your home directory to your **PATH**.

If then, the dot character is not included in these expansions, how do we include hidden files if we want them to be a part of our list of files to operate on, but we don't want the two directory shortcuts '.' and '..'?



Suppose we want to rename all files in a directory with the extension '.bak', but some of these files are hidden 'dot' files? If we try to rename all files, including those beginning with a dot, then we will include . and .., which we didn't intend (you will probably get an error).

We could make two lists of files, one of normal files and one of dot files with the two unwanted files filtered out, or we could get the shell to do our dirty work for us. The bash shell has a lot of options and is started by default with those options set that your distribution settled on as being most useful for everyday use.

For a list of those options that are set (on) type **shopt -s**. The **shopt** command is used to display the status of **shell options**. **shopt -u** lists those that are unset.

The one that we are looking for here is called **dotglob**.

Setting this option on expands the dots that signify a hidden file, but ignore the two directory shortcuts. **shopt -s dotglob** turns it on, while **shopt -u dotglob** turns it off again. Don't forget to do this or you will remain in unfamiliar territory.

To rename all of the files, we need to use a loop. I will explain the mechanism of this when we get to shell scripting – the real power of the shell. For now just follow along.

**ls -al** shows 2 unwanted directory shortcuts and 5 files, 2 of them hidden dot files.

```
jane@daisy > bak $ ls -al
total 8
drwxr-xr-x  2 jane jane 4096 Jan 12 13:10 ./
drwx----- 30 jane jane 4096 Jan 12 13:40 ../
-rw-r--r--  1 jane jane   0 Jan 12 13:09 a1
-rw-r--r--  1 jane jane   0 Jan 12 13:10 a2
-rw-r--r--  1 jane jane   0 Jan 12 13:10 .a3
-rw-r--r--  1 jane jane   0 Jan 12 13:10 .a4
-rw-r--r--  1 jane jane   0 Jan 12 13:10 a5
```

```
jane@daisy > bak $ ls *
a1 a2 a5
```

The **\*\*** does not expand to show the hidden dot files or the directory shortcuts.

```
jane@daisy > bak $ shopt -s dotglob
jane@daisy > bak $ ls *
a1 a2 .a3 .a4 a5
```

By turning on the shell option **dotglob**, the wild card **\*\*** expands into all of the files but ignores the directory shortcuts.

We now can run our loop, and a check reveals that all files have been renamed:

```
jane@daisy > bak $ for f in *; do mv $f $f.new; done
jane@daisy > bak $ ls *
a1.new a2.new .a3.new .a4.new a5.new
```

Turning off the option reverts to the normal mode of dot files being unexpanded.

```
jane@daisy > bak $ shopt -u dotglob
jane@daisy > bak $ ls *
a1.new a2.new a5.new
```

The bash shell expansion is not limited to file names. There are six levels of expansion recognized by bash. They are, in alphabetical order:

**Arithmetic expansion.**  
**Brace expansion.**  
**File name expansion.**  
**History expansion**  
**Shell parameter expansion.**  
**Tilde expansion.**

If you are not at least aware of these, then you may find yourself inadvertently using them and then wondering why you are getting such weird results.

**Arithmetic expansion.** The shell can do limited integer-only arithmetic, its operators all have different meanings under different circumstances in the shell. They are:

- + Addition
- - Subtraction
- \* Multiplication
- / Division
- \*\* Exponentiation (The exponentiation operator is more usually '^' however this is used for negation in the shell expansion rules.)
- % Modulo (remainder)

The syntax is **\$(expression)**. Again, note the two sets of braces. Expressions may be nested within the first pair of braces and standard operator precedence is observed. Whitespace (e.g., spaces and tabs) has no meaning within the expression.

```
jane@daisy > ~ $ echo $((3+4))
7
jane@daisy > ~ $ echo $((($((3+4))/2))
3
```

3 + 4 = 7  
7 / 2 = 3    Integers only!

**Tilde expansion.** we may as well get this one out of the way now as the only way we are likely to use it is very simple.

We can use the tilde '~' as a shorthand way of referring to our home directory by using it as the first letter of an expression.

**cd ~** change to our home directory

**cd ~/mydir1** change to the sub-directory mydir in my home directory.

If we follow the tilde with the login name of another user then the command is relative to that users home directory. **cd ~john** change to johns home directory. No forward slash is required between the tilde and the users login name but you obviously still need the correct permissions to enter the directory if you are not the owner or a member of the directories group.

So is that all there is to tilde expansion. Of course not, this is Linux!

A system administrator might use it to assign expansions to commands by manipulating the directory stack – but you really didn't want to hear that did you?

**Brace expansion** is particularly good for a situation where a sequence of files or directories need to be created or listed.

The syntax is **{a,b,c}** or **{A..J}** or **{1..8}**.

For example, to create a set of directories to hold a years notes

```
jane@daisy ~ notes $ mkdir notes_{jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nove,dec}_2010
jane@daisy ~ notes $ ll
total 48
drwxr-xr-x 2 pete pete 4096 Jan 12 16:51 notes_apr_2010/
drwxr-xr-x 2 pete pete 4096 Jan 12 16:51 notes_aug_2010/
drwxr-xr-x 2 pete pete 4096 Jan 12 16:51 notes_dec_2010/
drwxr-xr-x 2 pete pete 4096 Jan 12 16:51 notes_feb_2010/
drwxr-xr-x 2 pete pete 4096 Jan 12 16:51 notes_jan_2010/
drwxr-xr-x 2 pete pete 4096 Jan 12 16:51 notes_jul_2010/
drwxr-xr-x 2 pete pete 4096 Jan 12 16:51 notes_jun_2010/
drwxr-xr-x 2 pete pete 4096 Jan 12 16:51 notes_mar_2010/
drwxr-xr-x 2 pete pete 4096 Jan 12 16:51 notes_may_2010/
drwxr-xr-x 2 pete pete 4096 Jan 12 16:51 notes_nove_2010/
drwxr-xr-x 2 pete pete 4096 Jan 12 16:51 notes_oct_2010/
drwxr-xr-x 2 pete pete 4096 Jan 12 16:51 notes_sep_2010/
```

You can use brace expansion wherever you need to put a set of items into a common expression.

**History expansion.** When you type the command **history**, you are presented with a numbered list of previously typed commands. Entering **!number** (where **number** is the number in the list) executes that command, this is history expansion at work. There is more to it than that, but it shouldn't bother us for the moment.

**Shell parameter expansion** is at its most powerful when used in shell scripts, but in its simplest form, it takes whatever follows a '\$' and expands it into its fullest form. What follows the \$ is often an **environment variable**, which may optionally be enclosed in braces i.e. **\${var}**, but it can be much more.

**Environment variables** are names given to things that need to be remembered in your current working environment. An example of this is where you currently are in the file system. To find this out you

might issue the command **pwd**. This information is stored in the environment variable named **PWD** (uppercase is customary and does help to distinguish variables from other text. Whatever the case of the name you will have to use it as Linux is case sensitive).

Typing the command **echo \$PWD** uses parameter expansion to the expand the variable name **PWD** to the full path of your current directory **before** passing it to the command echo.

To see a list of environment variables and their contents that are currently set up in your environment, type the command **env**.

Some you will recognize, and you can always add your own by use of the **export** command in your **.bashrc** file.

**export SCRIPTS\_DIR="/home/jane/scripts"**

After that, the command **cp my\_new\_script \$SCRIPTS\_DIR** will make sure that it goes to the correct place, providing of course that it exists..

With all these different ways that the shell can interpret what you type, we need some method of controlling what gets seen, as what and when. After all, although the bash shell is very powerful, we do want to be remain in control.

Control of shell expansion is exercised through the use of quotes. In Linux you will find four different kinds of quotes in use:

1. “ ” The decorative 66 – 99 style used in word processors – these are of no interest to us.
2. ” Standard double quotes
3. ' Single quotes
4. ` Back ticks, also known as the grave accent

The last three all produce different results in the bash shell.

**Double quotes** tell the shell to ignore the special meaning of any characters encountered between them and to treat them exactly literally, with the exception of \$, ` and \. This is useful if we want to pass the shell a file name containing an unusual character or space. However, because the \$ is still interpreted, variables and arithmetic will still be expanded. But **after** expansion, everything between the quotes will be passed to the command as a single word with every character and all white space intact. The interpretation of the back tick you will see in a moment, just remember that here it is preserved. The backslash allows us to escape the \$ or ` so that we may pass expressions such as “You owe me \ \$100”.

```
jane@daisy > ~ $ echo "you owe me \$100"
you owe me $100
jane@daisy > ~ $ echo "you owe me $100"
you owe me 00
```

See the difference?

**Single quotes** are the strongest form of quoting and suppress **all** expansion. Whatever you put between these gets passed on verbatim. No changes whatsoever.

```
jane@daisy > ~ $ echo 'You owe me $`expr 4 `* 25`'
You owe me $100
```

**Back ticks** evaluate and execute the contents and then pass the result to the command. Here **'You owe me \$'** is passed literally as it is enclosed in single quotes. The next part, **`expr 4 `\* 25`**, evaluates the expression  $4 * 25$  to be 100 before passing it to the echo command. The backslash is needed before the asterisk to escape its alternative meaning as a wild card.

All of this globbing and wild card stuff should not be confused with **regular expressions** (often abbreviated to **regexp**), even though they do share some common features.

**regular expressions** are used to manipulate and scan data for a particular pattern and is something much bigger.

You've already used regular expressions when we used the **grep** command. The command **grep** (global regular expression print, from the original line editor **ed** which used the command **g/re/p!** to achieve the same thing.) has two brothers known as **egrep** and **fgrep** (there's another brother known as **rgrep** but we don't see much of him).

We use the **grep** command to find a matching pattern of characters in a file, set of files or in a stream of data passed to the command. The general syntax is **grep {options} {pattern} {files}**.

It can be used directly as a command, or used as a filter to the output from some other command.

To find janes entry in the `/etc/passwd` file we could use either **grep jane /etc/passwd** or **cat /etc/passwd | grep jane**

```
jane@daisy > ~ $ cat /etc/passwd | grep jane
jane:x:500:500:jane:/home/jane:/bin/bash
```

In the above examples, `jane` is a pattern that **grep** tries to match.

Regular expressions are sequences of characters that the software uses to find a **particular pattern** in a **particular position** within the target data.

Why bother with regular expressions at all?

Linux uses plain text files for most of its configuration, for the output from commands and scripts and for reporting on system activity, security and potential problems. That's an awful lot of text, and to be able to search accurately for some particular information is one of the most important skills a command line user can master.

To use regular expressions we use three main tools: **grep** is used to search for a pattern

**sed** is a stream editor used to filter and manipulate data streams. This enables us to pass pre-processed data directly on to the next command, to a file or to **stdout/stderr**.

**awk** is a scripting/programming language used to automate pattern searches and data processing.

Many other commands, such as **tr** (translate), use regular expressions, but if you can get a grip on these three tools, then you are on your way to a higher level of command line usage. Before we go on to discuss **grep** and **sed** (I'll leave **awk** until we have done some programming in the

bash scripting section), we need a good basic understanding of regular expressions. Regular expressions are used by line based commands and will not match patterns spread over two or more lines. I just thought that you ought to know that.

In regular expressions certain characters have a special meaning and these are known as **meta characters**. For any meta character to be taken literally, and to lose its special meaning, it has to 'escaped', usually by preceding it with a backslash character, as we have done previously with wild card characters. The following are meta characters, but not all are recognized by all applications.

- `.` The dot character matches any single characters
- `*` The asterisk matches zero or more occurrences of the preceding character.
- `^` The caret is a tricky one, it has two meanings. Outside of square brackets, it means match the pattern only when it occurs at the beginning of the line, and this is known as an **anchor** mark. As the first character inside a pair of brackets it negates the match i.e. match anything except what follows.
- `$` Another **anchor** mark this time meaning to only match the pattern at the end of a line.
- `<<` `>>` More **anchor** marks. They match a pattern at the beginning `<<` or the end `>>` of a word.
- `\` The backslash is known as the escape or quoting character and is used to remove the special meaning of the character that immediately follows it.
- `[ ]` Brackets are used to hold groups, ranges and classes of characters. Here a group can be an individual character.

- `{n}` Match **n** occurrences of the preceding character or regular expression. Note that **n** may be a single number `{2,1}`, a range `{2,4}` or a minimum number `{2,1}` meaning at least two occurrences.
- `( \)` Any matched text between `(` and `)` is stored in a special temporary buffer. Up to nine such sequences can be saved and then later inserted using the commands `\1` to `\9`. An example will make this clearer.
- `+` Match at least one instance of the preceding character/regexp. This is an extended regexp (ERE) – see later.
- `?` Match zero or more instances of the preceding character/regexp. This is an extended regexp (ERE) – see later.
- `|` Match the preceding or following character/regexp. This is an extended regexp (ERE) – see later.
- `()` Used for grouping regular expressions in complex statements

With these we can find just about any pattern at any position of a line. Some of these meta characters do take on a different meaning when used to **replace** patterns.

### grep

First off **egrep** is exactly the same as **grep -E** and **fgrep** is exactly the same as **grep -F** so whats the difference?

**fgrep** uses fixed strings to match patterns, no regular expressions at all, and so it does actually work faster.

**egrep** is actually a more complete version of **grep**. There are two sets of meta characters recognized by regular expressions, known as **BRE** and **ERE**, or Basic regular expressions and Extended regular expressions. BRE is a subset of ERE. BRE is used by **grep**, and **egrep** uses ERE. BRE does not recognize the meta characters `+` `?` and `|`, and requires the `() {}` meta characters to be escaped with a backslash. For now, we'll stick to plain old **grep**. Just to set the record straight, **rgrep**, the other brother, is just **grep** that will re-curse down through directories to find a pattern match.

The **grep** command comes with a set of options that would make any Linux command proud. Here I'll go only through those options that normal people might use.

- A -B & -C followed by a number, print that number of lines of context After, Before or around the match – this helps to recognize things in a long text file.
- c Only output a count of the number of occurrences of a match for each file scanned.
- E Use the extended set of regular expressions (ERE). The same as using the command **egrep**.
- F Don't use regular expressions – treat all pattern characters literally. The same as **fgrep**.



- f filename** Use each line of the named file as a pattern to match.
- h** Don't output the filename when searching multiple files.
- i** Ignore case
- n** Output the line numbers of lines containing a match.
- r** Recurse through sub-directories.
- s** Suppress error messages
- v** Invert the match to select only non-matching files
- w** Match only complete words. A word is a contiguous block of letters, numbers and underscores.

That's enough theory for now, so let's go visit the family `grep` and do a few examples.

```
jane@daisy > ~ $ grep ja[n,y] /etc/passwd
jane:x:500:500:jane:/home/jane:/bin/bash
janet:x:502:502:janet:/home/janet:/bin/bash
```

If we were unsure how jane spelled her name (jane or jayne), then to search for her name in the `/etc/passwd` file we may be tempted to use the `**` wild card with `grep j*ne /etc/passwd` but this would fail, as the shell would expand `j*ne` before passing it to `grep`, which uses regular expressions to match the search pattern.

We could use `grep ja[n,y] /etc/passwd`.

This would however also match names such as janet.

```
jane@daisy > ~ $ grep -Ew 'jane|jayne' /etc/passwd
jane:x:500:500:jane:/home/jane:/bin/bash
jane@daisy > ~ $
```

To get around this, we could use the extended set of regular expressions available with the `-E` option or the `egrep` command. `grep -Ew 'jane|jayne' /etc/passwd` match either jane or jayne. The `-w` option matches only complete words. The quotes are needed to prevent the shell expanding the vertical bar symbol into a pipe. How many users have bash as their default shell?

`grep -c '/bin/bash' /etc/passwd`

```
jane@daisy > ~ $ grep -c '/bin/bash' /etc/passwd
4
jane@daisy > ~ $
```

To search for files in your home directory that contain a match, burrowing down through subdirectories and discarding warnings about inaccessible files, we could use a command such as `grep -rs glenn ~/*`

```
jane@daisy > ~ $ grep -rs glenn ~/*
/home/jane/contacts-link:glenn
/home/jane/mydir/personal/mycontacts/contacts:glenn
```

Multiple use of the `grep` command can simplify complex searches. To search for directories that begin with an uppercase or lowercase 'm', use `ls -l | grep ^[d] | grep [M,m]`. This matches all lines output from a long directory listing that begin with a 'd' (i.e., are directories). The output from this is then piped

through another `grep` command to get the final result.

```
jane@daisy > ~ $ ll | grep ^[d] | grep '[M,m]'
drwxr-xr-x 2 jane jane 4096 Feb 26 2007 Movies/
drwxr-xr-x 2 jane jane 4096 Feb 25 2007 Music/
drwxr-xr-x 3 jane jane 4096 Nov 10 20:45 mydir/
drwxr-xr-x 2 jane jane 4096 Nov 10 12:06 mydir1/
jane@daisy > ~ $
```

As you can see from the examples, `grep` is a powerful tool to find the information that you want from files or from a commands output. It is especially useful if you don't know where that information is, or even whether it exists at all, in the places that you are looking.

Sometimes you know exactly what information is available but you want only certain parts of it. Linux has a command that will help you to get exactly what you want.

## cut

The `cut` command is only really useful when you have tabulated data but as so many commands output data in that format it is a tool that is really worth knowing about and it is really simple to use.

When you examine a file of tabulated data, you'll see groups of characters separated by a common character, often a space, comma or colon. This common character is known as a delimiter, and the groups of characters are known as fields. The `cut` command searches each line of text looking for the delimiter, and numbering the fields as it does so.

When it reaches the end of the line, it outputs only those fields that have been requested. The general form of the **cut** command is:

**cut {options}{file}**

The most useful options are:

- c list** Select only characters in these positions
- d** Specify the delimiter. If omitted, the default is tab. Spaces need to be quoted -d" "
- f list** Select the following fields
- s** Suppress lines without delimiters

List is a sequence of numbers separated by a comma or, To specify a range, by a hyphen.

If we look at a line of data from the `/etc/passwd` file, we will notice that the various 'fields' are delimited by a colon `:`.

```
jane@daisy > ~ $ cat /etc/passwd | grep jane
jane:x:500:500:jane:/home/jane:/bin/bash
```

The first field contains the users login name and the fifth field, known for historical reasons as the **gecos** field (from **General Electric Comprehensive Operating System**), and contains the users real name and sometimes some optional location information, although PCLinuxOS doesn't use this additional information.

To extract the users login names and real names we use the command like this: **cut -d: -f1,5 /etc/passwd**.

This tells the command to look for a colon as a delimiter and to output fields 1 and 5.

All of this is fine for nicely ordered data as we find in the `/etc/passwd` file, but in the real world things don't always work out like that. Take for example the **ls -l** command. This outputs a nicely formatted long directory listing. The catch here is that to make the output look nice and neat, the **ls** command pads the output with extra spaces. When our **cut** command scans the line using a space as the delimiter it increases the field count each time it encounters a space and the output is, at best, unpredictable. Many Linux commands pad their output in this way. The **ps** commands output is another example of this.

```
jane@daisy > ~ $ ls -l
total 28
-rw-r--r-- 1 jane jane 0 Feb 7 05:34 contacts
drwxr-xr-x 3 jane jane 4096 Feb 7 05:25 Desktop/
drwxr-xr-x 2 jane jane 4096 Feb 7 2010 Documents/
drwxr-xr-x 2 jane jane 4096 Feb 7 2010 Movies/
```

If I wanted to extract the owner, size and file names from this listing it would be reasonable to assume that I needed fields 3,5 & 9 and that the delimiter is a space.

```
jane@daisy > ~ $ ls -l | cut -d" " -f3,5,9
jane Feb
jane 4096 05:25
jane 4096
```

As you can see, the output is not as expected. We could try the **-c** option, ignoring fields and counting the characters from the start of the line.

```
jane@daisy > ~ $ ls -l | cut -c14-18,24-27,41-
jane 0 contacts
jane 4096 Desktop/
jane 4096 Documents/
```

But apart from being tedious and error prone, if the directory listing changes slightly then the numbers will be different and the code is not re-usable, we would have to start over.

```
jane@daisy > ~ $ ls -l
total 32
-rw-r--r-- 1 jane jane 0 Feb 7 05:34 contacts
drwxr-xr-x 3 jane jane 4096 Feb 7 05:25 Desktop/
drwxr-xr-x 2 jane jane 4096 Feb 7 2010 Documents/
drwxr-xr-x 2 jane jane 4096 Feb 7 05:58 Friends/
```

```
jane@daisy > ~ $ ls -l | cut -c14-18,24-27,41-
jane 5:34 contacts
jane 5:25 Desktop/
jane 2010 Documents/
jane cts 5:58 Friends/
```

To work around this, we need to prepare the output from the **ls** command by filtering out the extra spaces. We can do this by using a command we have met once before. The **tr** command translates data, and we used it previously to change text from lowercase to uppercase. If we use the **tr** command with the **-s** option followed by a space, it **squeezes** repeated spaces out of the file or data stream that we feed it.

```
jane@daisy > ~ $ ls -l | tr -s " "
total 32
-rw-r--r-- 1 jane jane 0 Feb 7 05:34 contacts
drwxr-xr-x 3 jane jane 4096 Feb 7 05:25 Desktop/
drwxr-xr-x 2 jane jane 4096 Feb 7 2010 Documents/
drwxr-xr-x 2 jane jane 4096 Feb 7 05:58 Friends/
```

We can now cut out exactly the data the we want.

```
jane@daisy > ~ $ ls -l | tr -s " " | cut -d" " -f3,5,9
jane 0 contacts
jane 4096 Desktop/
jane 4096 Documents/
jane 4096 Friends/
```

Two other commands, rather simple but occasionally useful, so worth mentioning, are **paste** and **join**.

Typing the command name followed by **--help** will give you enough information to use these commands, but a simple example may better show their usefulness and their differences.

```
jane@daisy > Friends $ cat friends1
jane 12 High Street Newtown
john 12 High Street Newtown
janet 1432 Long Avenue Old town
jane@daisy > Friends $ cat friends2
jane age 22 height 5'-4"
john age 19 height 5'-10"
janet age 28 height 5'-5"
```

```
jane@daisy > Friends $ join friends1 friends2
jane 12 High Street Newtown age 22 height 5'-4"
john 12 High Street Newtown age 19 height 5'-10"
janet 1432 Long Avenue Old town age 28 height 5'-5"
```

Suppose we have two files containing different data about common things such as these:

The names in these files are common but the data is different. We can merge the data from both files with **join**.

With **paste** we can add data we cut from one file to the end of another file on a line by line basis

```
jane@daisy > Friends $ cut -f2-4 friends2 > friends3
jane@daisy > Friends $ paste friends1 friends3
jane 12 High Street Newtown age 22 height 5'-4"
john 12 High Street Newtown age 19 height 5'-10"
janet 1432 Long Avenue Old town age 28 height 5'-5"
```

## sort

When you have found the data that you want, cut out all but the required information, and joined or pasted the results, it may not be in the order that you

want. Here, Linux has an exceptionally powerful & quick utility to do just that.

The syntax of the **sort** command is **sort {options} {file(s)}**. If more than one file is supplied then the **combined** contents of the files will be sorted and output.

The options available for the sort command make for a rather comprehensive utility. These are the most useful ones:

- b Ignore leading blanks in the sort field
- c Only check whether the data is sorted but do not sort
- d Sort in dictionary order, consider only blanks and alphanumeric characters
- f Treat upper and lowercase as equal
- i Consider only printable characters. This is ignored if the **-d** option is specified.
- k Specify the sort field
- n Numeric sort
- r Reverse the sort order
- t Specify the field separator
- u Output only the first of one or more equal lines. If the **-c** option is specified, check that no lines are equal

A couple of these options need further explanation.

**sort** doesn't have the hangups about field delimiters that commands like **cut** have. Fields, as far as **sort** is concerned, are separated by the blank space between them, even if these blank spaces contain multiple non-printing characters. This is generally a good idea, but occasions arise when this causes problems, as in **/etc/passwd**, which has no blank

spaces. In these cases, the field separator can be specified with the **-t** option. **sort -t: -k5 /etc/passwd** would sort the file on the 5th (users real name), using the colon to decide where fields start and end.

