Dr Brown's Administeria

Dr Chris Brown
The Doctor provides Linux training, authoring and consultancy. He finds his PhD in particle physics to be of no help in this work at all.

Esoteric system administration goodness from the impenetrable bowels of the server room.

Bit recycling

A recent visit to the Eden Project has strongly reinforced my need to recycle more and reduce my carbon footprint. I do a patchy job right now. It's true that I recycle the jokes I use when I'm running training courses, but that's because I only know four. I've turned down the speed on my treadmill. I've started shaving on alternate days *and* brushing the razor out into the compost. But I could do so much more.

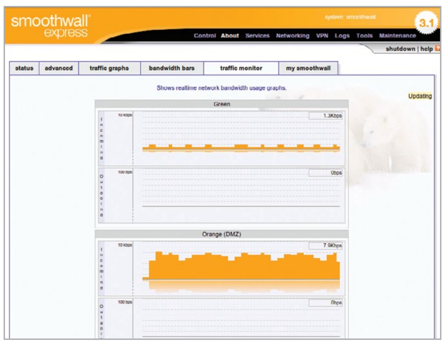
I'm especially worried about running out of bits – that is to say, binary digits – that form the very basis of our digital world. Think about it – when we're done with them, we just throw them all away. Nobody recycles them. At some point, when the very last one is used (or it could be the very last zero, it's hard to predict) every computer will come to a screaming halt. So I have decided to set up a bit recycling service. You'll be able to send all your scrap data to my servers which will sort it out into zeros and ones and store them on two very, very large disks. (If you're concerned about security I'll supply a little app that randomises the order of the bits before you send them). Then when you need some more, you just ask for however many zeros and however many ones you need, and I'll send them to you. At first it will be a free service, but the point is to create a dependency. Then when I've established a monopoly, and the bits really start to run out, and the chips are down (so to speak) I'll start charging. Eat your heart out, Mark Zuckerberg.
chris.linuxformat@gmail.com

Smoothwall

Firewalls Use open software to build closed systems – put that satin finish on your network.

Firewalls are an essential component of most corporate networks and you can spend good money (anywhere between £300 and £10,000) on dedicated hardware solutions. Alternatively, you might consider rescuing that ageing PC from the dumpster and installing *Smoothwall* on it. To quote the official website (**http://smoothwall.org**): "Smoothwall is a best-of-breed internet firewall/router, designed to run on commodity hardware and to provide an easy-to-use administration interface to those using it", and a new version (3.1) has just been released.

If you'd like to give it a try, for a modest (220MB) download you get an ISO image that installs in a couple of minutes. Hardware requirements are minimal. You don't need to know anything about Linux to install it, but you do need to understand the network topology surrounding your firewall or you'll come to a screaming halt, as I did, at the 'Network configuration type' screen.


smoothwall express
system: smoothwall
3.1
Control About Services Networking VPN Logs Tools Maintenance
shutdown | help
status advanced traffic graphs bandwidth bars traffic monitor my smoothwall
Shows realtime network bandwidth usage graphs.
Updating
Green
10 kbps
1.3Kbps
Incoming
100 bps
0bps
Outgoing
Orange (DMZ)
10 kbps
7.9Kbps
Incoming
100 bps
0bps
Outgoing

› **One of *Smoothwall*'s many web-based screens. Here, we see real-time network traffic on the green and orange networks.**

Coloured zones

Smoothwall divides your network up into colour-coded zones and asks you to allocate each network interface to a zone. For example, a typical scenario might use the green zone for the private internal network, the orange zone for the 'DMZ' network where your public servers sit, and the red zone for an internet-facing connections.

Once initial setup is complete, you configure *Smoothwall* via a web-based interface. There are screens to set up filter rules for incoming, outgoing, internal and external traffic. This approach means that you don't have to get down and dirty with *Iptables* rule sets but it doesn't entirely absolve you from the need to understand what's going on. In addition, *Smoothwall* provides proxies for web, instant messaging, SIP and POP3. You also get screens with pretty graphs and bar charts of network traffic and bandwidth, and screens for examining the log files.

Little and large

If you're really impatient to try out *Smoothwall*, there's a compressed *VMware* image available – just download, uncompress, and fire it up in *VMware* workstation or player. Be aware, though, that this is version 3.0 not 3.1. At the opposite end of the spectrum, the company's commercial arm at **www.smoothwall.net** offers a variety of products for network security, web content filtering, spam and malware filtering, and more.

Sed ain't dead, baby

Small languages The Good Doctor bats aside any suggestion that Sed is just too obscure to be useful and lays bare its genius for sysadmin scripts.

This month and next I want to take a look at a couple of 'small languages' that are popular in Linux – Sed and Awk. Both find widespread use in system administration scripts and it certainly helps if you are able to at least read and understand the code, even if you have little call to actually write any. This month I'll look at Sed.

Some would argue that the command set of Sed doesn't really qualify as a language; I will let you draw your own conclusion on that particular nugget (*But first read the box Is Sed a Language?*).

To begin at the beginning, Sed is a stream editor. It behaves as a classic filter – if you give it a file to operate on, it will stream input from that file. If you don't it will read its standard input, allowing it to sit on the downstream end of a pipeline to post-process output from some other command.

Whether reading from a file or from standard input (stdin), Sed reads the input stream a line at a time, performs a specified set of editing operations on the line, and writes the resulting line to standard out (stdout). Then it reads the next line and starts over. Unlike most interactive editors, which read the entire file into a buffer, sed works a line at a time, allowing it to operate efficiently on extremely large files.