Specifying the sort field used to be a strange affair, but with the **-k** option it is now reasonably straight forward.

**-k {number}** Specifies the field in position **{number}**.

Numbering starts at 1. More complex sort field arguments may be specified, such as which character within the sort field to start or end sorting on. I like to take a 'learn it if you need it' approach to these things, as I find that I rarely need such features and I don't like to clutter my poor brain unnecessarily.

**ls -l | sort -k9** Sorts a directory listing in dictionary order on the 9th field (file name).

```
jane@daisy > ~ $ ls -l | sort -k9
total 44
-rw-rw---- 1 jane jane 44 Nov 10 12:37 contacts2
-rw-r--r-- 2 jane jane 39 Nov 10 12:33 contacts-link
drwxr-xr-x 3 jane jane 4096 Nov 16 08:34 Desktop/
drwxr-xr-x 2 jane jane 4096 Feb 6 04:23 Documents/
drwxr-xr-x 2 jane jane 4096 Feb 26 2007 Movies/
drwxr-xr-x 2 jane jane 4096 Feb 25 2007 Music/
drwxr-xr-x 3 jane jane 4096 Nov 10 20:45 mydir/
drwxr-xr-x 2 jane jane 4096 Nov 10 12:06 mydir1/
-rw-r--r-- 1 jane jane 69 Nov 10 12:10 newfile
-rw-r--r-- 1 jane jane 2174 Nov 10 12:12 newfile2
drwxr-xr-x 2 jane jane 4096 Feb 8 07:54 Pictures/
lrwxrwxrwx 1 jane jane 5 Nov 12 12:39 tmp -> //tmp/
```

**ls -l | sort -nrk5** Sorts the listing by the 5th field (file size) in reverse numerical order.



```
jane@daisy > ~ $ ls -l | sort -nrk5
drwxr-xr-x 3 jane jane 4096 Nov 16 08:34 Desktop/
drwxr-xr-x 3 jane jane 4096 Nov 10 20:45 mydir/
drwxr-xr-x 2 jane jane 4096 Nov 10 12:06 mydir1/
drwxr-xr-x 2 jane jane 4096 Feb 8 10:40 Pictures/
drwxr-xr-x 2 jane jane 4096 Feb 6 04:23 Documents/
drwxr-xr-x 2 jane jane 4096 Feb 26 2007 Movies/
drwxr-xr-x 2 jane jane 4096 Feb 25 2007 Music/
-rw-r--r-- 1 jane jane 2174 Nov 10 12:12 newfile2
-rw-r--r-- 1 jane jane 69 Nov 10 12:10 newfile
-rw-rw---- 1 jane jane 44 Nov 10 12:37 contacts2
-rw-r--r-- 2 jane jane 39 Nov 10 12:33 contacts-link
lrwxrwxrwx 1 jane jane 5 Nov 12 12:39 tmp -> //tmp/
total 44
```

An awful lot can be done with these few commands and a little practice. If you want to do more, then of course you can. This is where the **sed** stream editor excels, enabling search and replace, insertion, deletion, substitution and translation to multiple files at the same time if that is what you want. **sed** can be a simple substitution tool or as complex as you like. We'll be meeting **sed** very soon.




**The NEW  
PCLinuxOS  
Magazine**

*Created with  
Scribus 1.3.5*

### Reach Us On The Web

**PCLinuxOS Magazine Mailing List:**  
<http://groups.google.com/group/pclinuxos-magazine>

**PCLinuxOS Magazine Web Site:**  
<http://pclosmag.com/>

**PCLinuxOS Magazine Forums:**

**PCLinuxOS Magazine Forum:**  
<http://pclosmag.com/forum/index.php>

**Main PCLinuxOS Forum:**  
<http://www.pclinuxos.com/forum/index.php?board=34.0>

**MyPCLinuxOS Forum:**  
<http://mypclinuxos.com/forum/index.php?board=157.0>

**ONE CLICK LINUX**  
A place for Linux beginners!

George © and © 2008 Mark Szorady



**PCLinuxOS**  
Magazine Forum

**Come  
Join Us!**

**PCLinuxOS**



**Phoenix Edition**



**PCLinuxOS 17**

**LinPC.US**

Hardware for your Linux PC.





# Command Line Interface Intro: Part 7

by Peter Kelly (critter)

## Shell Scripting

A script is simply a list of instructions that we want the system to execute, and in its simplest form it will do just that, line after line, obediently and blindly, with no concern of the consequences. Writing scripts is not difficult, but care must be taken to ensure that the instructions passed in the script perform what is **intended**, which unfortunately is not always what is actually written.

There are two common types of computer programs: compiled programs and interpreted programs. Compiled programs have their code converted to a machine language that the processor can understand, but is unintelligible to humans. This enables them to execute extremely quickly, but makes them more difficult to alter. Interpreted programs are mostly plain text files that are read line by line by the interpreter, which then instructs the processor. Shell scripts are interpreted programs, and in a bash script, the bash shell is the interpreter.

When we are at the command line, we can type in commands and have the shell perform some function for us. Sometimes, we type in the same commands regularly over a period of time, and at times, the commands get quite long and complex. Other times, we have to type in a whole series of commands to get our end result. If this sounds like you, then it is time to find out about scripting, and let the machine do the tedious work. The bash shell scripting language is a vast topic, and you will see many large volumes devoted to the subject in book stores. Fortunately, you need only a small part of all

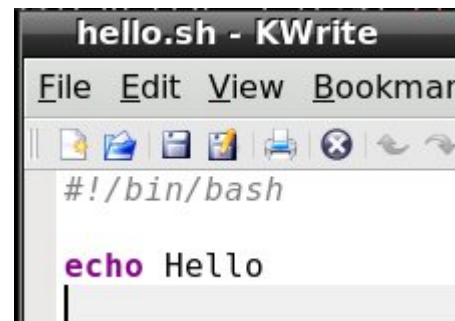
that wizardry to become a proficient script writer, and to be able to understand some of the scripts which are used to manage and control your system.

To write a script, you have to be able to type text into a file, and to then make that file executable. That can be as simple as entering text on the command line as follows:

```
cat > myscript
echo Hello
chmod +x myscript
```

Typing `./myscript` would then execute the script and print the word Hello to the screen. (The `./` is needed to tell the shell where the script is, as it is not in any of the usual places where executable files are to be found).

The method above works, but if we really want to write scripts, we should use a text editor, not a word processor, as these include strange formatting sequences that would confuse the shell. Any text



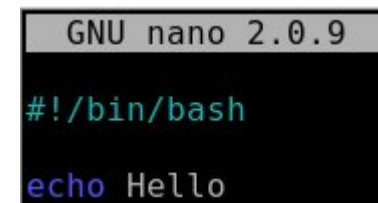
editor will do. My personal favorite is kwrite, but there are many others. Ideally, you want an editor that supports **syntax highlighting**.

Any programming language comprises many things such as comments, keywords, variables, text etc.

Syntax highlighting displays these things in different colors to make it easier to find a certain item.

If you want to be able to do this using a terminal editor, then nano supports syntax highlighting, but by default it is turned off. To enable it, you need to copy a file to your home directory. In a terminal, type `cp /usr/share/nano/sh.nanorc ~/.nanorc`.

Now every file that you edit in nano that ends in `.sh` will have syntax highlighting suitable for the bash scripting language. Other files will not be affected. The `.sh` extension is not required for scripts, but is a way of telling nano that "this is a bash script, so highlight it accordingly," and it does help to distinguish your scripts from other files. It is also a good idea to create a scripts directory in your home folder and store all your scripts in there. If you write a script that you find really useful, you can always transfer it to the `/bin` directory so that that is always available, as that directory is almost certainly in your `PATH` environment variable. Before you do that, please make sure that it is fully tested and won't one day spring a nasty surprise on you or any other user of the system.



When an executable file is passed to the shell, it passes it to the kernel, which then starts a new process and attempts to execute it. If this is not a compiled, machine language file (usually referred to as a binary), then this will fail and return an error to the shell, which then searches the file for commands that it knows how to process. It may get lucky, as in

the simple **myscript** example above, but as there are many scripting languages, this is not guaranteed and so we should tell the shell which interpreter to use. If the very first characters on the first line of a script are `#!` (known as shebang), then bash will take the rest of the line to be the fully qualified path of the interpreter. For a bash script we should always start with `#!/bin/bash`, or `#!/bin/sh` /bin/sh is usually a symbolic link to /bin/bash. For a perl script for example, we might use `#!/usr/bin/perl`.

What needs to be in a script? Well just the one line `#!/bin/bash` is technically a script, although it wouldn't do very much. In fact, it would do absolutely nothing more than start and end a process. To get results, we need to give it some commands, which it will execute one after another, **unless we tell it to do things differently**. This is the clever bit. **We** are in control and now have so much more power than when we simply typed in commands at the terminal. With a script, we can execute commands in the order that we want to, when we want to, dependent upon certain conditions that we define or that arise from system activity, and we can do this repeatedly or until a certain condition is met. We can pass options and arguments to the script at start up, or read information from a file or from the user at the terminal.

We could write a simple script to search a directory for all bash scripts like this:

```
#!/bin/bash

grep -rs "#!/bin/bash" /usr/bin/*
```

(Don't forget to make it executable with `chmod +x`).

The quotes are needed so that the script doesn't treat everything after the `#` as a comment. The second argument `/usr/bin/*` has no quotes, as we do want the shell to expand the `*` into a list of files.

We could do this at the command line without a script, or even define an alias that we might call `find-scripts`: **alias find-scripts='grep -rs "#!/bin/bash" /usr/bin/\*'**

Both of these would work, but would also find the pattern anywhere in a file or text embedded in a binary file, not only at the beginning, which denotes a bash script, but they suffice as examples.

## About variables

To make the effort worthwhile, we can enhance our script by passing it the name and path of the scripting language on the command line, making its use similar to a regular Linux command.

### find-scripts {search pattern}

To do this we need to use **variables**. We've met variables before. They are the names of bits of information that we or the shell need to keep track of, such as **PWD**, which holds the absolute path name of our current working directory, and **PATH**, which is a colon separated list of directories to search for executable files. These are **Environment Variables** used by the shell, but generally available to the user. You can also create your own variables. They are known as variables because if, for example you did a `cd` to another directory then the contents of **PWD** would change: Their contents are variable.

Many programming languages require that variables are declared before they are used, and that the type of content that they will be assigned is defined in that declaration. The type of content may be a string (of characters), an integer, floating point number or any one of many other types. Variable declaration is available in bash using the keyword **declare**, but for most purposes, it is not necessary, and the variables you create can be used to store strings or integers, as you require. Bash doesn't handle floating point arithmetic and needs to use utility commands, such as `bc`, when that functionality is required.

Bash also supports one dimensional **arrays** – one dimensional means that you can't (easily) have arrays of arrays!. An array is really just a group of variable elements with the same name and an index starting at zero for the first element. Arrays in bash are extremely flexible. For example, if we create an array named **pets** with the command **pets=(cat dog horse)**, then **pet[0]** refers to the value **cat** and **pet[2]** the value **horse**. If we now add **pets[4]=parrot** then that element gets added to the array even though **pets[3]** hasn't been assigned.

To access the contents of a particular element of an array we need to use brackets to enclose the index and braces to prevent expansion by the shell:

```
jane@daisy > ~ $ pets=(cat dog horse)
jane@daisy > ~ $ pets[4]=parrot
jane@daisy > ~ $ echo ${pets[1]}
dog
jane@daisy > ~ $ echo $pets[1]
cat[1]
```

`echo ${pets[1]}` correctly prints out **dog**, but `echo $pets[1]` prints **cat[1]** as the shell expands the variable **pets**, with no index number, to the first (zero) element of the array, and then echo treats the string **[1]** literally and adds it to the text on screen.

Quite often arrays in scripts are iterated through in a loop and their values passed directly to the rest of the code for processing which is a great way of getting things like lists of names or music into a script..

There are, of course, some special ways of accessing certain information about an array.

```
jane@daisy > ~ $ pets=(cat dog horse)
jane@daisy > ~ $ pets[4]=parrot
jane@daisy > ~ $ echo ${pets[*]}
cat dog horse parrot
jane@daisy > ~ $ echo ${#pets[*]}
4
jane@daisy > ~ $ echo ${!pets[*]}
0 1 2 4
jane@daisy > ~ $ echo ${#pets[2]}
5
```

`echo ${pets[*]}`

Outputs all the elements of the array

`echo ${#pets[*]}`

Outputs the number of elements in the array

`echo ${#pets[2]}`

Outputs the length of element **[2]** in the array

`echo ${!pets[*]}`

Outputs all the indexes of elements present in the

array. Notice that the unassigned index 3 is not present.

## Special bash variables

The shell has many variables at its disposal and uses some special ones to access arguments passed to a script. The first argument is known as **\$1**, the second as **\$2**, and so on. In the unlikely event that you need to pass more than 9 arguments to a script, then the number must be enclosed in braces, as **\${14}** for the 14th argument. **\$0** contains the name of the command as used on the command line.

Modifying the script like this

```
#!/bin/bash
grep -rs "#!$1" /usr/bin/*
```

allows us to call the script and pass it the absolute address of the interpreter.

`./find-scripts /bin/bash` to locate our bash scripts or `./find-scripts /usr/bin/perl` to find any perl scripts. We use the fact that the shell stores the first argument passed to it in the variable **\$1**.

Notice that here I have changed the single quotes to double quotes, which allow variable expansion (**\$1** is replaced by the first argument) to take place, but still treats the **#!** literally. This is where syntax highlighting is invaluable. In the first example, the **"#!/bin/bash"** in the command expression is displayed all in red text, which is the color used for strings. In the second example, **#!** is in red text, while **\$1** is in green text, the color used to highlight

variables. If I had used single quotes here, then the **\$1** would not have been expanded, leaving grep searching for files containing the pattern of characters **#!\$1**.

We can further refine the script by passing it the search directory as a second argument, which will be stored in **\$2**. We now call the script like this `./find-scripts /bin/bash /usr/bin`, passing two arguments to the script and making it much more flexible.

```
#!/bin/bash
grep -rs "#!$1" $2
```

During execution, **\$1** will be expanded to **/bin/bash**, and **\$2** expands to **/usr/bin**.

These enhancements unfortunately add a complication to the script, as we are now **required** to pass these arguments to the script. If we fail to pass the correct number of arguments to the script, then the variables **\$2** and/or **\$1** will be **undefined**. That means their value is not controlled by us and will contain a null value. As we are only reading files from a directory here, then we shouldn't cause any damage. But if the script was writing or deleting stuff, then the consequences can be imagined. You can get into a whole lot of trouble with an empty gun! This simple example should be enough to convince you!

**DON'T TRY THIS!**

`rm -rf /$RETURNED_PATH`

`rm` remove

-r recursing through the sub-directories

-f without asking or prompting

**/\$RETURNED\_PATH:** if this variable is undefined, then it expands to / and the command completes as “remove everything below the root directory recursively, without prompting” and deletes everything in and below the root directory – your entire system is gone, permanently and without so much as a “thank you”.

## Conditional Programming

Another special variable is  **\$#** , which contains the number of arguments passed to the script. We can use this to check that all is OK before proceeding.

```

1  #!/bin/bash
2
3  if [ $# != 2 ]
4  then
5      echo Usage: Find-scripts pattern directory >2
6      exit 1
7  fi
8
9  grep -rs "#!$1" $2

```

I've put line numbers in to help clarify things. They have nothing to do with the script are not typed in.

Lines 3 to 7 contain a construct known as an **if-then statement**. After the initial **if** keyword on line 3, we test a **condition** for truth. Here the test **[ \$# != 2 ]** checks if the total number of arguments passed is not equal to 2. The spaces inside the brackets are very important.

If it is true (that there are not 2 arguments) we execute lines 4,5 and 6. Line 4 is the entry point to

commands that are only executed when the test condition is true. Line 5 echoes a usage message to the terminal. Line 6 exits the script, as we don't have sufficient information to continue, and returns a value of 1. Line 7 ends the conditional statement and allows the script to continue.

In this instance we don't use the return value of 1, which by convention signifies a failure – 0 means success. Other numbers up to and including 125 are available for the programmers use. If this script was called from another, then that parent script would know the outcome from this value and could then act accordingly.

If you use the script in a couple of months time, or even a few years down the line, you might not remember what pattern and directory the script is complaining about. It is even less likely that another user would know. One thing we can and should do is to add comments to the script, detailing what is going on. A comment is any text on its own line, or at the end of a line, that starts with a # (with exception of the initial #! which has special meaning to the shell). This text is ignored by the script.

```

#!/bin/bash
# search a directory to find scripts
# Needs a pattern to find e.g. /bin/bash
# and a directory path to search
#
# Jane Doe February 2010

if [ $# != 2 ] # We need 2 arguments
then
    # Print out a message
    echo Usage: Find-scripts pattern directory >2
    exit 1 # Quit the script
fi
# The correct number of arguments
# have been supplied so continue
grep -rs "#!$1" $2 # find script files

```

There are more comments in this file than you may usually find, but an informative header can save a lot of head scratching. Indentation can also help to make a script more readable.

The test used in the example above,  **\$# != 2** , is derived from the negation symbol  **!** . And with the equality symbol  **=**  together, they give a 'not equal test.' But what if we want to test if a file was a directory or if the file even exists? Well, the shell has its very own test command with the following basic syntax: **test {expression1} {condition} {expression2}**.

Using this command the test in the if statement would have been written like this: **if test \$# -ne 2**. As a matter of the fact, the two forms are completely interchangeable, and the conditions available for the test command can be used equally well by the original format **[ \$# -ne 2 ]**. The shell has many functions like test built in to it. They are known, unsurprisingly, as **shell builtins**. The keyword **test** is a builtin, as is **[**, which has the same meaning.

The use of tests is so central to shell scripting to determine the flow of the program that you should be aware of the tests available. I give here a complete list of the tests available as described in the official man page documentation.

### Where EXP is an expression

( EXP )	EXP is true
! EXP	EXP is false
EXP1 -a EXP2	both EXP1 and EXP2 are true (logical and)
EXP1 -o EXP2	either EXP1 or EXP2 is true (logical or)



### where STR is a string

-n STR            the length of STR is nonzero  
 STR                equivalent to -n STR  
 -z STR            the length of STR is zero  
 STR1 = STR2       the strings are equal  
 STR1 != STR2     the strings are not equal

### Where INT is an integer

INT1 -eq INT2     INT1 is equal to INT2  
 INT1 -ge INT2     INT1 is greater than or  
                       equal to INT2  
 INT1 -gt INT2     INT1 is greater than INT2  
 INT1 -le INT2     INT1 is less than or  
                       equal to INT2  
 INT1 -lt INT2     INT1 is less than INT2  
 INT1 -ne INT2     INT1 is not equal to INT2

### Where F is a file

F1 -ef F2        F1 and F2 have the same device  
                       and inode numbers  
 F1 -nt F2        F1 is newer (modification date)  
                       than F2  
 F1 -ot F2        F1 is older than F2  
 -b F             F exists and is block special  
 -c F             F exists and is character special  
 -d F             F exists and is a directory  
 -e F             F exists  
 -f F             F exists and is a regular file  
 -g F             F exists and is set-group-ID  
 -G F             F exists and is owned by the  
                       effective group ID  
 -h F             F exists and is a symbolic link  
                       (same as -L )  
 -k F             F exists and has its sticky bit set  
 -L F             F exists and is a symbolic link  
                       (same as -h )  
 -O F             F exists and is owned by the  
                       effective user ID  
 -p F             F exists and is a named pipe  
 -r F             F exists and read permission is  
                       granted

-s F             F exists and has a size greater  
                       than zero  
 -S F             F exists and is a socket  
 -t FD            file descriptor FD is opened on a  
                       terminal  
 -u F             F exists and its set-user-ID bit is  
                       set  
 -w F             F exists and write permission is  
                       granted  
 -x F             F exists and execute (or search)  
                       permission is granted

That list should give you some idea of the flexibility you have when performing a test.

**Note!** The **-e** test for the existence of a file can also be written **-a**, but I choose to ignore this, as it is too easy to confuse with the **-a** (logical and) test. You may, however, see it used in other scripts.

The **if-then** statement may also contain the **else** keyword, which works like this:

#### if {condition}

```
then
    commands to execute if the condition is met
else
    commands to execute if the condition is not met
fi
```

In the next example, I use the command **read**, which is an easy way to get user input into a variable as a script is running.

```
#!/bin/bash

echo Did you enjoy the show? y/n
read ANSWER
if [ $ANSWER = y ]
.
    then echo "Yes"
.
else
    echo "No"
.
fi
```

After the first echo command, the script pauses until the user enters something at the keyboard and presses the return key. The users input is stored in the variable **ANSWER**. This time the script does something different, depending on the users input.

But what if the user types in something other than Y or N? To cope with this, we introduce another keyword – **elif**.

```
#!/bin/bash

echo Did you enjoy the show? y/n
read ANSWER
if [ $ANSWER = y ]
.
    then echo "Yes"
.
elif [ $ANSWER = n ]
.
    then echo "No"
.
else
    echo "I'm sorry I don't understand"
.
fi
```

In this script, the acceptable responses are caught and acted upon. Any other response is dealt with by the code after **else**. This would appear to solve the problem, but if the return key is pressed without the user entering a response, then nothing is assigned to the variable **ANSWER**, which defaults to a null

value, and the script would see the tests as [ = y ] and [ = n ], which produces the error message **unary operator expected**. The way around this is to use double quotes around the variable, which causes the test to be seen as [ "" = y ] or [ "" = n ], which are valid expressions that the shell can work with. The "" in the test is an empty string (a string with no characters), which is not the same as a null.

```
#!/bin/bash

echo Did you enjoy the show? y/n
read ANSWER
if [ "$ANSWER" = y ]
.   then echo "Yes"
elif [ "$ANSWER" = n ]
.   then echo "No"
else
.   echo "I'm sorry I don't understand"
fi
```

You can have as many **elif** tests as you wish, and the if statement can be nested as many times as you can keep track of.

```
If [condition]
then
    if [condition]
    then
        if [condition]
        ...
        ...
        ...
        fi
    fi
fi
```

And of course each **if** statement can have its own **elifs** and **elses**. Here's a longer one with those line numbers again.

```
1 #!/bin/bash
2
3 # Get the current month & day
4 MONTH=`date +%m`
5 DAY=`date +%d`
6 if [ "$MONTH" -le 3 ]
7 then #jan to mar
8     echo "It's a long time until Christmas"
9 elif [ "$MONTH" -gt 3 -a "$MONTH" -le 6 ]
10 then #apr to jun
11     echo "It's a while until Christmas".
12 elif [ "$MONTH" -gt 6 -a "$MONTH" -le 9 ]
13 then #jul to sep
14     echo "soon we'll be thinking about Christmas"
15 elif [ "$MONTH" -gt 9 -a "$MONTH" -lt 12 ]
16 then #oct to nov
17     echo "Not long now until Christmas"
18 elif [ "$MONTH" -eq 12 ]
19 then #it's dec - check the day
20     if [ "$DAY" -ge 1 -a "$DAY" -le 18 ]
21     then # up to the 18th
22         echo "Just a few days to Christmas".
23     elif [ "$DAY" -gt 18 -a "$DAY" -le 24 ]
24     then # 20th to 24th
25         echo "Christmas is less than a week away".
26     elif [ "$DAY" -eq 25 ]
27     then # It's Christmas day
28         echo "Happy Christmas"
29     elif [ "$DAY" -ge 26 -a "$DAY" -le 31 ]
30     then # After Christmas
31         echo "So That was Christmas"
32     else #Something went wrong with the date
33         echo "Are you sure about that date?"
34         exit 1
35     fi
36 else #Something went wrong with the date
37     echo "Are you sure about that date?"
38     exit 1
39 fi
```

Line 1 is our standard bash script header. Line 3 is a comment and ignored.

Lines 4 & 5 use the date function with format modifiers (%m and %d) to get the current month and date into our variables

Line 6 Starts the first of 2 if-then constructs checking if the month is **less than or equal to 3**.

Line 9 tests if the month is **greater than 3 and less than or equal to 6**. That is, it is either 4, 5 or 6.

Line 19 We've discovered that it is December so we start the second if-then construct to check the day.

Lines 23, 26 and 29 do more day testing.

Line 32 the default else statement. If we got here, the the day was not in the range 1 – 31, so something is wrong and we leave the script.

Line 36 We find ourselves back in the first if-then construct at the else statement. If we got here, the the month was not in the range 1 -12, so something is wrong and we leave the script.

Line 39 Terminates the first if-then construct.

While the above script is useful to demonstrate the use of nesting if – then statements and the use of multiple **elifs**, it is not the only way or the most efficient way to program this.

We could have used the **case** statement, which is another conditional construct. This is the syntax for the **case** statement.

```
Case {pattern} in
    value1)
        commands
        ;;
    value2)
        commands
        ;;
    ...
```

```
...
...
*)
    commands
;;
esac
```

In this structure, the pattern is something, like the contents of a variable, that you want to use to control the actions of the script. If it has value1, then those commands up to the ;; are executed. Value2 causes a different set of commands to be executed and so forth, until all values that you wish to test for have been checked. The default at the end of the statement catches any other value, and is used as a fail-safe for unwanted or unexpected values. It can also provide a way to exit the script (or code segment). To test for multiple values, separate them with a pipe symbol |.

In the next example, I have mixed a case statement and the nested if-then statement from the previous example, and added line numbers to the figure.

Because the values to be tested in line numbers 11 to 15 are integer numbers and the date function returns a two character string such as "02," the tests in lines 8 to 12 would fail because of the leading "0". To overcome this, we echo the value through a pipe to the tr (translate) command and use the option -d (delete) with the argument "0," which deletes any zeroes in the string. unless the string is "10," which is an integer. This expression is evaluated in the back ticks and assigned to the new variable RAWMONTH.

```
1  #!/bin/bash
2  MONTH=`date +%m`
3  if [ $MONTH != 10 ]
4  then
5  RAWMONTH=`echo $MONTH | tr -d 0`
6  else
7  RAWMONTH=10
8  fi
9  DAY=`date +%d`
10 case $RAWMONTH in
11 1|2|3) echo "It's a long time until Christmas" ;;
12 4|5|6) echo "It's a while until Christmas" ;;
13 7|8|9) echo "soon we'll be thinking about Christmas" ;;
14 10|11) echo "Not long now until Christmas" ;;
15 12).   if [ "$DAY" -ge 1 -a "$DAY" -le 18 ]
16         then # up to the 18th
17             echo "Just a few days to Christmas".
18         elif [ "$DAY" -gt 18 -a "$DAY" -le 24 ]
19             then # 20th to 24th
20             echo "Christmas is less than a week away".
21         elif [ "$DAY" -eq 25 ]
22             then # It's Christmas day
23             echo "Happy Christmas"
24         elif [ "$DAY" -ge 26 -a "$DAY" -le 31 ]
25             then # After Christmas
26             echo "So That was Christmas"
27         else
28             echo "Are you sure about that date?"
29             exit 1
30         fi ;;
31 *) #Something went wrong with the date
32     echo "Are you sure about that date?"
33     exit 1
34     ;;
35 esac
```

We could have used the two character string as returned from the data function in the case statements, but using integers demonstrates the need to be aware of the **type** of data we use in tests.

Each test in the case statement is on one line here to make it more compact. If there are multiple commands for a test, then they should be separated by a semicolon or by a newline character (which means on separate lines). I think that you'll agree that the **case** statement is easier to read than the many **elifs** in the **if** statement.

The if-then and case structures are examples of conditional programming where the progress and direction of the script is determined by the results of certain tests. The shell has two conditional operators, **&&** and **||**, known as "**logical and**" and "**logical or**". They work both in unary (one argument) and binary (two arguments) mode.

In unary mode:

```
[ "$A" -gt 4 ] && echo "yes"
```

If the expression [ "\$A" -gt 4 ] evaluates to true the the echo command is executed, if false the script ignores the interruption and continues.

The || operator has the opposite effect in that the expression has to evaluate to false for the command to be executed.

Binary mode is used to test two arguments:

```
if [ "$A" -lt 4 ] && [ "$B" -gt 9 ] echo "yes"
```

The echo command is executed if and only if both expressions are true.

if [ "\$A" -lt 4 ] || [ "\$B" -gt 9 ] echo "yes" The echo command is executed if either or both expressions are true.

This is similar, but not the same, as the test operators -a and -o. When using the test operators, both expressions are evaluated, and then the test is performed. The && shell operator evaluates the first expression, and if it is false, then there is no point in

looking at the second expression, as the **'and'** condition cannot be met.

In a similar manner, if the first expression in an 'or' test using the || operator evaluates to true, then the condition has already been met and the second expression doesn't need to be evaluated. For this reason, they are known as **short circuit operators**.

## The scope of variables

As we have now started to use our own variables, it is important that you understand the **scope** of variables before we move on. The scope of a variable is where its assigned value is valid. Variables may be **local** or **global**. For example, while on the command line, you are in a running shell and you may create variables

```
jane@daisy > ~ $ MYVAR=a_value
jane@daisy > ~ $ echo $MYVAR
a_value
```

The scope of that variable is the currently running shell process. When you exit the shell, the variable ceases to exist, and if you start a new shell the variable isn't available, as it is **local** to the shell process where it was created. When you run a script a new shell process is started and any variables that you create are **local** to that script and not available elsewhere.

Environment variables are **global** variables and are available to all processes. In order to make your variables available to other processes, they need to be **exported** to the **environment**. All new processes inherit the environment of their parent process.

When an exported variable is passed to a child process, it retains the value assigned it in the parent process. The child process may change the value of the variable, but the value seen by the parent remains unchanged.

```
jane@daisy > ~ $ export AGE=22
jane@daisy > ~ $ echo $AGE
22
jane@daisy > ~ $ su john
Password:
john@daisy > jane $ echo $AGE
22
john@daisy > jane $ AGE=19
john@daisy > jane $ echo $AGE
19
john@daisy > jane $ exit
exit
jane@daisy > ~ $ echo $AGE
22
jane@daisy > ~ $
```

Jane set the variable **AGE** to 22, her age, and exported it. When the **su** command was executed to switch to user john, a new shell process was started which could access the variable and its value, as set by jane, which john subsequently changed to 19, his age. Jane still sees the variable set as 22.

To remove a variable, use the command **unset**.

```
jane@daisy > ~ $ unset AGE
jane@daisy > ~ $ echo $AGE

jane@daisy > ~ $
```

Another command used with variables is **readonly**, which has the effect of turning a variable into a **constant** – a variable whose value, once set, cannot vary. For example, **readonly KB=1024**. The assigned value cannot be changed during the life of the process and readonly variables cannot be **unset**.

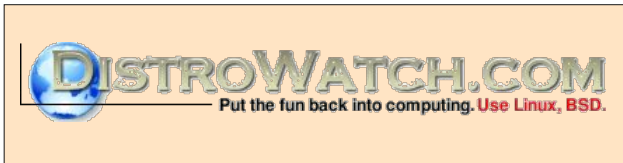
The **env** command is used to display the environment variables that are currently set and to control the environment that gets passed to commands. The **env** command on its own will display a list of all current environment variables. If the **env** command is followed by one or more variable assignments and used before a command, the environment passed to the command will be modified, but the current working environment will be unaffected. With the **-i** option the current environment will be ignored and only variables passed to the command will be used.

```
jane@daisy > ~ $ env -i HOME=/tmp env
HOME=/tmp
jane@daisy > ~ $ env | grep ^HOME
HOME=/home/jane
jane@daisy > ~ $
```

The environment variable **HOME**, which normally contains the full path name of the users home directory, is temporarily changed to /tmp, and all other environment variables discarded. This new environment is then passed to the command **env**, which starts in a new process and lists out all its known environment variables. There is only one, as the others were discarded.



The `env` command is then immediately executed in the current shell, and the output searched for lines that begin with the pattern "HOME". The changed environment existed only for the process to which it was passed.



# Screenshot Showcase



*Posted by exploder, March 21, 2010, running PCLinuxOS e17.*

# Command Line Interface Intro: Part 8

by Peter Kelly (critter)

## Shell Scripting Part 2

One of the reasons for writing a script is because we need to perform the same operation on many objects. To do this, we take the first object and subject it to a sequence of commands which examine, transform, copy, delete or otherwise act upon the object. When we have finished with that object, then we loop back to where we started, take another object and repeat the exercise. We do this until all the objects have been dealt with. We use a flow control statement known as a loop to achieve this.

## Loops

By far the most commonly used loop construct in scripts is the **For** loop, which I used to rename a batch of files when discussing shell expansion. Now I can explain how it works.

The syntax of a for loop is:

```
for {variable} in {set}
do
command 1
:
:
command N
done
```

variable can be any unused variable name and a single character. i or x is often used.

set is the set of values that you want to assign to variable at each iteration.

An example will explain this better.

```
jane@daisy > ~ $ for i in apples oranges grapes
> do
> echo $i are fruit
> done
apples are fruit
oranges are fruit
grapes are fruit
```

You can type in these constructs at the command line and the shell will keep prompting for more input until the closing keyword of the construct is entered, the word “done” in this example.

Each of the values in the set of values after the word in are successively assigned to the variable i and then all of the commands between do and done are executed. There is only one command here, and the \$i is expanded to its current value when the loop runs. While this example is of no practical use, other than to demonstrate the use of the for loop, a more common practical use is to loop through a list of file names, performing tests or actions on each file.

The following code loops through the contents of a directory and checks at each iteration if the current object is a regular file. If so, it echoes the file name to screen.

```
#!/bin/bash
for i in ~/*
do
    if [ -f "$i" ].
    then
        echo `basename $i`
    fi
done
```

The if – then loop is needed to exclude objects, such as directories. The basename command is used to strip away any leading directory names that make up the path to the file. Used in this way, it is possible to run through large lists of objects and to then select only the ones that you want to work with.

We can put almost anything in the list of objects to loop through. When using numbers we can use a range.

{5..10} will give the integer set (5 6 7 8 9 10), and we can include a step value.

{5..20..3} gives the set of integers (5 8 11 14 17 20)

You may occasionally come across an older script that uses the external command **seq** to handle sequences of numbers like this:

**for i in \$(seq start end step)**

rather than the bash notation **{start..end..step}**. Both work, but the bash way is faster.

The bash variable \$@ contains a list of the arguments passed to the script on the command line, and this can be put to good use to loop through those arguments. However this is not necessary, as simply omitting the list completely has the same effect.

```
#!/bin/bash
for i
do
    echo "$i"
done
```

```
jane@daisy > ~ $ ./for.sh first second last
first
second
last
```

The next two looping constructs, while & until, are very much alike.

```
while [test condition]
do
commands
done
```

and

```
until [test condition]
do
commands
done
```

The difference is that while loops as long as the test is successful and until carries on until the test is unsuccessful.

```
#!/bin/bash
while [ "$s" != "exit" ]
do
    echo "Type exit to quit, anything else to continue"
    read s
done
```

This keeps looping as long as the statement "string s does not have the value 'exit'" is true whereas,

```
#!/bin/bash
until [ "$s" = "exit" ]
do
    echo "Type exit to quit, anything else to continue"
    read s
done
```

keeps looping as long as the statement "string s has the value 'exit'" is false.

You will find that while loops are used more than until loops, and are often used to repeat an operation a fixed number of times.

```
#!/bin/bash
x=1
while [ $x -le 5 ]
do
    echo $x
    x=$(( $x+1 ))
done
```

All of the bash looping constructs can be nested and may contain other constructs.

Occasionally, you may find that during the execution of a loop a condition arises that requires the loop be exited, and execution of the rest of the script be resumed. For those occasions, bash provides the break command. In this example, we use another method of indexing the loop, using a three-parameter loop control expression.

```
#!/bin/bash
for (( i=1; i<=5; i++ ))
do
    if [ "$i" -eq 4 ]
    then
        break
    fi
    echo $i
done

echo "the loop has terminated "
echo "but the script continues"
```

The first expression, **i=1**, initializes the count, **i<=5** sets the maximal count and **i++** increases the count by one for each iteration. The second expression can be any valid test, and the third expression could be **i--** for a decreasing count or something like **i+=3** to increase the count by three for each iteration. In this script only, the values 1, 2 and 3 are printed to the screen.

If you want only to stop the current iteration of the loop before the end of the loop body, and then to continue the next iteration of the loop, then the command **continue** will do just that. This example loops through the contents of a directory, discarding all sub-directories.

```
#!/bin/bash
for i in *
do
    if [ -d "$i" ]
    then
        continue
    fi
    echo $i
done
```

Both of these commands take an optional numeric argument that allows you to specify the number of levels of enclosing loops to get out of, e.g. **break 2**, to back out of two nested loops.

In the previous examples, all output to the screen has been done using the **echo** command, which is easy to use but rather limited. A more useful tool is the bash builtin **printf**, which provides us with the means to format the text.

**printf {format-string} {arguments}**

The format-string part of the syntax is a mixture of ordinary text to be printed literally, escape sequences (such as `\n` to print a newline character) and format specifications like `%s` to denote a character string, or `%d` for a decimal integer. The arguments are what you actually want to print.

The most useful escape sequences are:

- `\b` backspace
- `\f` formfeed
- `\n` newline
- `\t` tab
- `\v` vertical tab

The format specifiers cater for character strings, signed and unsigned decimal integers and floating point numbers, with or without the exponent, as well as octal and hexadecimal numbers. If you don't know what some of these are, don't worry. Chances are that you won't need them.

You can, of course, just supply text to the command without any of the fancy escape sequences, or format specifications. But if, at a command prompt, you type:

**printf "Hello World"**

You will find that your command prompt is placed at the end of the text. Unlike the `echo` command, the `printf` command does not automatically supply a newline character, and so the text insertion point remains immediately after the printed text.

**printf "Hello World\n"** behaves as is normally expected.

While this may at first seem a burden, it actually enhances the usability of the function, allowing more precise control over the output.

```
1 #!/bin/bash
2
3 x=0
4 for i in *
5 do
6 x=$((x+1))
7 done
8
9 printf "You are currently in %s\nwhich holds %d files\n" $PWD $x
```

```
jane@daisy > ~ $ cd /bin
jane@daisy > bin $ ~/printf-demo.sh
You are currently in /bin
which holds 118 files
```

Line 3 initializes a variable, named `x`, to zero. This is not really necessary but it is good practice to precisely control variables.

The loop in lines 4 to 7 simply counts the number of entries in the directory.

Finally line 9 does the business starting with some literal text and then adding the first of the supplied arguments, the environment variable `$PWD`, which holds your current directory. The `%s` tells the command to treat the argument as a character string. Next is a newline character, followed by some literal text. The newline ensures that the following text is put on the next line down. Note that there is no space between the newline and the text. Had there been a space it would have been the first character at the beginning of the line, indenting the text. `%d` gets the next argument, `$x`, the file count, and treats it as an integer number when printing it.

The format string is ended with another newline and the whole of the format string is enclosed in double quotes.

Treating the variable `$x` as an integer had no effect in the previous example. I could have achieved the same result if I had used `$s` and output it as a string.

The format specifiers are able to accept optional modifying flags, which are inserted between the `%` and the format specifier `%` flags width.precision.

*width* is the total number of spaces that the inserted value will occupy. If the value is smaller than the specified width then it is padded out from the left (right justified)

*precision* is the number of digits or characters to output. This varies depending on the format specifier. For a string it is the maximum number of characters. For integers it is the minimum number of digits, default 1. For floating point numbers it is the number of decimal places.

Flags can be one or more of the following:

space prefix positive numbers with a space and negative ones with a minus sign

- left justify the inserted value
- + prefix numbers with a + or - sign
- 0 pad out numbers with zeroes instead of spaces
- # change the form of the output

If you need to use the last one, then you certainly don't need me to tell you how to use it.



A few examples to get you started:

```
#!/bin/bash
PI=3.14159265
S=supercalifragilisticexpialidocious

#pad the string to 15 character spaces wide
printf "The value of pi is %15s approximately\n" $PI
#left justified
printf "The value of pi is %-15s approximately\n" $PI
#15 spaces wide, 6 dec. place precision
printf "The value of pi is %15.6f approximately\n" $PI
# 2 flags - left justify & show sign
printf "The value of pi is %~+15.3f approximately\n" $PI
#truncate the string
printf "%.15s...\n" $S
```

```
jane@daisy > ~ $ ./printf-demo2.sh
The value of pi is      3.14159265 approximately
The value of pi is 3.14159265 approximately
The value of pi is      3.141593 approximately
The value of pi is +3.142 approximately
supercalifragil...
```

If there are more arguments than format specifiers, then the format string is reused, treating missing arguments as zero or an empty string. For example, if we modify the first script:

```
#!/bin/bash

x=0
for i in *
do
x=$((x+1))
done

printf "Directory is %s containing %d files\n" $PWD $x /bin
```

The first time around, all is fine, but there is still the unused “/bin” argument, so the format string is reused. However, it expects a string and an integer, so it inserts a zero for the missing argument.

```
jane@daisy > ~ $ ~/printf-demo.sh
Directory is /home/jane containing 20 files
Directory is /bin containing 0 files
```

If the second line of output was true we would have a major problem.

## Functions

You can think of a function as a sub-script. It is a block of code that is executed by calling its name, along with any arguments that you want the function to process, and the function must be defined before it is called. For this reason, it is usual to define functions at the beginning of the script, but they may also be called from a separate file. When the same code is used in several places in a script, then you should consider using a function definition.

As the shell moves through the script, it recognizes function definitions and stores the commands in memory for later use. This makes the use of functions in a script an extremely efficient way of coding. A function can be called from within a function.

This example script exits if the user is root, but a user who has used the su command to get temporary root privileges will not be detected. You

```
#!/bin/bash
checkroot ()
{
if [ $1 = "root" ]
then
echo "You are not permitted to run this script as root"
return 1 #false
else
return 0 #true
fi
}

if !( checkroot $USER )
then
exit
fi
```

need to also check the environment variable \$USERNAME to catch those users.

The arguments passed to the function use the same notation as arguments passed to the script on the command line, known as positional parameters. The command line arguments are temporarily stored in memory during the execution of the function. Here the first (and only) argument passed to the function is \$USER, and is referenced by the function as \$1. The return value can be examined to determine the outcome of the function. Zero is always considered to be true, and any positive integer is taken to be false. A function may be as simple or as complex as you like, but it may not be empty.

When processing the arguments passed to a script or a function, it is often useful to use the shift command. What this does is to shift all the arguments one or more places to the left, so that the contents of \$1 are replaced by the contents of \$2, \$3 goes into \$2 and so on. We can use this to hand down arguments, one at a time, to a loop, process it and then get the next argument. If the argument \$1 has its own qualifying argument, say a file name to be used with that argument, then this will be found in \$2, Then after processing, this argument pair can use an extra shift command or supply the shift command with an optional count parameter shift 2 to move the arguments the required places to the left.

```
#!/bin/bash

while [ $# -ne 0 ]
do
  if [ $1 = "filename" ]
  then
    echo "file is $2"
    shift 2
  fi
  echo $1
  shift
done
```

```
jane@daisy > ~ $ ./shift-demo.sh two four filename testfile six eight
two
four
file is testfile
six
eight
```

Useful as this command is for passing consecutive arguments in \$1 to a portion of code in a script or function for processing, the need to scan a set of options and arguments passed to a script has resulted in the **getopts** command. This command greatly simplifies the parsing of command lines. The **getopts** command accepts a list of options valid in the script or function, and recognizes that any options followed by a colon require an additional argument, which is placed in the variable **\$OPTARG**, each supplied option being stripped of a leading – before being placed in a variable supplied to the command **getopts {options} {var} {arguments}**.

If we were to write a script with the syntax

### myscript -cnh

```
-c [destination] copy a file to directory destination
-n print a count of files processed
-h print a help message and exit
```

which we might use to count or backup a set of files provided in the arguments section of the command line. To process the command, we could use code similar to the following:

```
COUNTING=0
```

```
while getopts c:nh options
do
  case $options in
    1. DEST=$OPTARG
  ;;
    n)COUNTING=1
  ;;
    h)echo "For usage please see the
    accompanying documentation."
    exit 0
  ;;
  esac
done
```

This sets up the script functionality so that testing the contents of the variable **\$COUNTING** tells us if we need to provide a count of the files, and if the -c option was specified, then the variable **\$DEST**, if it is defined, tells us to perform the copy operation on the files in the argument list and contains the path to where we wish to copy the files.

The command **getopt** does not remove the options from the command line, but maintains an index to the next option in the variable **\$OPTIND**. If we use the shift command after the while loop, as

```
shift $(( OPTIND - 1 ))
```

then all the options and their required options are removed, leaving only the arguments (file list to be processed) in the positional parameters \$1, \$2 ...

If you are using the **getopt** command to process arguments to both the script and to functions within that script, then you should be aware that the variable **OPTIND** is not automatically reset and should therefore be reset at the beginning of the function, to ensure that the first argument retrieved is, in fact, the first argument passed to the function.

## Zenity

We now have a nice set of tools to start building our scripts, and these few routines are sufficient to get started on the coding of some fairly sophisticated utilities. You just need to provide logic, intuition and patience. What we have in our toolbox so far is fine when we are writing scripts that only we shall be using, but if we want to provide a solution for more general use, then we need to make the scripts a little more user friendly. Some of the potential users may not be as command line savvy as you now are.

Fortunately, there are some excellent utilities in the repositories to help here, and more than likely, one or more will already be installed if you are using PCLinuxOS. The command **dialog** can be used to provide simple pop up boxes in the terminal:

```
dialog --msgbox 'Hello World!' 8 20
```

displays a simple message box 8 lines high by 20 characters wide, with a mouse click-able OK button and the message "Hello World!"

KDE provides `kdial` to provide a similar capability using dialog boxes directly on the KDE desktop and returning results to the running script.

There are others, and they all have their virtues and vices, but a very popular one that is extremely simple to use is called `Zenity`. PCLinuxOS users can see this in action if they run the excellent `Repo Speed Test` utility by `travisn000`. Reading the text of the script is highly recommended to better understand how a lot of the topics we have recently covered fit together to produce a useful utility, and you'll learn a few more tricks as well. The script can be found as `/usr/bin/apt-sources-update.sh`.

All of these dialog utilities are quite comprehensive, but easy to implement and a good overview of the capabilities can be had by typing the command name followed by `--help`. I shall demonstrate some of the ways that `zenity` can be used to spice up your scripts and provide a professional look.

The syntax of the `zenity` command is simple

## zenity options

The options determine the type of dialog to display, along with any options relative to that particular dialog. The types of dialog available and the option to call them include:

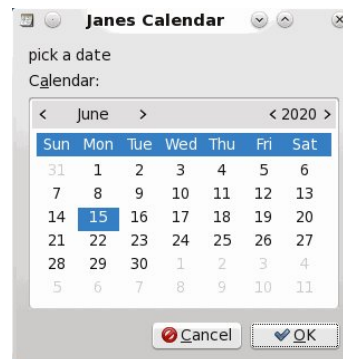
- `--calendar` calendar dialog
- `--entry` text entry dialog
- `--error` error dialog
- `--file-selection` file selection dialog
- `--info` info dialog
- `--list` list dialog

- `--notification` notification icon
- `--progress` progress indication dialog
- `--question` question dialog
- `--text-info` text information dialog
- `--warning` warning dialog
- `--scale` sliding scale dialog

## Calendar

The calendar dialog displays a nice monthly calendar in a window, from which you can select a date. You may specify some text and a title to be displayed on the dialog, as well as the day, month and year to be shown when the dialog is shown. The width and height of the dialog may also be specified. The command can get to be quite long, so I have used the line continuation character `\` to save space, but it is all treated as a single line by the shell.

```
zenity --calendar --title="Janes Calendar" \
--text="pick a date" \
--day=15 \
--month=6 \
--year=2020 \
--width=300
```



The selected date is returned by default in the format `06/15/2020`, but there is another option that allows you to completely control what you get.

## --date-option=STRING

where `STRING` conforms to the specification of the `strftime` function. There is far too much to cover here (Google is your friend) but briefly:

`"%A %d/%m/%Y"` produces `Monday 15/06/2020` and

`"%a %d %B %Y"` changes it to `Mon 15 June 2020`. Get the idea?

The returned date can be simply captured in a variable by enclosing the entire command in backticks:

```
MYDATE=`zenity --calendar`
```

Clicking the cancel button returns an empty string.

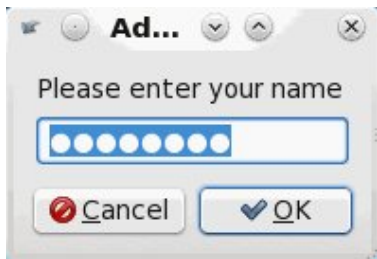
## Text Entry

The text entry dialog provides a simple way to read data into a script. The `--entry-text` option provides the default text when the dialog is shown.

```
zenity --entry --text="Please enter your name" --
entry-text="name"
```



A rather useful feature is the `--hide-text` option. This is useful for entering passwords.

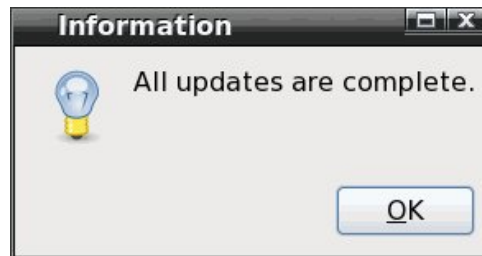
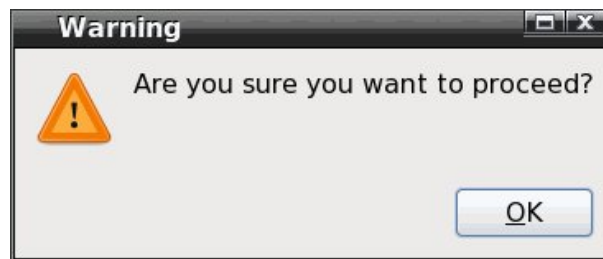


Beware though that this returns an unencrypted plain text string.

### Error, Warning, Question and Information

These four dialog boxes are very simple text boxes and are shown below with their default text and icons.

Of course the text, width and height can be changed with the corresponding `--text`, `--width` & `--height` options to suit the application.



### File Selection

The file selection dialog sets up file reading and writing through a nice interface. It doesn't actually do the read or write operation, as that must be done in the script, but it does make things easier for both the user and the script writer.

The dialog defaults to read mode and returns the selected file name and its full path. If the `--multiple` option is specified, then multiple files may be selected and are returned separated by a vertical bar character `|`. This separator character can be changed with the `--separator=SEPARATOR_CHARACTER` option. The `--directory` option restricts the selection to directories only. The `--save` option adds a text entry box which prompts for the file name, which may be preset with the `--filename=FILENAME` option. This allows you to select the name and directory to save the file through a graphical method, and this file name and path are returned by the command. If the `--confirm-overwrite` option is used then a warning dialog prompt will pop up if the file already exists.

**zenity --file-selection --save --confirm-overwrite** brings up this dialog:

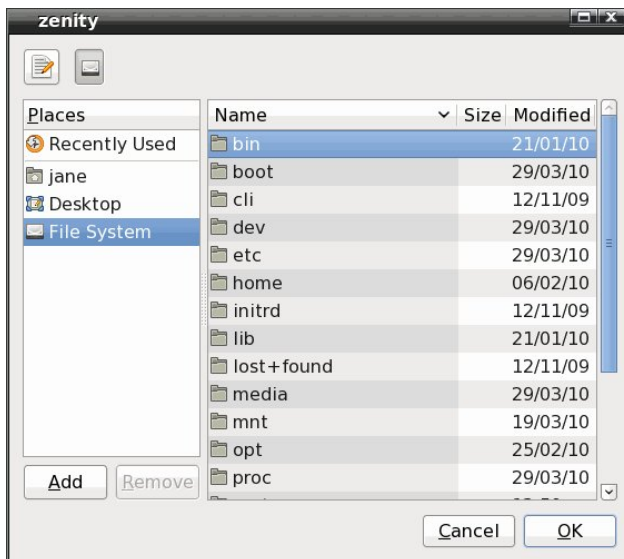




And typing in the name of an existing file warns the user with this.



Clicking on "Browse for other folders" or opening the dialog in the default read mode by not issuing the --save option brings up a fully search-able file dialog that most GUI users would be comfortable with.



## Notification

The --notification option puts a tiny icon in the system tray which will display a tooltip when the mouse hovers over it. The text of the tooltip you can specify with the --text=TEXT option.



This command takes one more option, --listen, which listens for data on stdin. Using this option is a little more difficult. stdin usually uses file descriptor 0, but we can send data through another file descriptor, using echo. The listen option expects one of three option-commands - tooltip, icon and visible - allowing us to dynamically control the displayed text, the icon in the system tray and the visibility of the icon, which is a useful way of getting feedback from the script to the user.

**zenity --notification --text="PCLinuxOS"**

will place the triangular warning icon in the system tray, as in the graphic above, with a tooltip announcing "PCLinuxOS."

When we use the --listen option, we can write

**exec 3> >(zenity --notification --text="PCLinuxOS" --listen)**

which sends all data using file descriptor 3 to the zenity command. File descriptor 3 has been used, as 0, 1 & 2 are already used by stdin, stdout and stderr, but I could have used, for example, 7 or even 27, with the same effect.

To change the icon to the "info-icon," we can echo that information through file descriptor 3:

**echo "icon: info" >&3**

which changes the icon like this:



To change the tooltip we would issue

**echo "tooltip: Radically Simple" >&3**

and we can keep on sending new information to the command in this way.

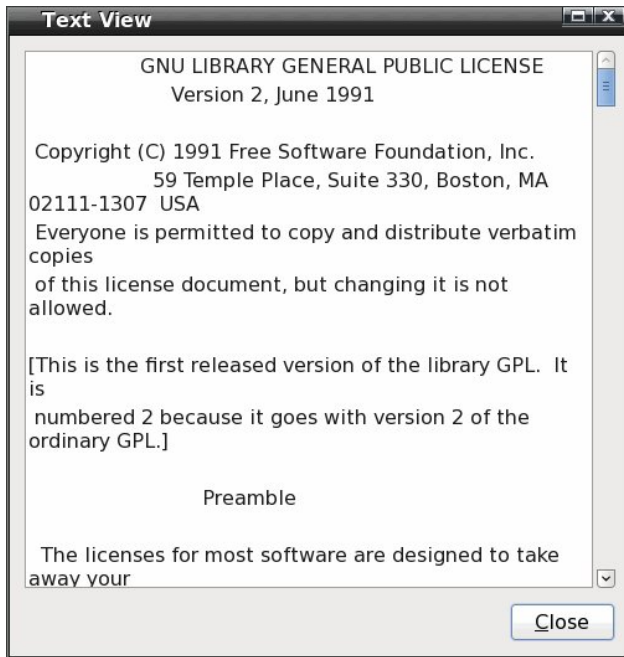
To end the notification command we simply need to close the file descriptor:

**exec 3>&-**

## Text Information

The text information dialog allows you to display text from a file to the user. The text can also be piped to the command from another command.

**zenity --text-info --filename=FILENAME**



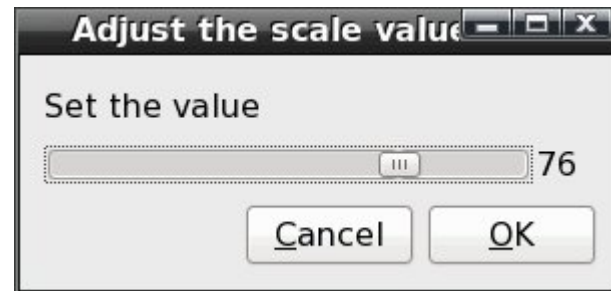
You may make the displayed text editable with the `--editable` option. The edited text is returned by the command as text which must be captured, as it is not written to the source file.

### Scale

The scale dialog displays a sliding scale for which you can specify the maximum, minimum and starting value, the step by which it increases and whether or not to display the current value. With the `--print-partial` option, you can echo the current value back to the calling program as to move the slider. Clicking the OK button closes the dialog and returns the

current value. The slider may be moved by the keyboard arrow keys or by the mouse, although in the latter case the step value is ignored.

**zenity --scale --min-value=0 --max-value=100 --value=76 --text="Set The Value"**



### List

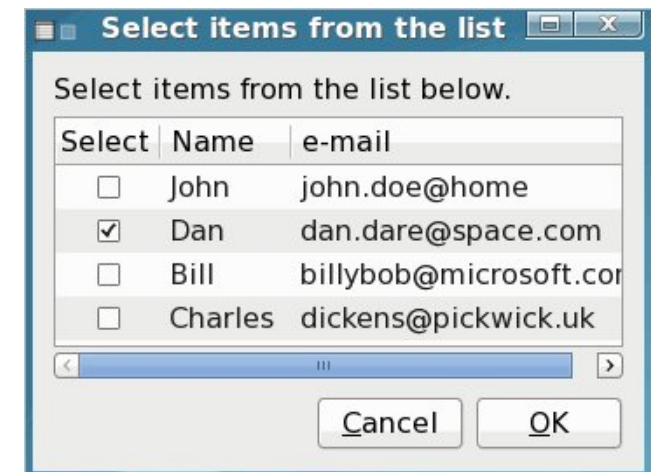
The list dialog has lots of options. You can set up a number of named columns, and supply data to be displayed under them, in rows. The user can select one or more rows and click the OK button to return the selected data to the script. By using the `--checklist` option, the first column of each row will contain a check box, which the user can click to select the row. All checked boxes return data to the script. The first item of data sent to each row should be either TRUE or FALSE to set the initial state of the box. The `--radiolist` option works in the same way and provides radio buttons.

By default, the command returns data from the first data column, but this can be changed with the `--print-column` option, since a value here of ALL returns the entire row. Returned data is separated by

the vertical bar character | but this can be changed with the `--separator` option. The `--editable` option enables the user to edit the data before returning it to the script with a click of the OK button.

I think an example is in order.

```
zenity --list --column="Select" \  
--column="Name" --column="e-mail" \  
FALSE "John" "john.doe@home" \  
TRUE "Dan" "dan.dare@space.com" \  
FALSE "Bill" "billybob@microsoft.com" \  
FALSE "Charles" "dickens@pickwick.uk" \  
--print-column=ALL \  
--separator="|" \  
--checklist \  
--height=240 \  
--width=350
```



## Progress

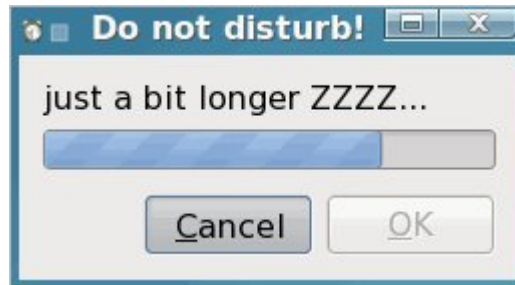
You can see a very good example of a progress dialog when you run the Synaptic package manager. At first glance the available options don't seem to offer a very wide choice but this little dialog can be quite impressive.

These are the options:

- text=STRING Set the dialog text
- percentage=INT Set initial percentage
- auto-close Close dialog when 100% has been reached
- auto-kill Kill parent process if cancel button is pressed
- pulsate Pulsate progress bar

And of course, all of the general options like width, height are also available. The data to the command is probably most easily piped in through a previous command, but you may also wish to feed it in through a file descriptor, as in the previous notification example.

```
#!/bin/sh
{
  echo "20" ; sleep 1
  echo "# ZZZZ..." ; sleep 1
  echo "50" ; sleep 1
  echo "# Not ready yet ZZZZ..." ; sleep 1
  echo "75" ; sleep 1
  echo "# just a bit longer ZZZZ..." ; sleep 1
  echo "75" ; sleep 1
  echo "# OK! I'm ready now" ; sleep 1
  echo "99" ; sleep 1
} |
zenity --progress \
  --title="Do not disturb!" \
  --text="Taking a nap..." \
  --percentage=0 \
  --auto-close \
  --width=250
```



This echoes text into the command, updating the progress bar as new data is sent in. Text prefixed with a # updates the --title option while the numbers update the --progress option. The sleep n command does nothing for n seconds so that you can see what's going on, but normally you would actually do something useful here. Another option is --pulsate, which causes the progress bar to slide back and forth for the duration of the command or an end of file character is received. The --auto-close option is used to automatically close the dialog when progress reaches 100% without requiring any user interaction.

When you use redirection to feed information to a dialog with a command like

```
ls /bin | (zenity --text-info)
```

then that information is absorbed by the zenity command. To overcome this, we can use the tee command. tee duplicates the data, sending it to multiple pipes.

```
ls /bin | tee >(zenity --text-info) >bin.txt
```

This will put the output of the ls command into the dialog, and also send it to the file bin.txt.



# Command Line Interface Intro: Part 9

by Peter Kelly (critter)

Almost everything that you do on the command line involves moving, changing, comparing or deleting text data. This data may reside in a file on a disk drive or be generated as the output from a previous command in the form of a data stream.

When the UNIX operating system was first developed at Bell Laboratories in the early 1970's, Ken Thompson, who is generally regarded as the chief architect of the project, was keen to implement a system of inter-connecting streams of data as an alternative to using a series of discrete processes to achieve the required output. Today we refer to this system as 'pipes' and 'redirection'.

The software tools available in those dark days were rather primitive, but have mostly survived and evolved into what we use today. The program `ed` has survived mostly unchanged in its usage since those times (which probably accounts for its lack of use today). `ed` is a line editor, unlike text editors such as `vi(m)`. A line editor reads in a file and works on one line at a time, not on the whole file. You make changes to the line and then move to another line.

Just as the program was simple, and so were the commands. You used `p` to print the line to the terminal so that you can actually see what you are editing (this is not done automatically), `d` to delete the line, `s` to substitute some text for some other text, but only in that line. To edit a large text file interactively, by hand, this is far too restrictive. So the text editors that we more commonly use today were developed.

With the introduction of pipes for streams of data through this method of editing line by line, non-interactively is ideal and so a new tool was introduced known as a stream editor. This reads in data and applies a series of commands to the data as it flows through. These commands, deletions, substitutions etc, could be supplied on the command line, or read in from a file or script. If the input data is from a file, then that file is not changed. Only the data in the output stream is affected, and this can be saved as a new file or further processed along the 'pipeline'.

As a model for this new tool, the `ed` editor was chosen and named `sed` - stream editor, which you may have heard of.

## SED

The `sed` utility retains a lot of the simplicity of commands it inherited from `ed`, but it adds a lot more functionality. Its command line or script can be a bewildering gibberish text when you first encounter it.

```
sed -n -e 's/M/ MegaBytes;/s/-.{12}|(.l.  
MegaBytes) |([0-9]{4})-|([0-9][0-9])-|([0-9][0-9])  
.... |(.*$)/4|3|2 |1 |5/p' sed-demo
```

If we break down this gibberish into manageable phrases, then it becomes more comprehensible. It really does, trust me.

Now before you throw your hands in the air and say "This is not for me!" let me say that it is very unlikely that you would ever need to construct such a complicated command.

Here's something a lot simpler and is actually useful. Many Linux users also use MS Windows, but if you try to read a Linux created text file in Windows, then you find that the line breaks don't work and extend to the full width allowed by the editor, probably notepad. This is because Linux terminates its lines with a newline character `\n`, while DOS and Windows need a newline and a carriage return pair `\n\r` (just like the old typewriters, where you move the paper up a line and push the carriage back to the beginning of the line). A newline on its own is not recognized as a line termination. `Sed` makes light work of this.

```
sed 's/$/\r/' linux-file > dos-file
```

 makes Linux files DOS-readable.

```
sed 's/\r/' /dos-file > linux-file
```

 converts them back, although this usually isn't necessary as Linux will disregard the extra carriage returns.

It would be a trivial matter to put these two files in a script and create a couple of aliases to them, perhaps `l2d` & `d2l`.

The syntax of `sed` is very simple:

```
sed {options} {commands} {file}
```

### options

The most useful options available for GNU `sed`, which is the version that Linux users will most likely find that they have installed, are:



- e is required when you are specifying commands on the command line and tells the application that what follows should be treated as a command to be applied to the current line. The option can be repeated on the command line to apply multiple commands to the input data.
- n silent mode, don't automatically print the lines to stdout.
- f script add the contents of the named script to the commands to be executed
- r use the extended set of regular expressions (like egrep)
- posix disable gnu extensions. This makes scripts portable to systems that have the standard unix-like version of sed
- help covers all the options for the version that you are using.

**commands**

These define what you do to the data as the stream passes through, and I will describe the use of the most important ones in the body of this text.

**file**

This is the input data stream, and if the file name is supplied on the command line, it is treated as stdin. That is, **sed command file** and **sed command < file** mean the same thing. The input may also be

piped in to the command, e.g. **cat file | sed command** or **ls -l | sed command**.

To get started, find or create a file to play around with. I have used a short listing of my /boot directory, in a file named sed-demo, **ls -ALLGgh /boot >sed-demo**, which looks like this:

```
total 31M
-rw-rw-r-- 1 440 2010-04-02 10:59 boot.backup.sda
-rw-r--r-- 1 111K 2010-04-03 15:11 config
-rw-r--r-- 1 108K 2010-03-16 15:11 config-2.6.32.10-pclos2.pae
-rw-r--r-- 1 111K 2010-04-03 15:11 config-2.6.33.2-pclos1.pae
-rwxr-xr-x 1 579K 2010-04-02 10:59 gfxmenu*
drwxr-xr-x 2 4.0K 2010-04-05 04:11 grub/
-rw----- 1 6.4M 2010-04-02 11:35 initrd-2.6.32.10-pclos2.pae.img
-rw----- 1 6.4M 2010-04-04 08:56 initrd-2.6.33.2-pclos1.pae.img
-rw----- 1 6.4M 2010-04-04 08:56 initrd.img
-rw-r--r-- 1 1.5K 2010-04-10 14:04
```

**kernel.h**

```
-rw-r--r-- 1 1.5K 2010-04-02 11:59 kernel.h-2.6.32.10-pclos2.pae
-rw-r--r-- 1 1.5K 2010-04-10 14:04 kernel.h-2.6.33.2-pclos1.pae
-rw-r--r-- 1 249K 2006-11-05 23:23 message-graphic
-rw-r--r-- 1 1.4M 2010-04-03 15:11 System.map
-rw-r--r-- 1 1.4M 2010-03-16 15:11 System.map-2.6.32.10-pclos2.pae
-rw-r--r-- 1 1.4M 2010-04-03 15:11 System.map-2.6.33.2-pclos1.pae
-rw-rw-r-- 1 256 2010-04-02 10:59 uk-latin1.klt
-rw-r--r-- 1 2.0M 2010-04-03 15:11 vmlinuz
-rw-r--r-- 1 2.0M 2010-03-16 15:11 vmlinuz-2.6.32.10-pclos2.pae
-rw-r--r-- 1 2.0M 2010-04-03 15:11 vmlinuz-2.6.33.2-pclos1.pae
```

This file contains a mixture of lines of varying length, and fields of differing construction. To select only data that meets certain criteria, and to re-format parts of it to more accurately meet my requirements, would be very difficult without a utility like sed.

The changes I want to make to this set of data are:

1. Remove the total count
2. Keep only regular files, no links, directories etc.
3. Remove the permissions fields
4. Remove the link counts
5. Keep only lines that contain files of 1MB or larger
6. Change 'M' to 'MegaBytes'
7. Change the date format from year-month-day to day/month/year
8. Remove the time field
9. Output the date size and file name - in that order.

Now that looks like a lot of work, but thanks to the flexibility of sed, I can do it in one command.

To get rid of the line 'total 31M' and leave only the lines with file details, I could issue the following command:

### sed -e '/total/d' sed-demo

This is the beginning of the output from this command.

```
jane@daisy > ~ $ sed -e '/total/d' sed-demo
-rw-rw-r-- 1 440 2010-04-02 10:59 boot.backup.sda
-rw-r--r-- 1 111K 2010-04-03 15:11 config
-rw-r--r-- 1 108K 2010-03-16 15:11 config-2.6.32.10-pclos2.pae
-rw-r--r-- 1 111K 2010-04-03 15:11 config-2.6.33.2-pclos1.pae
```

The line at the start of the listing that contained the expression total has disappeared from the output.

So what did I do here? I issued the sed command with the -e option, which told sed to treat the next

command line argument, '/total/d', as a command to apply to the input file sed-demo.

What sed did was to read in the entire sed-demo file line by line into an area of memory known as pattern space and examined each line to see if it could match the regular expression total, which is surrounded by a pair of slashes. Whenever a match was found, sed applied the d command, which deletes the current line from pattern space. This results in no output from sed from the analysis of that line. Lines that do not contain a pattern match are unaffected and flow through the command to stdout, which in this case is the terminal, as output has not been redirected elsewhere.

While that simple example of sed usage is not difficult to follow, the key phrase here is 'regular expression,' and a good understanding of regular expressions is required to make effective use of this command.

We covered the basics of regular expressions when we discussed the grep command, so perhaps a refresher is in order.

A regular expression is a sequence of literal characters and meta-characters. Literal characters are treated exactly as they are written and are case sensitive. Meta-characters have a special meaning in regular expressions, and must be expanded to produce the search pattern from the regular expression. These are the basic meta-characters:

- . The dot character matches any single character

- \* The asterisk matches zero or more occurrences of the preceding character. This is not the same behavior as the shell wild-card character.
- ^ The caret is a tricky one, it has two meanings. Outside of square brackets it means match the pattern only when it occurs at the beginning of the line, this is known as an anchor mark. As the first character inside a pair of brackets it negates the match i.e. match anything except what follows.
- \$ Another anchor mark this time meaning to only match the pattern at the end of a line.
- \<< \> More anchor marks. They match a pattern at the beginning \<< or the end \> of a word.
- \ The backslash is known as the escape or quoting character and is used to remove the special meaning of the character that immediately follows it.
- [ ] Brackets are used to hold groups, ranges and classes of characters. Here a group can be an individual character.
- \{n\} Match n occurrences of the preceding character or regular expression. Note that n may be a single number \{2\}, a range \{2,4\} or a minimum number \{2,\} meaning at least two occurrences.

\( \) Any matched text between \( and \) is stored in a special temporary buffer. Up to nine such sequences can be saved and then later inserted using the commands \1 to \9.

In the previous example, the regular expression we used, total, contained only literal characters. But more usually, you will build up a regular expression from literals, meta-characters and character classes such as [:digit:] or [:space:]. The use of meta characters in regular expressions enables you to very quickly match quite complicated or unknown patterns. Some examples:

**sed -e '/^#/d' .bashrc** would strip out any comments from your .bashrc file as comments begin with a #.

**sed -e '/^\$/d' .bashrc** would remove any blank lines by matching the beginning and end of the line with nothing in between.

It is quite safe to try these out since the source file is not altered. Only the output to the terminal is changed.

In my test file, I have one directory, /grub, and this is denoted by the letter d at the beginning of the line. To remove the line, we can use sed's delete command with a regular expression that matches only that line.

**sed -e '/^d/d' sed-demo** matches all lines beginning, that's the ^, with d, and applies the delete command. The command is single quoted to prevent shell expansion of meta characters. Recall that single quotes are known as strong quotes, and

protect the contents from the effects of shell expansion, which wouldn't have had any effect here, but it is a good habit to get into.

To keep directories and remove all other lines, we need to reverse the effect of the command, which we can do with:

**sed -n -e '/^d/p' sed-demo**

The -n turns off automatic echoing of pattern space to the terminal, and the p command, on finding a pattern match, prints the current contents of pattern space to stdout which, as the output has not been re-directed, is the terminal.

Alternatively we could look for lines that begin with a hyphen, and that would also exclude anything that wasn't a regular file.

**sed -n -e '/^-/p' sed-demo**

image  
The directory line has been removed but so has the total line as that also did not begin with a hyphen. In this case it helps, but we have to be extremely careful about what we want to include or exclude. Similarly, we can remove all lines that do not contain an uppercase M followed by a space to keep only files of one MB or larger. Without the space, the total line would be included, as that also contains an uppercase M but no trailing space.

**sed -n -e '/M /p' sed-demo**

```
jane@daisy > ~ $ sed -n -e '/^-/p' sed-demo
-rw-rw-r-- 1 440 2010-04-02 10:59 boot.backup.sda
-rw-r--r-- 1 111K 2010-04-03 15:11 config
-rw-r--r-- 1 108K 2010-03-16 15:11 config-2.6.32.10-pclos2.pae
-rw-r--r-- 1 111K 2010-04-03 15:11 config-2.6.33.2-pclos1.pae
-rwxr-xr-x 1 579K 2010-04-02 10:59 gfxmenu*
-rw----- 1 6.4M 2010-04-02 11:35 initrd-2.6.32.10-pclos2.pae.img
-rw----- 1 6.4M 2010-04-04 08:56 initrd-2.6.33.2-pclos1.pae.img
```

In the first example, I matched the pattern total to remove the first line, but I could more easily have specified an address.

**sed -e '1d' sed-demo** the 1 is the line number that I want to remove. Addresses can be ranges, so sed -e '8,20d' will remove lines 8 to 20 from the output.

```
jane@daisy > ~ $ sed -n -e '/M /p' sed-demo
-rw----- 1 6.4M 2010-04-02 11:35 initrd-2.6.32.10-pclos2.pae.img
-rw----- 1 6.4M 2010-04-04 08:56 initrd-2.6.33.2-pclos1.pae.img
-rw----- 1 6.4M 2010-04-04 08:56 initrd.img
-rw-r--r-- 1 1.4M 2010-04-03 15:11 System.map
-rw-r--r-- 1 1.4M 2010-03-16 15:11 System.map-2.6.32.10-pclos2.pae
-rw-r--r-- 1 1.4M 2010-04-03 15:11 System.map-2.6.33.2-pclos1.pae
-rw-r--r-- 1 2.0M 2010-04-03 15:11 vmlinuz
-rw-r--r-- 1 2.0M 2010-03-16 15:11 vmlinuz-2.6.32.10-pclos2.pae
-rw-r--r-- 1 2.0M 2010-04-03 15:11 vmlinuz-2.6.33.2-pclos1.pae
```

Notice that the total line and the directory line are still in the output, as the original data has not been altered.

In this case, I knew the address was 1, but usually you have to search for it. You do this by specifying a regular expression surrounded by slashes. The address of the line to delete in the first example was given by matching the regular expression /total/

### Substitution

Now that we have a means of keeping only those lines that we want in our final data set, we need to change some of that data.

Probably the most used command for sed is s, to substitute one regular expression for another. The format for this is:

**sed -e 's/old/new/' {file}**

So that the command

**sed -e 's/M/ MegaBytes/' sed-demo**

would change all the uppercase Ms to ' MegaBytes' (note the preceding space) in my test file. Note that sed by default only matches the first occurrence of the pattern on each line. If you need to match every occurrence, which is often what you want, then you have to add the g-global command:

**sed -n -e 's/r/R/p' sed-demo** would only replace the first r with R.

**sed -n -e 's/r/R/gp' sed-demo** replaces every occurrence.

Combining two commands we can make a substitution and output only the lines that we want to keep.

**sed -n -e 's/M/ MegaBytes/' -e '/Mega/p' sed-demo**

```
jane@daisy > ~ $ sed -n -e 's/M/ MegaBytes/' -e '/Mega/p' sed-demo
total 31 MegaBytes
-rw-r--r-- 1 6.4 MegaBytes 2010-04-02 11:35 initrd-2.6.32.10-pclos2.pae.img
-rw-r--r-- 1 6.4 MegaBytes 2010-04-04 08:56 initrd-2.6.33.2-pclos1.pae.img
-rw-r--r-- 1 6.4 MegaBytes 2010-04-04 08:56 initrd.img
-rw-r--r-- 1 1.4 MegaBytes 2010-04-03 15:11 System.map
-rw-r--r-- 1 1.4 MegaBytes 2010-03-16 15:11 System.map-2.6.32.10-pclos2.pae
-rw-r--r-- 1 1.4 MegaBytes 2010-04-03 15:11 System.map-2.6.33.2-pclos1.pae
-rw-r--r-- 1 2.0 MegaBytes 2010-04-03 15:11 vmlinuz
-rw-r--r-- 1 2.0 MegaBytes 2010-03-16 15:11 vmlinuz-2.6.32.10-pclos2.pae
-rw-r--r-- 1 2.0 MegaBytes 2010-04-03 15:11 vmlinuz-2.6.33.2-pclos1.pae
```

The substitute command can also be used to good effect to delete a part of the line. To remove the time field, we could match two characters followed by a colon followed by two characters and replace it with something like this:

**sed -e 's/..:..!/' sed-demo**

To remove the permissions, the link count and trailing space at the start of the line, we could match a hyphen followed by exactly 12 characters by using the dot, { & } metacharacters.

**sed -e 's/-.{12}!/' sed-demo**

The dot matches any character, and the number inside the escaped braces tells the command how many matches to make. In other words, match exactly 12 characters.

One thing to be aware of when using regular expressions with meta characters is that they are greedy. They will always try to match the longest possible string.

If you try to remove the permissions with a command like this:

**sed -n -e 's/.\*-!/' sed-demo**

looking for a hyphen followed by some characters followed by a hyphen, then you might be disappointed to see that it matched strings like this **-rw-r--r-- 1 111K 2010-04-03 15:11 config-2.6.33.2-** and output only **pclos1.pae**

Putting all this together makes for a pretty long command line, so I have used the shell line continuation character, the backslash, to make it more legible. But remember that it is all one line, as far as the shell is concerned.

```
jane@daisy > ~ $ sed -n -e 's/-.{12}!/' \
> -e 's/M/ MegaBytes/' \
> -e 's/..:..!/' \
> -e '/Mega/p' \
> sed-demo
total 31 MegaBytes
6.4 MegaBytes 2010-04-02  initrd-2.6.32.10-pclos2.pae.img
6.4 MegaBytes 2010-04-04  initrd-2.6.33.2-pclos1.pae.img
6.4 MegaBytes 2010-04-04  initrd.img
1.4 MegaBytes 2010-04-03  System.map
1.4 MegaBytes 2010-03-16  System.map-2.6.32.10-pclos2.pae
1.4 MegaBytes 2010-04-03  System.map-2.6.33.2-pclos1.pae
2.0 MegaBytes 2010-04-03  vmlinuz
2.0 MegaBytes 2010-03-16  vmlinuz-2.6.32.10-pclos2.pae
2.0 MegaBytes 2010-04-03  vmlinuz-2.6.33.2-pclos1.pae
```

The skill in using sed is recognizing what you want to match, and the building of a regular expression that matches that part of the line, and only that part. This comes with practice and an understanding of regular expressions. Matching the file name is quite tricky, as there seems to be no 'standard format' that could be easily matched. So the easiest way out is to match everything else.

**sed -e 's/-.{34}!/' sed-demo**

Here's the start of the output:

```
jane@daisy > ~ $ sed -e 's/-.{34}!/' sed-demo
total 31M
boot.backup.sda
config
config-2.6.32.10-pclos2.pae
config-2.6.33.2-pclos1.pae
gfxmenu*
```



No, I didn't count all 34 characters. I took a guess, tried it and adjusted it. This trial and error method is quite common when building regular expressions, although not everyone admits it.

If you noticed that the total line is in the output, it is because it doesn't begin with a hyphen and we hadn't already removed it. The order of operation of your commands can have a great effect on the resultant output.

**sed -e 's/[0-9]{4}-[0-9][0-9]-[0-9][0-9]//'** sed-demo matches the date part of the line and removes it. Here's how it works.

The first slash starts the search expression.

`[0-9]{4}` matches exactly 4 digits

- matches a literal hyphen

`[0-9][0-9]` matches 2 digits

- matches a literal hyphen

`[0-9][0-9]` matches 2 digits

The second slash ends the search expression.

When a match is found, it is replaced by whatever is between the second and third slash. In this case, nothing.

For example, in the second line of the test file, 2010-04-02 is a match and will be removed.

Now we have most of the methods needed to complete the task, but the command line is getting rather clumsy.

We could write a script and put all of the commands in there, but there is another way using the meta characters `\(` and `\)`.

Any thing that matches the regular expression that appears between this pair is stored and can be recalled for later inclusion. A match from the first pair can be recalled with `\1`, the second pair with `\2` and so on up to `\9`.

Here's the final command which turns off automatic line echoing, matches the entire line storing parts of it and then outputs some of those stored parts in the required order.

**sed -n -e 's/M/ MegaBytes/' -e 's/-.{12}\(.\. MegaBytes\) \([0-9]{4}\)\-([0-9][0-9])\-([0-9][0-9])\ ... \(.\*\$)/\4/\3/\2 \1 \5/p'** sed-demo

Another way of writing this command without repeating the `-e` option is to separate the commands with a semicolon

**sed -n -e 's/M/ MegaBytes';s/-.{12}\(.\. MegaBytes\) \([0-9]{4}\)\-([0-9][0-9])\-([0-9][0-9])\ ... \(.\*\$)/\4/\3/\2 \1 \5/p'** sed-demo

Personally, I find this harder to follow, but the choice is yours.

This is easier to follow if I break it down. Just to remind you, here's a typical line from the input file.

```
jane@daisy > ~ $ sed -n -e 's/M/ MegaBytes/' -e 's/-.{12}\
}{\.\. MegaBytes\) \([0-9]{4}\)\-([0-9][0-9])\-([0-9][
0-9])\ ... \(.*$)/\4/\3/\2 \1 \5/p' sed-demo
02/04/2010 6.4 MegaBytes initrd-2.6.32.10-pclos2.pae.img
04/04/2010 6.4 MegaBytes initrd-2.6.33.2-pclos1.pae.img
04/04/2010 6.4 MegaBytes initrd.img
03/04/2010 1.4 MegaBytes System.map
16/03/2010 1.4 MegaBytes System.map-2.6.32.10-pclos2.pae
03/04/2010 1.4 MegaBytes System.map-2.6.33.2-pclos1.pae
03/04/2010 2.0 MegaBytes vmlinuz
16/03/2010 2.0 MegaBytes vmlinuz-2.6.32.10-pclos2.pae
03/04/2010 2.0 MegaBytes vmlinuz-2.6.33.2-pclos1.pae
```

**-rw-rw-r-- 1 440 2010-04-02 10:59  
boot.backup.sda**

`sed -n -e 's/M/ MegaBytes/'`

The first expression is a substitution replacing 'M' with a space, followed by the word MegaBytes

`-e 's/-.{12}`

The second expression is also a substitution, replacing a hyphen, followed by exactly 12 characters.

`\(.\. MegaBytes\)`

Followed by a character, a dot (which has to be escaped to retain its literal meaning), another character, a space, and the word 'MegaBytes'. All of the matching data is stored in `\1`. This is the size.

`\([0-9]{4}\)`

followed by exactly four digits, which are stored in `\2`. This is the Year.

-\([0-9][0-9]\)

followed by two digits, is stored in \3, and represents the month.

-\([0-9][0-9]\)

followed by two digits, is stored in \4, and represents the day.

....

followed by a space, two characters, a colon, two more characters and a space. This is the time, but as we don't use it, it isn't stored.

\(.\*\$\)/

followed by any number of characters that end the line, hence the dollar sign. This is the file name, and is stored in \5. This ends the search section of the substitution.

This is what we are going to replace the data we just matched with:

\4 First \4 - the day

\ followed by a forward slash that has to be escaped, or else we would terminate the substitution command

\3 followed by \3 - the month

\ followed by a slash

\2 followed by \2 the year

\1 followed by a space and \1 - the size

\5/p' and finally a space and \5 the file name, the substitution command terminating slash and the p command to print out the substituted data.

sed-demo This is the name of the file that we want sed to process.

Easy peasy :)

Of course in real life, that is far too long a command to enter on the command line. Normally, such a complex operation would be written to a file and referenced by sed with the -f option.

With our commands in a file, we can easily test and adjust until we get the required result. We can also group multiple commands that you want to apply to the same address, or to each line, by placing them within braces. In the previous example we changed M to Megabytes, but more realistically, we might want to change M to MB and K to KB. If we create a file named sed-file (call it what you like) with the following text:

```
{
    s/K/KB/
    s/M/MB/

    p
}
```

Then execute:

### sed -n -f mysed-file sed-demo

As we haven't specified an address before the opening brace, all lines will be processed by applying both substitutions to each line, and then printing them to the terminal.

Lets try another test file. Here's a very simple html file that lends itself nicely to reconfiguration by sed. I've named it 2010.html. Don't worry if you don't know any html code. You only need to know that the things in the «» affect the look and format of the web page. «p» starts a paragraph and «/p» ends it. These are known as tags.

«body»

```
<h1>PCLinuxOS 2010
Release</h1>
<p>Texstar recently anounced
the release of the 2010 version of
this popular distribution.</p>
<h2>Now available in the
following versions</h2>
<p><li><em>KDE4</em> The base
distribution</li></p>
<p><li><em>Minime</em> Minimal
KDE4 installation</li></p>
<p><li><em>Gnome</em> Full
installation of Gnome</li></p>
<p><li><em>ZenMini</em>
Minimal Gnome distribution</li></p>
<p><li><em>LXDE</em> A
lightweight desktop</li></p>
<p><li><em>Phoenix</em> The
XFCE desktop</li></p>
<p><li><em>Enlightenment</em>
The beautiful e17 desktop</li></p>
```

**Openbox** Suitable for older hardware

This is how it appears in Firefox:

### PCLinuxOS 2010 Release

Texstar recently announced the release of the 2010 version of this popular distribution.

#### Now available in the following versions

- *KDE4* The base distribution
- *Minime* Minimal KDE4 installation
- *Gnome* Full installation of Gnome
- *ZenMini* Minimal Gnome distribution
- *LXDE* A lightweight desktop
- *Phoenix* The XFCE desktop
- *Enlightenment* The beautiful e17 desktop
- *Openbox* Suitable for older hardware

The emphasized or italic text is turned on and off by the `<em>` and `</em>` tags. To change this to bold, we need to replace the `em` with `b`.

To do that change only for the KDE based distributions, we need to supply a start and end address, and write the output to another file.

```
sed -e '/KDE4/,/Minime/s/em>/b>/g' 2010.html >2010b.html
```

The beginning address is the first match of KDE4, and the ending address is the first match of Minime.

Voila! all done.

### PCLinuxOS 2010 Release

Texstar recently announced the release of the 2010 version of this popular distribution.

#### Now available in the following versions

- **KDE4** The base distribution
- **Minime** Minimal KDE4 installation
- *Gnome* Full installation of Gnome
- *ZenMini* Minimal Gnome distribution
- *LXDE* A lightweight desktop
- *Phoenix* The XFCE desktop
- *Enlightenment* The beautiful e17 desktop
- *Openbox* Suitable for older hardware

sed has three more commands that aren't used very much, and because of their unusual two line syntax, are best applied from a file. They are `a` - append, `i` - insert and `c` - change.

Our new html test file has one sub-heading, which is identified by the `<h2></h2>` pair. Normally, there would be many such headings, and possibly a folder of many html files. In such a case, the overall look of a website can be completely changed with a small sed script. Our little test file will suffice to show the operation of these commands.

If I create a file with this text and name it sed-file

```
<h2>{
il
_____
al
_____
}
```

And then issue the command:

```
sed -f sed-file 2010.html > 2010new.html
```

Then any line that has the `<h2>` tag (that's the address to which the group of commands between the braces will applied) will have a row of underscores inserted before it and appended after it.

The `c` command works in the same manner, changing any matching line or lines with the supplied text. If the address supplied covers a range of lines, then the entire block of text is replaced with a single copy of the new text.

In addition to the commands that I have covered here, sed has many more that would stretch us beyond an introductory text. There are flow control commands such as `b` - branch which enable scripts to loop under certain conditions, labels to which we can jump to perform certain operations dependent upon the outcome of a previous one, and is usually determined by the `t` - test command.

There are also a group of commands to manipulate an area of memory known as hold space. sed reads input lines into the area of memory known as pattern space and some, or all, of that data can be temporarily copied to hold space as a sort of scratch pad. sed can't operate on the contents of hold space. It can only add to it, read from it or swap one with the other. The commands are: `g`, `G`, `h`, `H`, `x`. They are known as `get` (from pattern space), `hold` (in pattern space) `end exchange` (swap). The uppercase versions append, and the lowercase versions overwrite the data.

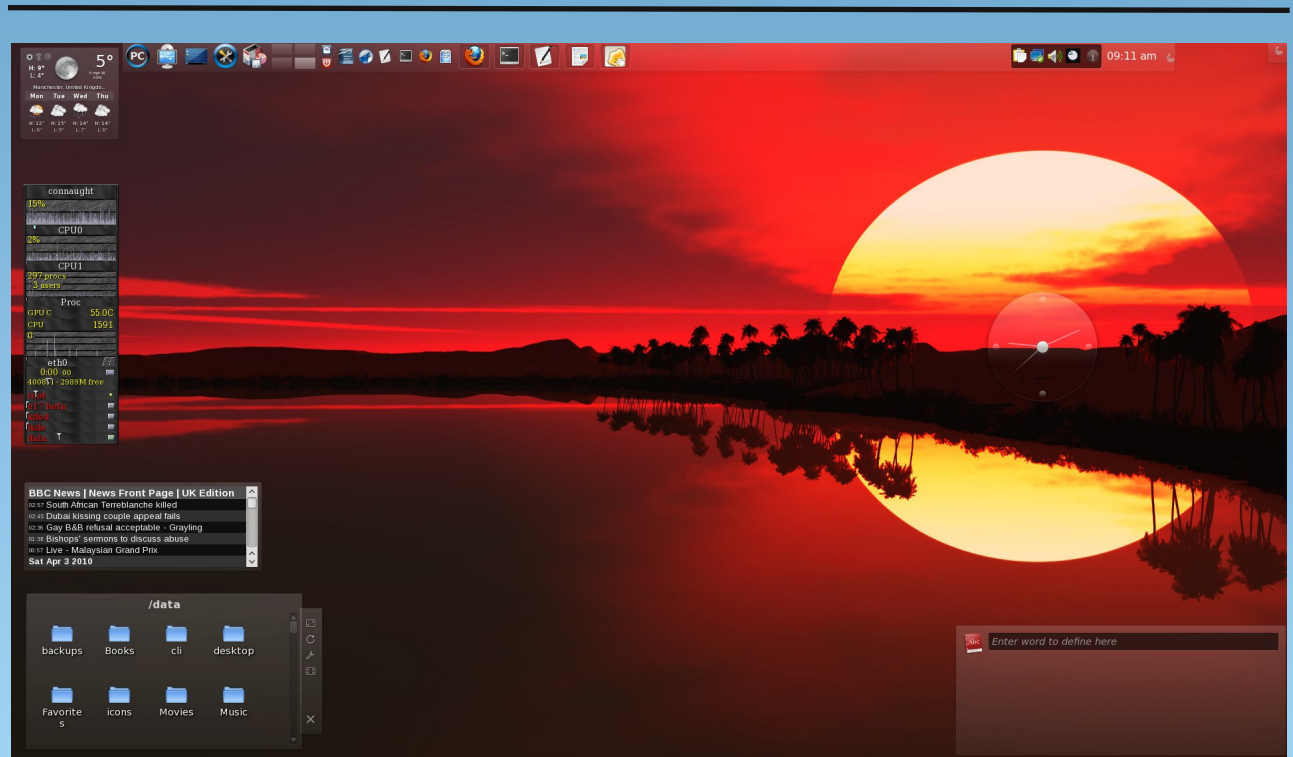
The last command that I want to mention here is `y` which, in a not very intuitive way, means transform the characters in one string to the characters in another string by character position. The `n`th character in the first string is replaced by the `n`th character in the second string. As usual, an example shows this better.

To force a line to lowercase so that there will be no misunderstandings when the shell is interpreting a script:

`sed 'y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/' mysript.sh`



# Screenshot Showcase



*Posted by critter, April 4, 2010, running PCLinuxOS KDE 4.*



# Command Line Interface Intro: Part 10

by Peter Kelly (critter)

With the release of the Unix operating system in the early 1970's, there was finally a solid operating system and a set of tools that had been written to utilize the advanced features that it supported. The computing community welcomed it, and some of the tools raised a lot of interest. One of these was the C programming language developed by Dennis Ritchie to enable the system and the tools to be so rapidly developed. Another one was sed, the stream editor. Because of the interest generated by sed and some of the other text manipulation tools, three of the engineers at Bell Laboratories set about developing a programming language that would refine and simplify some of the work done with these tools. It was released in 1977 under the name awk, which was derived from the initials of the three developers Alfred Aho, Peter Weinberg and Brian Kernighan. Brian had worked with Dennis Ritchie on the C Language (*The basic C language is still known as K & R C*), and a lot of the structure of C found its way into awk.

Awk was written to enable quick and dirty commands to be constructed to strip and reformat lines of text, but by the mid 1980's so much was being done with this program, much to the surprise of the authors, that it was re-visited to become **nawk** (new awk). Much more programming functionality was added to help it become the scripting utility that we have today. Linux users will most likely have gawk, which is similar enough to **nawk** as to make no difference to most users.

You may see awk written as 'awk' and as 'AWK'. It is generally agreed that awk is the interpreter program

for awk scripts, and AWK is the scripting language used in those scripts.

## AWK

Alfred Aho, one of the developers, described awk like this:

*"AWK is a language for processing files of text. A file is treated as a sequence of records, and by default each line is a record. Each line is broken up into a sequence of fields, so we can think of the first word in a line as the first field, the second word as the second field, and so on. An AWK program is a sequence of pattern-action statements. AWK reads the input a line at a time. A line is scanned for each pattern in the program, and for each pattern that matches, the associated action is executed."*

This pretty much sums up what it does, but doesn't even begin to do justice to the power and flexibility of the language - as we shall see.

Using awk need not be a complicated affair. It can be a simple one line command issued at the console. **awk '{ print \$1 }' test** would print out the first word or 'field' on each line of the file test. The variables **\$1, \$2 ...** etc. are assigned to the corresponding fields in a record. The variable **\$0** contains the entire input line/record, **NF** the number of fields in the current record and **NR** the number of the current record.

We should pause here and be clear about with what it is that we are working.

A 'word', which is also referred to as a 'field', is not only a language word it is a contiguous group of

characters terminated by white space or a newline. White space is one or more spaces or tabs, and is the default **field separator**. This can be changed to any arbitrary character by use of the **-F** option (uppercase) on the command line, or by setting the variable **FS** in a script. **awk -F":" '{print \$1}' /etc/passwd** changes the field separator to a colon and prints out the first field of each line in the file /etc/passwd which provides us with a list of all named users on the system.

A record is a group of fields and can be considered as a card in a card index system. The data on the card can be details from a directory listing, a set of values from the result of some test or, as we have seen, a line from the system /etc/passwd file. The variable **RS** contains the **record separator**, which by default is set to a newline **\n**. Changing the value of **RS** enables us to work with multi-line records

The command line syntax of the awk command is as follows:

**awk {options}{pattern}{commands}**

The options for awk are:

<b>-F</b>	to change the field separator
<b>-f</b>	to declare the name of a script to use
<b>-v</b>	to set a variable value.

We could have used **-v FS=":"** to change the field separator.

There are some others, but as most awk usage is done in a script, they are little used.

**pattern** is a regular expression set between a pair of forward slashes as in **sed** and is optional. If omitted, the commands are applied to the entire line.

**commands** are also optional, and if omitted, any line that matches **pattern** will be printed out in its entirety, unchanged.

If both pattern and command are omitted then you will get a usage reminder, which is no more than you deserve.

If using awk in a shell script, then its use is more or less as on the command line.

An awk script is called in one of two ways

1. Create a script file named awkscript or whatever:

```
{
FS=":"
print $1" uid="$3
}
```

Call it with the -f option: **awk -f awkscript /etc/passwd**

2. Add a line like this as the first line of the script:

```
#!/bin/awk -f
```

I prefer to give files like this an 'awkish' name - **uid.awk**  
Make it executable **chmod +x uid.awk**  
Call it like this: **./uid.awk /etc/passwd**

The **#!** line must contain the actual address of the awk executable, which you can check with the command **which awk**.

Actually, if you are running Linux, **awk** is more than likely a symbolic link to gawk, the gnu version of awk which has a few extras, but everything here will work with either version - unless otherwise stated. If you want to know which one you are actually using, the command **awk --version** will tell you.

In the script we just created, everything between the braces is executed once for each line of the input file or each record. We can also have a 'header' and a 'footer.' These are known as the **BEGIN** and **END** blocks. This is where we put code that we want to execute just once at the beginning or at the end of execution. The **BEGIN** block is where we would normally initialize variables such as **FS**, and the **END** block can be used to print a final completion message or summary of completed commands.

The script then consists of three sections:

```
BEGIN{
commands}
```

```
{
    command
    |
    This is the main part of the script
    |
    command}
```

```
END{
commands}
```

All of the sections are optional, although omitting all three would prove pretty pointless. The following code prints out the name of all users on the system who have bash as their default shell.

```
#!/bin/awk -f

BEGIN {
FS=":"}

$7 == "\/bin\/bash" {
    print $1}
```

Note that the slashes need to be escaped. Here, I have used two equal signs as the equality symbol, but awk also uses the tilde ~ symbol to match regular expressions. Normally, we use this as shorthand for our home directory.

### But what does it do?

Well it processes text and other data.

Yes, **sed** does that, but if you liken sed to the search and replace functions in a word processor, then with awk you can add to that the programming power of a high level language like C, floating point calculations including trigonometry and exponentiation, string manipulation functions, user defined functions, data retrieval, boolean operators and multi-dimensional and associative arrays. Unix/Linux commands often generate tabulated test output, and awk is the ideal tool to generate reports from this type of data, easily

providing a header, selecting and manipulating selected parts of the data and then providing a final section to calculate and output a summary.

In short then, awk is a report generator and advanced scripting language that will do almost anything, although without some serious hardware modifications, it will not make your coffee.

With such a complex program as awk, it would be reasonable to assume that learning to use it was going to be an uphill struggle, but fortunately this is not the case. If you have followed along so far through shell scripting, regular expressions and sed, then you have already covered most of the hard work. There are some differences, but nothing that will hurt your brain.

### Simple awk scripts

Although awk can be, and often is, used on the command line, it becomes most useful when used in a script. The script can be saved and easily modified to perform other similar tasks.

Suppose we wanted to know which ext file systems were listed in /etc/fstab, and where they would be mounted. We can do this easily with awk using an if conditional statement. I have used a nested statement here to ensure that comments are excluded.

```
#!/bin/awk -f
{if ( $1 !~ "^#" ) { # ignore comment lines
  if ( $3 ~ "ext." ) {
    print $3 " Filesystem mounted at "$2 " "
  }
}
```

This reads 'If the first field does not begin with a #, then if the third field contains "ext" followed by one other character, then print out the file system type and its mount point.'

This is the output on my machine from the command `./awk1.awk /etc/fstab`

```
ext3 Filesystem mounted at /
ext3 Filesystem mounted at /data
ext3 Filesystem mounted at /mnt/e17
ext3 Filesystem mounted at /mnt/icybox
ext3 Filesystem mounted at /mnt/kde3
ext3 Filesystem mounted at /mnt/kde4
ext3 Filesystem mounted at /mnt/lxde
ext4 Filesystem mounted at /mnt/phoenix
```

awk is often thought of as an alternative to sed, and it can indeed be used as such. Which one you use depends upon what you need to do. Remember the tortuous route we had to go in sed to output the size, re-formatted date and file name from a directory listing?

```
sed -n -e 's/M/ MegaBytes/' -e 's/-.{12}\)(\.\.
MegaBytes\)\|([0-9]{4})-|([0-9]{0-9})|-[0-9]{0-9}]
.... \|(*$)/4\|3\|2 \|1 \|5/p' sed-demo
```

To do this in an awk script we can start by only considering records (lines) that:

```
1 #!/bin/awk -f
2
3 /^-/ {
4   if (NF ~ 6) {
5     sub(/M/, " MegaBytes", $3)
6     split ($4, fdate, "-")
7     if ( $3 ~ /Bytes$/ ) {
8       print fdate[3] "/" fdate[2] "/" fdate[1] " " $3 " " $6 }
9   }
10 }
```

start with a hyphen (line 3), and that contain 6 fields (line 4)

In line 5, we call the built in function `sub()`, which substitutes "MegaBytes" for the "M" in the third field.

In line 6, we call another built in function called `split()`. This splits up the fourth field, the date field, using a hyphen as the field separator, and stores each part as an element of an array named `fdate`.

Line 7 restricts operation to only those lines where the third field ends in "BYTES."

Line 8 prints out the re-formatted date, pulling the elements from the array, followed by the size and file name fields.

Even though the script contains a lot of material that you have never seen, I believe it is a lot less daunting than its sed counterpart, and the output is identical.

Of course awk can also be called from a shell script, and indeed many system scripts make extensive use of awk. There is an important concept to consider when calling awk from within a shell script. In a shell script, the \$ indicates a variable name such as \$USER, whereas in awk, it references a field, such as \$2, which refers to the second field in a record. When you call awk in a shell, the awk commands must be single quoted to protect them from shell expansion. If you passed awk the command `'{ print $USER }'` expecting the output to be the users name as the shells echo command would output, you would be in for a surprise.

Awk does not see the variable, but sees a reference to field number 'USER'. As USER is not defined, it has a zero value, hence \$0, and the entire record is output.

```
#!/bin/bash
owner=3
group=4
filename=9
echo owner = $owner
echo group = $group
echo filename = $filename
echo
echo

ls -l /home/$USER/Documents | awk '
BEGIN { print "Owner\tGroup\tFile Name" }
{ if ( NF ~ 9 ) {
print '$owner' "\t" '$group' "\t" '$filename' }
}
END { print "\nAll Done" }
```

In this bash script, we pipe the output from a directory listing of the users home directory into an awk command, which outputs the owner, group and file name of each entry.

The first part of the script assigns the values to the variables, and then echoes them to screen to show the values that they have in that part of the script. The directory listing is then piped into the awk command, which has a **BEGIN** section to print a header, the main section has a single print statement which is applied to all input lines with 8 fields and an **END** section to end the report.

Syntax highlighting shows how the quoting is turned on and off to allow or deny shell expansion.

Each unquoted variable is expanded to its value so that **print '\$owner'** becomes **print \$3** (the third field). Two \$ signs are required, as **\$owner** is seen as simply **3**. The **lt** in the command is a tab character.

```
jane@daisy > ~ $ ./owners.sh
owner = 3
group = 4
filename = 9

Owner Group File Name
jane jane addresses
jane musicclub club_letter_1
jane musicclub club_letter_2
jane jane contacts
jane jane jan_report

All Done
```

The use of built in functions, as used in the script, demonstrate some of the power available in awk. So perhaps we should look at some of the available functions, what they do and how we call them.

awk's built in functions can be grouped as:

- text manipulation functions
- numeric functions
- file handling functions

File handling functions in awk are best left alone. Use something more suitable wherever possible. If you need to control the opening and closing of files, call the awk command from within a shell script and let the shell control the files. Shells excel at file handling.

Integer numeric functions included in awk are quite complete, and should satisfy the needs of most people. Floating point operations in awk are fine if you need them, or as a work around to the shell's inability to handle floats, but remember to return the value to the shell as a text string. I have found little use for these functions in awk, despite my daily work requiring a considerable amount of mathematics, there are always better tools for this, just as you wouldn't write a letter in a spreadsheet even though it is possible.

Text manipulation functions are really what awk is all about, so I'll start with those.

Substitution is a common task, and awk provides three functions to achieve this:

**sub()** and **gsub()**. These are analogous to the **s** command in sed and the **s** command with **g**(lobal) option.

**gensub()** This a general substitution function in gawk. It is not found in the original awk, so beware if your code is meant to be portable.

The first two functions are called by **sub(/regexpl, replacement, search-target)**. This is like saying "substitute( whatever-matches-this, with-this, in-this)."

The 'in-this' part is where to search for the match, and can be a variable (\$myvar), a reference to a field (\$1) or an element of an array (array[1]). If omitted, then \$0 (the entire record) is searched. Note that if the search-field is omitted, then omit the second comma or you will get an error.



This enables you to easily replace a particular occurrence where multiple matches may be possible within a record.

The **gsub()** function works identically with the search target, restricting the 'global' replacement to a particular part of the record.

The **gensub()** function is called by **gensub(/regexp/, replacement, how, search-target)**.

The parameter how is new. If it is **g** or **G**, then all matches are replaced. If it is a number, then only the match corresponding to that number is replaced.

**sub()** and **gsub()** modify the original string as it passes through, as demonstrated in our first little script where 'M' was changed to ' MegaBytes'. (The string or record is modified, not the original file).

**gensub()** does not alter the original string, but **returns** it as the result of the function. Therefore, an assignment is required to make use of the changes.

```
#!/bin/awk -f

{ owner = gensub(/jane/, "me", "1" )
print owner
print $0
}
```

This changes the first occurrence of the string "jane" to "me," and returns the result in the variable "owner."

As the first occurrence of "jane" is in the third field of the file listing, we can see that "owner is indeed "me," but the original third field \$3 is unchanged, as we can see by printing out \$0 - the original input record.

```
jane@daisy > mydir1 $ ls -l m* | ./awkscript.awk
-rwxr-xr-x 1 me jane 908 Nov 10 2009 myfile1*
-rwxr-xr-x 1 jane jane 908 Nov 10 2009 myfile1*
```

Instead of assigning the result of the function to a variable, it can be assigned directly to the print command like this:

```
print gensub(/jane/, "me", "1" ).
```

**split()** is another function used in the first example and is an extremely convenient tool to have around.

**Split( string\_to\_split, name\_of\_an\_array, separator )**

It takes a string, specified as the first parameter, searches for what is specified as a separator in the third parameter, and stores each separated 'chunk' as an element of the array specified as the second parameter. The separator can be a single character or a regular expression. If it is omitted from the command, then the current value of the awk variable **FS** is used. If it is the empty string "", then each individual character is stored in a separate element of the array. The return value of the function is the number of elements in the array.

This is great for tasks like dealing with names and addresses, or for converting a numerical value into its text equivalent.

In this example we feed a date to the script in a space separated numeric form and output the date with a textual month.

```
#!/bin/awk -f
BEGIN {
shortmonth = "Jan Feb Mar Apr May Jun Jul"
shortmonth = shortmonth " Aug Sep Oct Nov Dec"
split (shortmonth, mnth)
}
{ print $1 " " mnth[$2] " " $3
}
```

The months are pre-loaded into an array in the **BEGIN** section of the script. The second assignment statement needs to include a separating space at the beginning, or we would get a month called 'JulAug'. Also, in the second assignment statement is another feature of awk, concatenation by including a space between the variable name and the string to be joined to it.

```
jane@daisy > ~ $ echo 22 5 2010 | ./awkmonth.awk
22 May 2010
```

**length()** a nice, easy one.

**length( string)**

It simply returns the length of the supplied string or, if no string is specified, the length of the current line **\$0**.

**substr()**

**substr( string, start-position, max\_length )**

This function returns the sub-string that begins at **start\_position**, and extends for **max\_length**

characters or to the end of the string. If **max\_length** is omitted, the default is the end of the string.

The function **returns** the sub-string found. It is not used to change a part of a string. Use **sub()** for that.

These functions can also be used on the command line, although they are more usually found in scripts. To demonstrate command line usage, we can send the output from the **uname -r** command (which shows the release of the currently used kernel) through a pipe to **awk**, and apply the **substr()** function to find only a part of the output and print that part to screen.

```
jane@daisy > ~ $ uname -r
2.6.26.8.tex3
jane@daisy > ~ $ uname -r | awk '{print substr($1,5,4)}'
26.8
```

When you need to find the position of a sub-string within a string **awk** provides the **index()** function.

**index( string, substring )** The return value is the start position of the sub-string, or 0 if it is not found.

```
#!/bin/awk -f
{ place = index( $0, "AMD")
  print substr($0, place)
}
```

```
jane@daisy > ~ $ uname -a
Linux daisy 2.6.26.8.tex3 #1 SMP Mon Jan 12 04:33:38 CST 2009 i686 AMD Athlon(tm)
) 64 X2 Dual Core Processor 4800+ GNU/Linux
jane@daisy > ~ $ uname -a | ./awkindex.awk
AMD Athlon(tm) 64 X2 Dual Core Processor 4800+ GNU/Linux
```

We find the start of the processor description, and then use the return value to cut out a sub-string from there to the end of the line. In this way, we don't have to know how many words will be in the description.

A similar function is **match()**.

**match( string, regular\_expression )**

Instead of searching for a simple substring, the **match()** function uses the much more powerful regular expression pattern matching. The return value is, like **index()**, the starting position of the match, but this function also sets the value of two of **awk's** variables: **RSTART** & **RLENGTH**

Here's a file we created right at the beginning of this course:

```
jane@daisy > ~ $ cat newfile
This file was created in terminal 2 on
Sun Dec 6 09:40:50 GMT 2009
```

If we look for the beginning of the time string in the second line:

```
#!/bin/awk -f
{
  place = match( $0, /[0-9][0-9]:/)
  if ( place ) print place, RSTART, RLENGTH
}
```

We get this result, only the second line contained a match.

```
jane@daisy > ~ $ cat newfile | ./awkmatch.awk
12 12 3
```

Something that we often need to do is to convert the case of characters or strings from upper case to lower case, or from lower to upper.

**Awk** has a pair of functions that automate this process. They are called, not surprisingly **toupper()** and **tolower()**.

They each take a single string as an argument and return a **copy** of that string, with **all** of its characters converted to the respective case.

```
jane@daisy > ~ $ uname -o
GNU/Linux
jane@daisy > ~ $ uname -o | awk '{print tolower($0)}'
gnu/linux
jane@daisy > ~ $ uname -o | awk '{print toupper($0)}'
GNU/LINUX
```

What could be easier?

While we are dealing with text, I should mention the **sprintf()** function.

This function works just like the **printf()** function we used in **bash** shell scripting, except that this one doesn't print out the text. It returns a formatted copy of the text. This is extremely useful and can be used to create nicely formatted text files, where the fields of a record may be of indeterminate size.

You probably noticed that the output from the **owners.sh** script we used to demonstrate passing variables in a shell script was ragged and untidy. If we use the **printf** statement, instead of the simpler **print** command, we can specify exactly how we want the report to look.

```
#!/bin/bash
owner=3
group=4
filename=9

ls -l /home/$USER/Documents | awk '
BEGIN {
printf("%-10s%-15s%s", "Owner", "Group", "File Name\n\n") }
{ if ( NF ~ 9 ) {
printf("%-10s%-15s%s\n", '$owner', '$group', '$filename')}
}
END {
print "\nAll Done"
}
```

```
jane@daisy > ~ $ ./owners2.sh
Owner      Group      File Name

jane       jane       addresses
jane       musicclub  club_letter_1
jane       musicclub  club_letter_2
jane       jane       contacts
jane       jane       jan_report

All Done
```

The formatting rules are the same, and the fields to be output can be given a fixed width or, in the case of numerical fields, a pre-determined format or number of decimal places. Leading and trailing zero suppression is supported, as is padding of shorter fields with spaces or zeroes, as appropriate. Actually, all variables in awk are stored as strings, but variables containing numeric values can be used in arithmetic functions and calculations.

A nice feature of awk is that arrays are associative. What this means is that an array is a group of pairs. An index and a value. The index doesn't have to be an integer, as in most programming languages. It can be a string. The index is associated with the value. The order then is irrelevant, although you can

use numbers as the index to an element of an array. Its numerical value has no meaning in awk, only the fact that it is associated with a particular value is of interest. This makes arrays in awk particularly flexible and efficient. In most programming languages, arrays must be declared as to the type of values that will be stored, and the maximum number of elements that will be used. A block of memory is then allocated for that array, and size and type cannot be changed. awk, however, doesn't care about any of that. The indices may be anything that you wish. The stored values may be any mix of things that you wish, and you may add as many elements as you wish.

Associative arrays are particularly useful in look up tables. If you had a text file named phonetic with contents like this

```
a Alpha
b Bravo
c Charlie
:
:
y Yankee
z Zulu
```

Then we could read it into an associative array and use the array to convert characters to their phonetic equivalents.

```
#!/bin/awk -f
{codes[$1]=$2}
END{print codes["c"] " " codes["a"] " " codes["t"]}
```

```
jane@daisy > ~ $ ./phonetic.awk phonetics
Charlie Alpha Tango
```

If you happen to run out of steam with awks built in functions, or you find yourself repeating code, there is nothing to stop you writing your own functions.

Functions must be declared before any code that uses them, pretty obvious really except that they must be declared before the **code block** that calls them. This means that the function code should usually be written outside of and before the main loop.

The syntax for a function declaration is **function\_name ( parameters ) {actions}**

The keyword **function** is mandatory.

**function\_name** may be almost anything you like that is not a reserved word, a variable name or a pattern of characters that could be wrongly interpreted by awk. It should also begin with a letter or underscore character.

**parameters** are a comma separated list of variables that get passed to the function by the calling code. The names of the parameters are used by the function, and do not have to be the same as the name of the argument being passed. Only the value is passed to the function. Mostly though, it is less confusing if the names are kept the same.

Any **actions** inside the braces are what the function does with the passed parameters, and if a return statement is included, then that value will be returned to the calling code.

If a script called a function name **myfunction** with the command **result = myfunction( string)**, then

**return newstring** in the function code would return the value that the variable **newstring** holds in the function to the variable **result**.

If we wanted to make more use of our phonetics script by passing it any phonetics look up list and an arbitrary string to translate, we could write a function to do the translation.

```
#!/bin/awk -f

function translate ( c , codes){
split(c, letters, "")
l=length(c)
for ( i = 1; i <= l; ++i){
print codes[letters[i]]}
}# End of function declaration

{#Main loop
codes[$1]=$2
}

END{#finally call the function
translate( c , codes )
}
```

The function appears before the main loop and has two parameters passed to it, *c* is the string to translate and *codes* is the array of lowercase letters and their associated phonetic codes. The string is split into single characters by using an empty string as the field separator, and then stored in an array named *letters*. The length of the string is required to limit the loop, which loops round from one to the number of characters in the string printing the code that corresponds to the current letter.

In the main loop, the input data file named *phonetics* is read into the *codes* array.

In the END section the function is called and passed the string *c*, which is passed on the command line, and the *codes* array.

Here is the output from a sample run.

```
jane@daisy > ~ $ ./phonetic2.awk phonetics c=pclinuxos
Papa
Charlie
Lima
India
November
Uniform
Xray
Oscar
Sierra
```

Passing the name of the data file on the command line is useful if there are several data sets that you wish to switch between, but if there is only one, we can get the script to read it in by using awks **getline** function.

```
#!/bin/awk -f
BEGIN{ while (getline < "phonetics")
codes [$1]=$2}

function translate( c , codes){
split(c, letters, "")
l=length(c)
for ( i = 1; i <= l; ++i){
if (letters[i] ~ /[a-z]/).
    print codes[letters[i]]
else.
    print letters[i]
}
}

{
c=tolower($0)
translate(c, codes )
}
```

In the **BEGIN** section, the data file is read in using a while loop to repeat the process until we get an empty line. Each line is stuffed into the array **codes**. The string to translate is converted to lowercase at the start of the main loop, and in the body of the function, a check is made with a regular expression to see if the letter is in the range a-z, in which case it gets converted. If it is not in that range, then it is output as is, this takes care of spaces, numbers and punctuation. The strings to be converted may be piped in to the script, or can be typed interactively on the command line as below.

```
jane@daisy > ~ $ ./phonetic3.awk
Radically Simple
Romeo
Alpha
Delta
India
Charlie
Alpha
Lima
Lima
Yankee

Sierra
India
Mike
Papa
Lima
Echo
```



# Command Line Interface Intro: Part 11

by Peter Kelly (critter)

## A handful of the more useful commands

So far in this series, we have looked at most of the important aspects of working on the command line. This time I want to spend a little time looking at some of the more useful utilities available in a Linux distribution.

This is not meant to be an exhaustive list. There are far too many such commands available for that, but is a look at those utilities that you should find in almost any distribution, either installed by default or available from their repositories.

These are commands that you may not use every day, but you will find invaluable when you do need them. They are commands that I have not already covered, or that are deserving of a little more explanation. I am not going to cover every possible feature of the commands. The manuals are there for that, but I hope to cover them in enough detail to give a good understanding of what the command is capable of.

### **apropos {string}**

Ever been stuck at the command line, knowing what you want to do, but can't for the life of you remember the name of the command? Of course, we all have, and this is where this command saves the day. It searches a database of manual page descriptions for things that match the string that follows:

```
jane@daisy > ~ $ apropos calc
bc                (1) - An arbitrary precision calculator
calcomp           (4) - Calcomp input driver
cvt               (1) - calculate VESA CVT
dc                (1) - an arbitrary precision calculator
Digest            (3pm) - Modules that calculate digests
Digest::file      (3pm) - Calculate digests from files
gtf               (1) - calculate VESA GTF
ipcalc            (1) - perform simple math
Math::BigInt::Calc (3pm) - Pure Perl module
Math::BigInt::CalcEmu (3pm) - Emulate low-level math
Math::BigInt::FastCalc (3pm) - Math::BigInt::Calc
```

### **at {options}{time}{date}**

This enables you to have commands executed **at** a particular time. Where **cron** is used for running repetitive tasks at a particular time or frequency, the **at** command is useful for one-off tasks. To use it, the **atd** service must be running. This can be determined by issuing, as root, the command **/sbin/service atd status**

It may be started, if necessary, with the command **/sbin/service atd start**

You start the command by typing **at** followed, by a time, and optionally, a date. If the **-f** option is specified with a file name, then the list of commands to be executed are read from that file. Otherwise, you are prompted to enter them at the terminal. Entry is terminated with **Ctrl-d**.

The way time is specified is rather unusual. It may be entered in the format **15:30** for a 24 hour clock, or **3:30pm** for a 12 hour clock. You may also specify time as **midnight**, **noon**, **teatime** (4:00pm) or **now + 2 hours**, and you can add **today** or **tomorrow**. Date

takes the format **March 15** or **Mar 15**. The year is 4 digits, e.g. **2010**. You can simply specify a day, **Sat** or **Saturday**. An increment can be added to any part of the time date string.

When commands are executed any output or error messages are mailed to the user, but you must have the sendmail service installed and configured to make use of this. Instead, most users will want to redirect the output to a file.

Each set of commands is given a unique job number, and pending jobs can be listed with the command **atq**. Jobs can be removed from the queue with the command **atrm job-number**.

### **basename {name}{suffix}**

This command is used to remove all leading directory names and, optionally, the suffix from a fully qualified file name. So, **basename /home/jane/scripts/myscript1.sh** would return **myscript1.sh**, and **basename /home/jane/scripts/myscript1.sh .sh** would return **myscript1**. You will find this command invaluable in scripts.

### **bc**

**bc** is a command line calculator. That much is basically true, but it is also a gross understatement. It is a complete, compilable language, similar to **C**, which is capable of unlimited precision arithmetic. It has an extensive library of mathematical functions,

which you can include with the **-l** option, and you may define your own functions. Most people however, will use it interactively on the command line, or use it in a script to do a quick calculation or conversion.

To use **bc** on the command line, you may simply type **bc** and you will be greeted with a banner declaring version and licensing information, (this can be suppressed by using the **-q** option ), and a prompt.

```
jane@daisy > ~ $ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
```

You can then simply type in expressions, such as **2+3** or **3.1416\*6**, and the result will be echoed to the screen (stdout). To end the session, type **quit** or **ctrl-d**.

If you don't use the **-l** option, which will use long floating point numbers, then integer arithmetic is used. In the **bc** language, the scale of precision is zero, which means that zero decimal places are displayed. This can be changed by setting the variable **scale**. Using the **-l** option sets this value to 20 by default, which displays 20 decimal places. Note that if you use a value in an expression with more decimal places than the current value of scale, then that number of decimal places is used to display the result.

If you intend using **bc** then, as you don't need to know the version stuff every time, and bash can handle integers just fine, then you may want to put an alias in your **.bashrc** file.

```
alias bc="bc -lq"
```

The **bc** command is often used in pipelines and redirection, as in the following examples:

```
echo 3.1416*2.5 | bc -l
```

```
bc <<< 2^32
```

Or, the expressions can be read from a text file:

```
cat > sums
```

```
2+3
```

```
s(1.34) #sine 1.34 radians
```

```
scale=5
```

```
7/3
```

```
quit
```

Comments are allowed, and are preceded by a hash **#**. Without the quit statement, bc would wait for more input from **stdin**.

```
jane@daisy > ~ $ bc -lq sums
5
.97348454169531937478
2.33333
```

The **bc** command can work in any base from 2 to 16, but defaults to base 10.

To change this use the variables **ibase** and **obase** for the input and output bases. This is extremely useful if you need to convert between binary, hexadecimal, octal and decimal. The following bash script converts decimal to hexadecimal:

```
#!/bin/bash
#d2h.sh convert decimal to hexadecimal

if [ $# != 1 ]; then
    echo "Usage: d2h decimal-value"
    exit 1
fi
echo "ibase=10; obase=16; $1" | bc
```

```
jane@daisy > ~ $ ./d2h.sh 123456789
75BCD15
```

Should you want to edit this script to reverse the operation, hex to decimal, be aware that **obase** must be in **ibase** format:

```
ibase=16; obase=A
```

And hexadecimal letters must be in uppercase, since lowercase characters are reserved for bc variables. You could pipe the value through the **awk toupper** function, or use the **tr** command before passing it to **bc** to ensure that this is so.

```
cksum {files}
```

If you are transferring or compressing files, you may find this command useful. It calculates a cyclic

redundancy (CRC) value for all files specified on the command line. This value can later be used to check the integrity of the file.

```
jane@daisy > mydir1 $ cksum m*
2199948655 281389 musicfiles
1538883127 960 myfile1
```

### clear

Clears the terminal screen, nothing else. Simple, but essential.

### comm {option}{file\_a}{file\_b}

When you need to know how similar two files are, use this command. The output is in three columns:

```
jane@daisy > ~ $ comm contacts contacts2
          AMY
          BOB
    GEORGE
    GLENN
    GUSTAV
    JOHN
JURGEN
GLENN
JOHN
          SIMON
HUGH
BILL
          LEN
          BILL
          SALLY
          FRED

jane@daisy > ~ $ comm -12 contacts contacts2
AMY
BOB
SIMON
SALLY
FRED
```

lines only in **file\_a**, lines only in **file\_b** and lines common to both files. There is an option to suppress certain columns by listing their numbers with no spaces. **-12** would suppress columns 1 and 2, listing out only lines common to both files.

### cut {options}{files}

You can use this to chop out fields of data from files specified, or from **stdin**. The options allow you to specify a list of bytes **-b**, columns **-c** or fields **-f** to cut. If **-f** is specified, the default delimiter is the tab character, but this can be changed with the **-d** option. **-s** with **-f** suppresses lines with no delimiters.

If the file contains multi-byte characters, you have the **-n** option to avoid splitting them. The list of fields can be comma separated values, or ranges **2-5**.

```
jane@daisy > ~ $ lsusb | cut -d" " -f1-4,7-20
Bus 001 Device 001: Linux Foundation 2.0 root hub
Bus 001 Device 003: Standard Microsystems Corp.
Bus 001 Device 005: Realtek Semiconductor Corp. Card Reader
Bus 001 Device 006: Canon, Inc. CanoScan N670U/N676U/LiDE 20
Bus 001 Device 007: Hewlett-Packard DeskJet 930c
Bus 002 Device 001: Linux Foundation 1.1 root hub
Bus 002 Device 002: Logitech, Inc.
Bus 003 Device 001: Linux Foundation 1.1 root hub
Bus 003 Device 002: Novatek Microelectronics Corp.
Bus 004 Device 001: Linux Foundation 1.1 root hub
Bus 005 Device 001: Linux Foundation 1.1 root hub
Bus 006 Device 001: Linux Foundation 1.1 root hub
```

Here, we have asked for a list of usb devices available on the system. The delimiter is a space character, and we are not interested in fields 5 & 6. As the delimiter is a space and the name contains spaces the final range, 7-20, has been made wide enough to catch all words.

### df {options}{device\_name}

This is a handy little utility to report disk free space. If name is omitted, then all **mounted** file systems are reported. Name can be a device name, such as **/dev/sdb1**, or a mount point, such as **/home**. If a directory name is used, then the report is of the entire file system on which that directory is mounted.

Options include:

- a shows all file systems including 'dummy' files systems such as /proc
- i to show inode usage rather than byte usage.
- h reports in human readable form e.g. **15G**
- t only include type file systems **-t ext4**
- T include the file system type in the report
- x exclude listed file system types.

If you are using PCLinuxOS you will find that you have an alias to the command:

```
alias df='df -h -x supermount
```

This makes the report human readable, and doesn't probe supermount file systems. Supermount is a fake file system used to handle removable media, such as CDs and floppy disks.

It is important to monitor drive usage as a full drive can cause strange problems, particularly if an application or script has not been crafted in such a manner as to trap this.

**dirname {name}**

The opposite of **basename**, this strips away the file name in its entirety, by searching for the last */*.

**dirname /home/jane/scripts/myscript1.sh** would return **/home/jane/scripts**.

If there are no leading directories, as in **myscript1.sh**, then **.** (a period specifying current directory) is returned.

**du**

Reports the disk usage or the space used by a directory, the default is the current directory.

This is one of the 'core utilities.' You should find it on any Linux system, as it is considered essential, although most people will use few of its twenty or so command line options. If you are using PCLinuxOS, then you will have an alias to **du** that adds the **-h** option automatically, to make the output 'human readable'. Other options that are often used include:

- c** Print a grand total at the end of the output
- s** Summarize only. Outputs just a total
- x** Limit the count to files on one file system only

**--max-depth=n** Restrict the report to directories **n** levels deep. note that this is a 'long-option' and is preceded by two hyphens not one.

**file {file name}**

When you want to know what type of data is in a particular file, use this command.

```
jane@daisy > pixmaps $ file svg-viewer.svg
svg-viewer.svg: SVG Scalable Vector Graphics image
```

**fmt**

If you have a plain text file that you need to fit in a restricted screen or paper space, then this command does a nice job of it. It endeavors to retain as much of the original files formatting as possible. Spacing and indentation are kept intact wherever possible, and line breaks are done at the ends of sentences. The most useful option here is **-w** to set the width.

**fmt -w 36 a-text-file** will output the file in a single 36 character wide formatted column.

**free**

When your system seems particularly slow or sluggish, you should use this command to check your memory usage. The options **-b -k -m** & **-g** display the output in bytes, kilobytes megabytes or gigabytes respectively. The default is kilobytes.

```
jane@daisy > ~ $ free -m
              total        used         free   shared  buffers   cached
Mem:          1011         258          753         0         48        115
-/+ buffers/cache:
Swap:         520           0          520
```

Normally, you have two types of memory: physical, and virtual or swap space. When an application is started, the kernel allocates it some memory to work in. Additionally, the application may request more

memory as demand from the user increases. Data can be stored in a cache, in readiness to be operated on by the application, and temporary data and results are stored in buffers. For example, when a CD burning application is writing data, then that data is read from the hard drive much more quickly than it can be burned to the CD, and so the data is stored in a buffer where it can be accessed as required.

When the kernel has exhausted its supply of physical memory, it begins to 'swap out' some of the allocated memory contents to virtual memory to relieve the situation. Swap is much slower than physical memory, and so the result is that the machine slows down.

If you experience high swap memory usage, then something needs to be done.

You can shut down some applications, which will help, or you may have a rogue process hogging resources, remember orphans and zombies from when we discussed processes? If you need to track down which processes are using the most memory, use a tool like the command **top**. It may even be that you need to fit more memory to the machine if you regularly get this situation.

In the screen dump above, the top line is the physical memory, the bottom line swap and the middle line is the amount of memory used or free, without that which is promised to buffers and cache storage.



**fuser {file or file-system}**

Have you ever tried to **umount** a usb or other type of external drive, only to be told that it is busy with some process or other? Although you thought that you closed down all of the files on the device, the system thinks otherwise, and it refuses to let you safely remove it.

This command is what you need at times like this. It lists the process ID (PID) of all active files on the specified path. Now, path here can be a path name, a mounted directory, a block device such as /dev/sdb1 or even a remote system such as an ftp server.

For our usb drive example, we need to specify the **-m** option to tell the command that what follows is a mount point or block device. The PID is not very informative, so we specify the **-v** option to get verbose output that tells us what the file(s) actually is (are). As we want to remove the device, we need to kill the process(es), so we add the **-k** option. This however, is potentially dangerous, as we have forgotten what the file is. How can we be certain that we have finished with it and saved any changes? To be safe, rather than sorry, we can add the **-i** option that interactively asks for confirmation before killing each process.

**fuser -mvki /media/usb/** This will list any files open on the device and prompt you to delete them or not.

Once we have closed all the open files, we can resume the **umount** command.

**gzip & gunzip**

There are many file compression tools around, and **gzip** is one of the best of them. This one is so tightly integrated into so many Unix/Linux practices and processes that knowing how to use it is a must if you are going to spend any time on the command line. Using it is easy anyways, so why not? Compressed files save on storage space, not such an issue these days, and transfer much more quickly over slow transmission lines and networks.

The easiest way to use this is simply **gzip file name**. Unusually, this changes the original file and does not create a new one. The file is compressed and **.gz** appended to its name. Among the options you can pass to it are:

**-d** decompress -- this is exactly the same as gunzip

**-s suffix** to change the default **gz** to something that you prefer

**-n** where n is not the letter n but a number from 1 to 9 that will determine the amount of compression to be applied. 9 is the maximum compression. You may also use **-fast** or **-best** in place of **-1** and **-9**. The default here is 6, which is fine for most uses.

**-r** if you pass a directory to the command, then this will recursively work on the files it contains.

Multiple files may be passed on the command line as arguments.

**head**

**head** is useful for displaying just the first few lines of a file. By default, 10 lines are shown, but this can be changed by passing the number of lines required as an option, e.g. **head -15 logfile**. See also **tail**.

**kill**

When you find that a process is misbehaving and you want to end it, then use this command. Obviously, you must be the owner of the process or have superuser privileges to do this.

To use it, you send it a signal as an option, and provide the PID of the process(es) to be dealt with. If no signal is specified, then the default signal **TERM** is used. The signal can be specified as a number or as a name. The **TERM** signal is number **15** and is named **SIGTERM**. Some processes can trap the signal being sent to them as a sort of survival tactic, and stubbornly refuse to die. In these cases, use signal number **9**, **SIGKILL** this cannot be caught, but it is rather drastic and a more graceful closure is to be preferred.

**Kill -9 1729** will stop process number 1729 in its tracks.

To get a list of all signal numbers and their names, use the **-l** option.

**less**

To display text files on a terminal screen in Unix, a command named **more** was created (pressing the space bar showed more text). It was quite limited and fairly clumsy to use, with most of its commands being based on the **vi** editor.

Soon, a much more capable alternative came on the scene and, perversely, was named **less**. This is one of those commands that has far too many options for me; I like to keep things as simple as possible. If there is something that you want to do with a text file, then **less** is probably capable of doing it. Usually, I just want to look at the text and scroll back and forth. For this, it is excellent.

Type **less textfile** and the beginning of the file will be displayed on screen. Press **f** or the space bar to go forward, **b** to go backward or use the arrow keys to navigate around. Press **q** to quit. What could be simpler?

To search within the file, type a forward slash, followed by the word or pattern that you want to find and press return. Use **?** in place of the slash to search backwards.

For me, this suffices. If I want to do more, I use an editor.

**namei**

Trace a path name to its conclusion.

This rather unusual command is actually quite useful. On my system, I have the 'all bells and whistles' version of **vi** installed – **vim**.

When I type **/bin/vi** on the command line I am given an editor that happens to be **vim**. If I use the **which** command to find out what will be executed by typing **vi**, I get this:

```
which vi
/usr/bin/vi
```

So what's going on here?

**namei** will follow the path name, through any links, and display a character to describe the type of each file found. These characters are:

-	regular file
<b>b</b>	block device
<b>c</b>	character device
<b>d</b>	directory
<b>f</b> :	the path name currently being worked on
<b>l</b>	link
<b>s</b>	socket
<b>?</b>	an error

Typing **namei /bin/vi** gives the following output:

```
jane@daisy > ~ $ namei /bin/vi
f: /bin/vi
d /
d bin
l vi -> /etc/alternatives/vi
d /
d etc
d alternatives
l vi -> /usr/bin/vim-enhanced
d /
d usr
d bin
- vim-enhanced
```

This shows the links that the command **/bin/vi** has gone through, and that I am really executing **/usr/bin/vim-enhanced**.

**ps**

Display information about processes running on the system.

This is how you can find out which processes are using system resources, who owns them and their PID's.

Armed with this information, you can deal with any processes that step out of line or that are just taking up space. The output can be quite long, and you will find that this command is often used with **grep** to filter the information, and then piped to **less** to give time to read it.

The command comes with enough options to satisfy even the most enthusiastic systems administrator, but us mortals can mostly get the results with just four.

- a** list all processes
- u** include the username of the owner of the process
- x** include processes that are not associated with a terminal
- f** display a tree like structure to show parent/child process relationships

A leading hyphen is not required with these options and should not be used, as it can change the

meaning of some options. The output is a series of fields displayed in columns, and what is output is dependent upon which options are used.

The first line of output contains the column headers, which describe what follows below. The interesting ones for us are:

**USER** process owners user name  
**PID** process ID number  
**%CPU** amount of processor time used  
**%MEM** percentage of physical memory used by the process  
**VSZ** virtual memory size in KB  
**TTY** the terminal on which it is running, if any.

**STAT** process status which can be one of the following:

- D** in a deep, uninterruptable sleep
- S** sleeping or waiting
- R** running or in the queue to be run
- T** stopped
- X** dead, you should never see this as it should have been removed from the process queue
- Z** zombie - you should remove these

Additionally, you may also see alongside these codes one or more of the following:

- N** nice, low priority
- <** high priority
- l** multi-threaded
- L** locked pages in memory

- s** session leader
- +** foreground process

**COMMAND** The name of the process

## reset

Occasionally, you may find that your terminal gets corrupted. This is quite rare these days, but it can still happen, and you get nothing but garbage on screen. Just type **reset**, even if you can't see what you are typing, and things should soon be back to normal.

## rm

This is one of the most basic file handling commands, and potentially one of the most destructive. It removes files, or rather it removes the directory entry for the file, although the data is still on the disk until over-written. You do not need write permissions on the file to remove it, only write permissions on the directory that contains it. When used with the **-r** option, it will remove files recursively for directories, which is obviously dangerous, and for that reason, most systems have an alias that reads **alias rm='rm -i'**. This makes the command interactively prompt for a y or n before removing the file.

## script textfile

After typing this command, everything you do at the terminal is copied to the file 'textfile' (or whatever you called it).

This can be helpful if you need to show somebody how to do something at the terminal, and it has the added advantage that once you have it right, you can edit out all your failed attempts.

## stat

Prints out information about a file gathered from the inode that holds the files metadata.

```
jane@daisy > ~ $ stat musicfiles
File: 'musicfiles'
Size: 5324          Blocks: 16          IO Block: 4096   regular file
Device: 386h/774d  Inode: 98559        Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 501/   jane)   Gid: ( 500/   jane)
Access: 2010-06-13 12:54:17.000000000 -0500
Modify: 2009-12-06 05:14:10.000000000 -0600
Change: 2010-06-06 10:53:15.000000000 -0500
```

If you ever see the error message **"can't stat file \*\*\*\*\*"**, it usually means that the file doesn't exist (in the directory that you instructed the command to look for it).

## tac

Where **cat** prints out files first line to last line, **tac** prints last line to first line, reversing the order. This behavior can be modified by changing the separator using option **-s**. The default is a newline.

**tail**

Tail is useful for displaying just the last few lines of a file. By default, 10 lines are shown, but this can be changed by passing the number of lines required as an option e.g. **tail -15 logfile**. See also **head**.

**tee**

Use this when you want the output from a command to go to more than one place. You might want to view the output of the command and to also save it to a file:

```
cat /etc/fstab | tee ~/myfstab
```

This would display the file on screen, and also write it to a file in my home directory.

Multiple destinations may be specified. The **-a** option appends the information to a file.

**uname**

You can display information about the machine and its operating system, and this information can help when troubleshooting.

```
jane@daisy > ~ $ uname -a
Linux daisy 2.6.26.8.tex3 #1 SMP Mon Jan 12 04:33:38 CST 2009 i686 AMD Athlon(tm)
) 64 X2 Dual Core Processor 4800+ GNU/Linux
```

The options are:

```
-a      all information
-m      system hardware
-n      network host name
-r      kernel release
-s      operating system
-p      processor type
-v      kernel build info.
```

**uniq**

This command finds duplicate lines in a sorted file. The lines must be next to one another, so the file must first be sorted.

This is usually done on the fly with the **sort** command.

Suppose that you had a file containing a list of objects, and you believed that some of them may be duplicated. Jane has a list of American states in no particular order.

To show only those states that are duplicated, you can use the **-d** option.

```
jane@daisy > ~ $ sort states | uniq -d
Arkansas
Kentucky
Minnesota
```

To find out how many times each duplicated entry appears, use the **-c** to add a count. This would be

useful to make an inventory if the list was a stock list.

```
jane@daisy > ~ $ sort states | uniq -cd
 3 Arkansas
 2 Kentucky
 2 Minnesota
```

To get the list that Jane wants, a sorted list with no duplicated entries, the output of the command with no options is sent to a new file.

```
jane@daisy > ~ $ sort states | uniq > states2
jane@daisy > ~ $ ls -l states*
-rw-r--r-- 1 jane jane 508 Jul 11 13:18 states
-rw-r--r-- 1 jane jane 472 Jul 11 13:22 states2
```

The file size shows that the duplicated entries have been removed.

**wc**

Use this command to count the number of characters, words or lines in a file, with the corresponding options **-c**, **-w** or **-l**.

```
jane@daisy > ~ $ wc -l /etc/passwd
32 /etc/passwd
jane@daisy > ~ $ wc -w /usr/share/dict/words
483523 /usr/share/dict/words
```

Of course, the input doesn't need to be a file. You can pipe the output from another command to count the number of results returned.



## whatis

Have you wondered, "what does that command do?" This command may help by printing a one line description from the man pages.

If nothing is found, then it politely replies "Nothing appropriate."

## whereis

This command will search for, and output, the full path to the executable file, the man pages and the source of any command.

The **-b**, **-m** and **-s** options will limit the search to the binary (executable), manual or source files only.

## who

Although originally intended for multi-user systems to find out who was logged on, this command has a few useful options on a stand-alone system.

**who -b** will tell you the last time the system was booted

**who -d** gives a list of dead processes

**who -r** displays the current run level

Adding **-H** will add a row of column headings to the output.

You may also type **who am i**, which seems pretty pointless, and there is also a stand alone command called **whoami**. These return different results depending upon your situation. Consider this:

```
[jane@daisy ~]$ who am i
jane pts/2 2010-07-16 17:06
[jane@daisy ~]$ whoami
jane
[jane@daisy ~]$ su
Password:
[root@daisy jane]# who am i
jane pts/2 2010-07-16 17:06
[root@daisy jane]# whoami
root
```

Here, both commands return your usual user name when you are operating as a normal user. However, when you switch users with the **su** command, the **who am i** command tells you who you really are logged in as, and the **whoami** command tells you who you are being seen as when issuing commands.

This is particularly useful in scripts to check the user integrity before issuing a potentially disastrous command.

## xargs

This is one that you really should have a good idea about. It allows you to pass as many arguments as you like to a command.

This can be difficult on a command line, but this command is a boon in scripts, and that is where it is often found.

With the **xargs** command, you can re-direct the output from a command as a series of arguments to another command.

```
find . -iname "c*.sh" | xargs lpr
```

This will print out the contents of all the files in the current directory that begin with **c** and end with **.sh** (shell scripts?) to your printer - even if there are thousands of them that couldn't possibly be listed on the command line.

In a shell script, you will rarely know how many results you will receive from a command, but this command will pass them all to your destination command sequence, and you may filter the stream to pass over only the results that you are interested in.

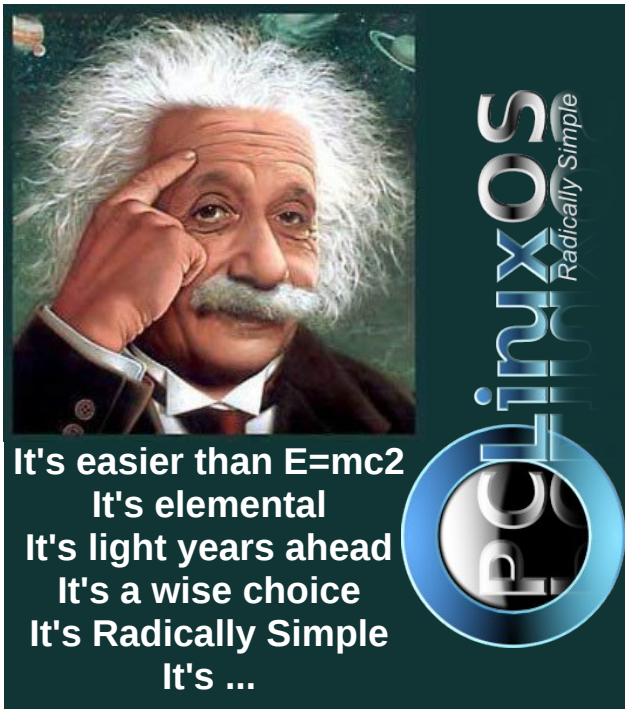
## And the rest?

Well yes, there are many, many more commands at your disposal when you are using the command line interface to a Linux or Unix system. The few that I have described above are the ones that I find most useful, and all of them have a great deal more functionality than I have described here.

Linux has almost all of the commands readily available to do whatever you wish, and you can

create your own, personally tailored commands to perform the functions that you cannot readily find available.

You can then offer them to others who may find them useful. This is the way that Linux/Unix developed, and sharing and developing is the basis of the open source community, which has provided us with a superb, free operating system.



It's easier than  $E=mc^2$   
 It's elemental  
 It's light years ahead  
 It's a wise choice  
 It's Radically Simple  
 It's ...

**PCLinuxOS**  
 Radically Simple

# Screenshot Showcase



*Posted by Leiche, June 27, 2010, running PCLinuxOS LXDE.*

# Command Line Interface Intro: Part 12

by Peter Kelly (critter)

## The vi editor

I'll be honest. I don't like **vi**. The newer **vim** (**vi** – improved) is well named, being an **improvement**, but none the less, still **vi**. I find that the commands are not intuitive and, unless you use it regularly, difficult to remember.

The main reason to learn **vi** is that you must do it if you are going to use the command line in any sort of a serious way. Also, it is sometimes the only editor available, but it invariably will be available. Some system commands, such as **cron**, rely on **vi**, and it will drop you straight into **vi** when editing **crontab**. The **sudo** command insists that you use a special version of **vi** named **visudo** to edit its configuration file **/etc/sudoers**, although it is not really necessary to do so. Many other system utilities base their commands on this editor.

Although I personally don't like **vi**, I have to admit that it is a very powerful editor. And, once you are familiar with **vi**, it can be a very fast way of editing text files. There is a lot of documentation available for **vi** and **vim**, if you want to learn how to use it as a professional. Here, I will show you the basics that can be learned in just a few minutes, and will enable you to do most of the editing that you need to do to get out of a sticky situation, when **vi(m)** is the only editor available.

You start the application by typing **vi**, followed by a file name. If the file doesn't exist, then it will be created when you save your changes. You may also open a file at a particular line number by typing a

plus sign (+), followed by a number, or at the first occurrence of a pattern of characters with a +, then a forward slash (/) and the pattern to be matched. This is useful when editing a script and trying to get a particular section working correctly. Try **vi +/\$USER /etc/passwd** to open that file at your entry in it. If working as an ordinary user, the file will open, but as you don't have write permissions, it will be in read-only mode. This fact will be displayed at the bottom of the screen.

The first thing that most new users fail to do is to get out of the application, as there is no easy "quit" or "exit" command. Let's get this out of the way right now. **vi** is bi-modal. This means that it has two different modes of operation: command mode and insert mode. When you open a file, you are placed in command mode, with the text of the file on screen, which you can move around in but not edit directly. To edit the file, you need to issue a command that will put you into insert mode. But you cannot exit the application from here, and that is mostly what confuses new users. To exit insert mode, you press the escape key. If you forget which mode you are in, or just feel lost, then press the escape key and you will always be put back to command mode.

Once in command mode, you can exit the program. To do this, type a colon, which will appear at the bottom left of the screen. **vi** then waits for you to type a command. The command to quit is **q**. If there have been no edits, then the application exits and returns you to the command line. If the text of the file has changed, you will get an error warning stating "no write since last change". Now you can do one of two things.

**:wq** the **w** writes out the changes then **q** quits  
**:q!** the exclamation point tells **vi** to discard the changes and then quit.

In summary, to exit the file, press escape, then type **:wq**.

Moving around in the file can be done with the cursor keys, but was traditionally done by using the **h j k l** keys to move left, down, up or right respectively (l to go right?). To move one full word forward press **w** and then **b** to move back a word, and **0** and **\$** to move to the beginning or end of a line. **Ctrl-f** and **Ctrl-b** moves forward or backward a screen at a time. **Ctrl-d** or **Ctrl-u** moves up or down half a screen at a time.

In command mode, you can use the following commands:

**c** change  
**d** delete  
**y** yank which means copy  
**p** put or place the yanked text at the cursor

When you issue the **c** or **d** commands, the text is removed from the screen and placed in a buffer known as the cut, or yank, buffer. The **y** command places a copy of the text into the buffer, leaving the screen unchanged. You can then re-position the cursor and press **p** to "put" the contents of the buffer at that position. The stored text can be re-used as many times as required, until it is replaced by another operation.



What you change, delete or yank are objects, including words, lines, sentences, paragraphs or sections. But for our simple editing needs, I will limit it to words and lines. You can also specify how many objects you want the command to operate on. To make the whole line the object, you repeat the command **cc dd** or **yy**.

Examples:

**5cw** change the next 5 words. this deletes the next five words and allows you type in some new ones.

**3dd** delete 3 lines starting with the current line.

**2yw** copy the next 2 words starting at the cursor, not necessarily at the beginning of the word.

**4yy** copy the current line and the next 3 lines

Insert mode is entered by typing one of the following commands: **a**, **A**, **c**, **C**, **i**, **I**, **o**, **O**, **R**, **s** or **S**. You will then see **- - INSERT - -** at the bottom left of the screen. These commands allow you to **append**, **change**, **insert**, **replace** or **substitute** text or **open** up a new line to type in some text. When users had to make do with a rather unforgiving dumb terminal, most of these options would have been welcomed. Today's desktop computer keyboard interface is rather more sophisticated and standardized.

With the movement keys outlined above, we can quickly move to the part of the text that we need to modify, and press **i** to enter insert mode. We can now begin typing new text. Press the insert key on

the keyboard to toggle overwrite mode. You will notice the **"- - INSERT - -"** at the bottom changes to **"- - REPLACE - -"**, or use the delete key to remove text.

If you are using the more advanced **vim**, which I would recommend if you have a choice (and PCLinuxOS users do have this choice), you can activate a visual highlighting mode, which can be character-wise, line-wise or block-wise. Press the escape key to get into command mode, and press **v**. You are now in character-wise visual mode, and text under the cursor is highlighted as you move around. Uppercase **V** puts you in line-wise mode and full lines only can be highlighted. **Ctrl-v** enters block-wise mode. Here a rectangle of text is highlighted as you move across and up or down. A simple experiment in each of the three modes will demonstrate this much more easily than I could describe the effects.

With the text highlighted, you can issue the **c**, **d** or **y** commands, with the **c** command automatically putting you in insert mode to type in the replacement text.

This brief introduction to **vi** will allow you to perform almost all of the editing that you will ever need to do on the command line. Obviously, if you learn some more of the available commands, then your editing will become even more efficient. But this is enough to get you out of trouble when things aren't going so well, or to enable you to edit files like crontab or sudoers.

## Midnight Commander

One of the most useful utilities for the command line user is Midnight Commander. For those of you who aren't familiar with it, I'll explain. Midnight Commander is a two panel file manager, very similar to **KDE's Kruzader**. The main difference is that it is entirely text based and used from a terminal. It provides a graphical interface to most file system management tasks, using elements from the **ncurses** and **S-lang** libraries to provide the text drawn graphics. The application is extremely customizable, and it is installed by default in most full variants of PCLinuxOS. It will also be found in most other Linux distributions. Mouse interaction is supported and works fine in a terminal emulator under a windowing system. But for use in a 'true' terminal, as you will get by typing Ctrl-Alt-F2, you will need to install the **gpm** mouse server from the repositories. Midnight Commander includes a text file viewer and an excellent editor, and can be used over remote connections. Midnight Commander will also let you look inside compressed files and rpm packages by simply pressing enter when the file is highlighted.

Midnight Commander (hereafter referred to as MC) can be started from the command line by simply typing **mc**. It is intuitive enough to be used immediately by even the newest Linux user. You may be wondering why I have not introduced such a wonderful time saving utility before now, and why you have had to jump through hoops in an unfriendly and often unforgiving environment to achieve even the simplest file system commands, such as copying and moving files. Quite simply, you have now seen the interior workings of the Linux system and are



more able to take full advantage of it, and to understand the many advanced features, which many users do not comprehend or miss completely.

By default you will start with a screen like this:

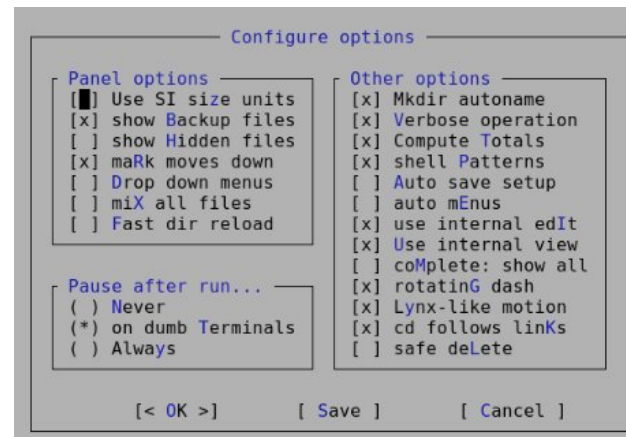


The top line is a drop down menu bar, accessible with the mouse or by pressing F9 and then the arrow keys. Directly below are the two panels that are split vertically by default, but can be changed to horizontal from the configuration menu. Then **left** and **right** on the top line will read **above** and **below**. At the bottom of each panel is a status bar, which displays some file information and file system usage (default). Next down is the command line. Anything entered from the keyboard that is not interpreted as a command to MC goes here, and enter sends it to the shell for processing. The bottom line is a set of buttons corresponding to the Function keys, but they are also mouse clickable. The panels are overlaid on the output screen, and can be toggled on or off by pressing **Ctrl-o** (that's letter o, not zero). You may want to do this to see, for example, the output of a command executed from the command line window.

Anything that you type in MC is examined to check whether it is a MC command. If it's not, it is passed to the shell to be dealt with. There are a lot of commands in MC and shortcuts to them are shown in the drop down menus like this:

- Ctrl-u** hold down control and press u - this one swaps the panels over
- Ctrl-x c** hold down control, press x, release both and type c - brings up the chmod dialog
- Meta-?** hold down the meta key, more usually known as the Alt key, and type ? bring up the find file dialog.

Basic configuration is done through a menu found under options on the top menu bar. Drop this menu down and press **Enter** on the configuration entry. You will get a dialog like this:



Use the arrow keys to move around, and press the space bar to add or remove an option, or hold down the Alt key and press the letter in blue. Most of the

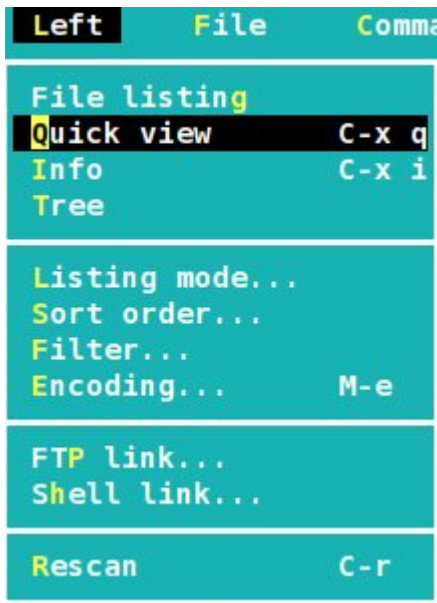
options can safely be left at their default settings. I prefer to not **show hidden files** unless necessary, as they are hidden for good reason. I also recommend checking **Lynx-like motion**. Lynx is a text only web browser which uses the arrow keys for navigation through links, and this option allows you to move up or down through highlighted directories by using the left and right arrow keys. The **shell Patterns** option, when checked, uses search patterns such as wild cards, as you would use in shell 'globbing.' Unchecked, it uses the full power of regular expressions, which makes it an extremely powerful tool. If you need more help on the other options, **F1** will bring up a fairly comprehensive help system.

For most operations, you will want to have the two panels showing the contents of two different directories, perhaps source and destination directories for copy and move operations. Switch between panels with the tab key, and select files by tagging them with **Ctrl-t** or the **insert** key. **F5** copies selected files from the active panel to the other panel by default, but pops up a dialog box to allow you to change this. **F6** moves them across, with the option to rename the file, and **F8** deletes them. While a file is highlighted (not tagged), pressing **F3** displays the contents where practical, and **F4** opens it in the editor, although you must have write permission to save any edits to the file.

If you find that you need to frequently drill down to a directory buried deep within the file system, you can add it to the hot-list dialog. type **Ctrl-l** and select add current (Alt-a). You will be prompted for a name for the entry. The full path-name will already be there if you want to use it. Want

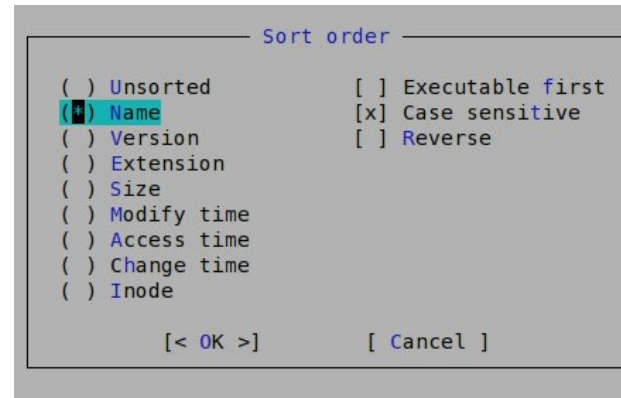
to go home? Type **cd**, (the letters go into the command line box as you type), then press **Enter** and the active panel will show your home directory.

The panels are not limited to displaying a directory listing. By dropping down the **left** or **right** menus (F9),



you have the option to change the display in that panel to the contents of the currently highlighted file in the other panel, or to display a heap of information about the file. It can also be set to display the file system in a tree like structure. If you keep the directory listing, the same menu will allow you to select the amount of detail shown similar to the **-a**, and **-l** options of the **ls** command. Or you can set up a custom display to show exactly what you need.

The listing can be sorted in any way you like as shown below.



You can set up a filter to show only files that match a pattern. The rescan option, **Ctrl-r**, refreshes the contents of the active panel if the contents have changed since the directory was entered.

The **ftp** and **shell** link options are one of the cleverest parts of MC. They allow you to display the contents of a directory on a remote system, and let you navigate around as though it was on your own hard drive.

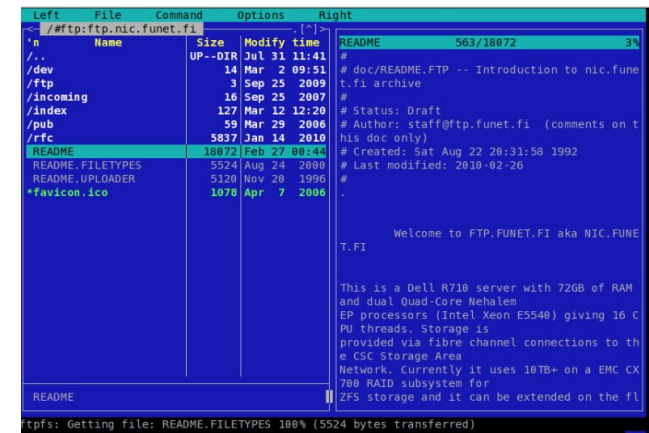
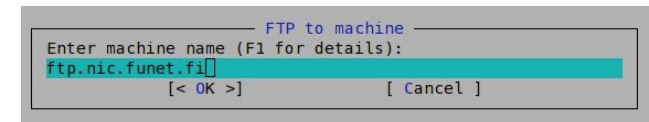
Try this:

Set the right panel to quick view.

Tab back to the left panel and drop down the **left** menu.

Select **ftp link ...**

Enter **ftp.nic.funet.fi** in the dialog box that appears.



On the right hand panel are displayed the contents of the highlighted file in the left panel. The files shown in the left panel are on a file server in Finland. Funet is the Finnish University and Research Network. You can freely browse any directories for which you have been granted access, and you may read or copy documents to your own home directory. It's a great research tool. Try opening the **pub/** directory. You can even add it to your directory hotlist (**ctrl-l**), and give it a nicer name for quick future access.

There are hundreds of free ftp sites that can be accessed in this manner. If you want to download software or a live cd image, I would recommend a dedicated ftp client application, such as **gFTP**, or the ftp capabilities of a decent web browser.

To use the **shell link...** option to connect to another machine on the local network, make sure that the **/etc/hosts** file contains a line with the IP address and host name of the remote computer to allow address translation. In the dialog that is presented when you select **shell link...** from the menu, type in something along the lines of **jane@daisy**. You may then be asked for Jane's password before being granted access to the machine as Jane.

When you enter a directory with lots of files and sub-directories, you can home in rapidly with the quick search function, **Ctrl-s**. Try navigating to the **/etc** directory and type **Ctrl-s fs**. You will be taken straight to the **/etc/fstab** file, where you can press **F3** to view the contents, or **F4** to edit it.

Under the file menu are options to perform most of the file handling commands that you would normally carry out on the command line. For example, to create a symbolic link in the right panel to a file in the left panel, simply select **Symlink**, and a dialog is shown with the defaults already filled in. Press **Enter** to accept or change the symlinks name to your preference.

At the bottom of the file menu are a group of commands to tag a group of files according to a pattern. As an example, if shell patterns are disabled, then a pattern such as **^\.bash.\*** in your home directory will tag all of your (hidden) bash related files ready to be copied to a backup directory.

Pressing the **F2** key brings up the user menu. What this shows depends upon the contents of the file **~/mc.menu**, and you can edit the file to your hearts content to customize the menu. Open up the default

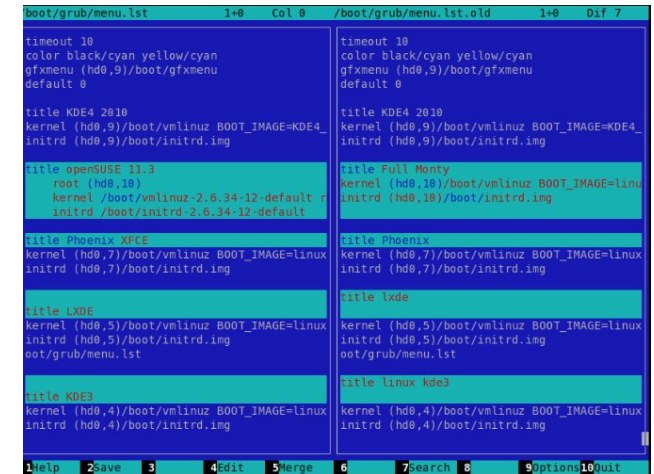
menu, and you will see just how complex you can make the menu commands. But simple commands are also acceptable.

There is a simple find files dialog accessible from the **command** drop down menu or by pressing **M-?** This is pretty easy to use, with a space for the start directory, which can be filled by selecting a directory from the **Tree** option and spaces for the search criteria, which may be either the file name or content. Both can use either shell expansions or regular expressions. When the list of files is displayed, you have the option to 'panelize' them, which means display them in the active panel for further processing. To get back the previous contents of the panel, use the refresh **Ctrl-r** command. This is an excellent place to practice your regular expression skills, with the **Again** option taking you back to the pattern for modification or refinement if the results are not what you expected.

If you need even more power, the **External Panelize** command will provide it. This is activated from the **command** drop down menu, or **Ctrl-x !**. This command allows you to execute an external command, and to put the results into the active panel. You can even save regularly used commands under a friendly name.

Also available from the **command** drop down menu is the **directory tree** command, which displays a dialog showing the file system in a tree like structure, and changes the function key definitions at the bottom of the screen. **F4** or **Rescan** refreshes the tree display. **F3** temporarily removes a directory from the display. This useful when there are lots and lots of sub-directories making navigation difficult. **F4**

toggles between **static** and **dynamic** tree navigation. Play around with it and you'll see the difference. Pressing the **enter** key on any directory closes the dialog and switches the active panel to that directory.



To compare the contents of two files that you have been editing, select one file in the left panel, and the other in the right. Then, from the **command** drop down menu, select **view diff files** and a new, two panel window will open, showing the contents of both files, with the differences highlighted

If you need to keep two directories synchronized, the **Compare directories**, or **Ctrl-x d**, is a boon. With one of each of the directories in its own panel, execute the command, and you will be prompted for the type of comparison to make, and the files that differ will be tagged on both sides. You can then simply copy the tagged files across with the **F5** key.



When you open a file in the editor or viewer, or compare two files, a new screen is shown. MC can have multiple screens open at any time, and you can switch between them as you wish, using these commands:

**Alt-}** forward one screen  
**Alt-{** back one screen  
**Alt-`** list the open screens (that's the back tick)

Unfortunately, you may only have one file listing screen open at a time.

If you forget to close an open screen and try to leave, MC will issue a warning.

If you think that I have done a comprehensive job of covering the features in MC, then you are not even close. I've only covered the features that I regularly use. Read the help files and you will find a lot more to play with.

## Playtime

When you are working in a text only terminal, either because you are trying to achieve a command line oriented goal, are simply locked out of your beloved X windowing system temporarily, or just because you want to, (sadly), you don't have to sit there in monastic silence.

There is a widely available tool known as **sox**, which is an incredibly powerful audio application with a bewildering array of options. It recognizes most

audio formats, can play back or record, add effects, split, combine and do just about anything that a reasonable person would wish to do with, or to, an audio file.

If you are so inclined, then please, be my guest. Read the manuals and produce your masterpiece. Personally, I would much rather use a graphical application, such as **audacity**, to perform such magic.

**sox** can be called in one of three incarnations:

**sox** the full version  
**play** the play back part and  
**rec** the recording function part.

Let's concentrate on **play**.

To simply have some music while you work, you can call **play -v {number} {song}**. The **-v** option controls the volume and **number** is a real number, the default being 1. Enter 0.5 for playback at half volume, 2 for double the default volume, or normal volume and so forth. Be warned that entering too high of a number may damage your hardware or ears.

Here's a little script that will allow you to have your favorite music playing while you work. It expects a folder containing compatible music files on the command line as a play list. This version looks for mp3 files and plays them back at half volume. Edit it to your own preferences.

Open a virtual terminal with **Ctrl-Alt-F2** and run the script. Use **Ctrl-Alt-F3** to open another terminal to do your work. Enjoy!

```
#!/bin/bash
# play-it-sam.sh
# play a folderfull of files consecutively
# while working on the command line
|
if [ $# != 1 ]
then
echo Usage: you need to pass me a directory
echo That contains some mp3 files
echo So that I can play them
echo
exit 0
fi

for song in $1/*.mp3
do `play -v 0.5 "$song"`
done
```

Call it with a command like **~/play-it-sam.sh /data/Music/The-Who/** (so, I like 60's pop, okay?).

The script could use a little more error checking, and has the potential to provide more functions, such as volume control. I'll leave that to you. If you make any significant improvements, and I am sure that you can, I would be interested in seeing them.

## Where to next?

If you want to stop right here, that is fine, What I have covered in this introduction is more than enough to lift you out of the 'newbie' class, and will almost certainly cover most of what the average user needs to make efficient use of the command line.



Should you wish to delve a little deeper, there are several ways to do this.

Almost all of the commands include some sort of documentation as part of the installation. As a minimum you can follow the command with `--help`, which will usually give you some idea of the usage, along with the available options of the command.

Most commands are also documented in the **man pages**, a special type of built in help system. The manual pages are not always installed by default in every distribution, but are almost certainly available from the software repositories, and are well worth installing. If a man page is not available from the repository for a given command, try <http://linuxmanpages.com/>. It's a great resource.

The manual pages are accessed by typing **man command**, where command is the name of the command you are interested in.

As a reference, they are invaluable. But they are not very beginner friendly, although the information that they contain is usually accurate. If you see a reference to a man page, it is often followed by a number. This is the section number. For convenience, the manual pages are organized into sections, but the number is optional. The sections include:

1. user commands
2. system calls
3. library functions
4. special files
5. file formats
6. games

7. conventions and miscellany
8. administration and privileged commands

When the **man** command fails to provide sufficient information, use the **info** command. The content is similar to the manual pages but often much more detailed. Unfortunately, the info command's user interface is terrible so remember this: type **q** to quit, and type **h** for help.

That should get you by.

There is, of course, lots of information available on the internet, but beware that some of it may be inaccurate. The most reliable source is **LDP** - the Linux documentation project (<http://www.tldp.org>). Here you will find a wealth of information on all things Linux, in a variety of formats and languages.

Another reliable online source is The Linux Gazette (<http://linuxgazette.net/>), with all back issues available in the archives. The Gazette has been around since 1995.

If you prefer a good, old-fashioned book, then you will be spoiled for choice, as there are literally thousands of them available. Which to choose can be a headache and a kind of lottery. My personal experience is that you can't go far wrong with the excellent O'reilly series of Linux reference titles. They are usually well written, reliably factual, and durable. I have a 10 year old copy of [Linux in a nutshell](#) that shows little sign of wear despite the years of rummaging through its 600 odd pages).

PCLinuxOS users are fortunate enough to have their own magazine, which is an excellent source of distribution-centric information. All [previous issues](#) are available for free download.

Last, but not least, there is the PCLinuxOS forum. If you can't find a solution to your problem, then ask there and surely one or more of the friendly resident experts will help. Even Texstar, the distribution's main man, is a regular contributor there.

The only other thing you need to become more proficient is practice. Only you can provide that. The more that you use the methods outlined in this introduction, the easier you will find them to use. Reading about a command is fine. But to understand it fully, you must use it regularly.

*Editor's Note: This, the 12th installment of the Command Line Interface Intro article series, is also the last. Critter, a.k.a. Pete Kelly, has provided us, the PCLinuxOS community, an outstanding tutorial on how to use the command line. If you have followed along, I'm sure that you have discovered just how powerful the Linux command line truly is, and how easy it can be.*

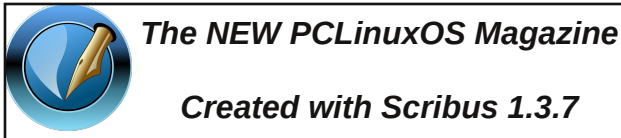
*Critter is not, however, "going away." He has agreed to stay on and write additional articles for The NEW PCLinuxOS Magazine. You will be seeing more from him in the coming months.*

*Meanwhile, we will be publishing a special edition of The NEW PCLinuxOS Magazine, containing all of Pete's excellent Command Line Interface Intro articles, in order from the first article, up to and including this final article.*

If you are (or get) serious about learning the Linux command line, then the special edition would serve as an excellent starting point, not to mention an excellent reference resource. Watch for it, coming soon.

Thank you, Pete, for all of your hard work in producing this outstanding tutorial series for The NEW PCLinuxOS Magazine.

Paul Arnote, PCLinuxOS Magazine Chief Editor



# Screenshot Showcase



Posted by Archie, May 6, 2010, running PCLinuxOS KDE 4.