Substituting with Sed

To get us started, here's a simple example of using Sed to perform a substitution – probably the commonest use for Sed. Suppose we have moved our users' home directories from **/home** to **/users** and need to modify all the home directory names in **/etc/passwd**. That is, we need to change lines of the form

```
chris:x:501:501::/home/chris:/bin/bash
```

to

```
chris:x:501:501::/users/chris:/bin/bash
```

This will do the job:

```
sed s/home/users/ /etc/passwd
```

Let's be clear what's happening here. Sed reads the password file line by line, performs the substitution on each line, and writes the result to stdout. It does *not* modify the original file. If you really do want to change the original file, it is tempting to try this:

```
sed s/home/users/ /etc/passwd > /etc/passwd
```

But that way lies doom and disaster. When the shell sees the output redirection it will truncate the output file down to zero length before Sed even gets to see it. Bye bye password file! That is generally true of filters – you cannot redirect their output back to the original file. Instead you would have to do something like this:

```
sed s/home/users/ /etc/passwd > /tmp/passwd
mv /tmp/passwd /etc/passwd
```

In fact, the GNU version of Sed also has an 'in place' option, **-i**, that does the job, so

```
sed -i s/home/users/ /etc/passwd
```

would also work, although in this particular case I would caution you not to mess with the password file unless you are sure your Sed command does what you think it does.

Our next example is even simpler. The **df** command generates a nice table of disk usage for each of your file systems, but it includes a heading line that can mess things up in downstream processing. We can delete that initial line:

```
df | sed 1d
```

Here, Sed is reading from standard input (the output piped from df). The **d** command means 'delete the line' and the **1** means 'just do it on line 1'. So the first line gets the chop, but all the others make it through unchanged. This is equivalent to doing a **tail -n +2**.

Let's go back to the **s** (substitute) command. Suppose you want to just get the user names from **/etc/passwd** – that is, the field up to the first colon. This is easy once you discover that the 'old pattern' part of the substitution can be a regular expression.

```
sed s/:.*// /etc/passwd
```

The example is a little deceptive. The 'old pattern' is the **regex ':.*'** which matches from the first colon through to the end of the line. (We're relying here on the 'greediness' of regex – it starts matching as soon as possible, and carries on matching for as long as possible). The 'new pattern' is empty, so whatever the regex matches is removed. Magic!

Let's take one more example of substitution. Suppose you wanted to change strings like "£25" to "25 GBP". This is trickier because the replacement text "GBP" needs to appear *after* the number. This command will do the job:

```
sed -r 's/£([0-9]*)/\1 GBP/g' prices
```

which will change a line like:

```
fees range from £20 to £40 typically
```

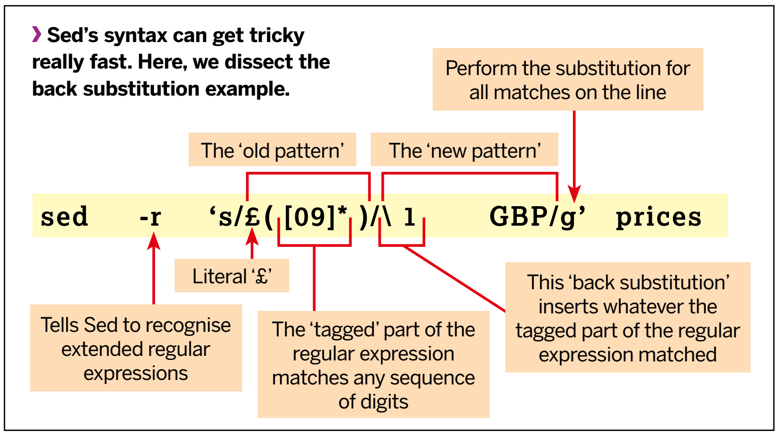
to:

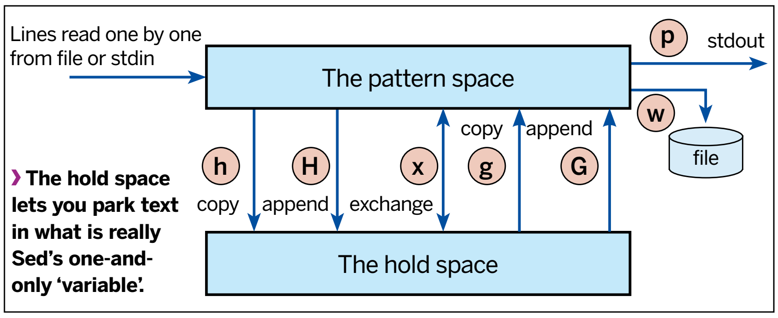
```
fees range from 20 GBP to 40 GBP typically
```

I've drawn a diagram to explain how this works (below). »

Is Sed a language?

You might argue that Sed's command set does not qualify as a programming language. In fact, a remarkable Sed script by Christophe Blaess shows that Sed is 'Turing Complete' which means that it can (in theory) approximately simulate any other general-purpose programming language. Julia Jomantaite has even written a *Tetris* game in Sed (see the links on Peteris Krumin's blog post **http://bit.ly/17DVI3o**). But I don't see anyone re-writing, say, the Linux kernel in Sed anytime soon.


› Sed's syntax can get tricky really fast. Here, we dissect the back substitution example.
Perform the substitution for all matches on the line
The 'old pattern'
The 'new pattern'
sed -r 's/£([09]*)/\1 GBP/g' prices
Literal '£'
Tells Sed to recognise extended regular expressions
The 'tagged' part of the regular expression matches any sequence of digits
This 'back substitution' inserts whatever the tagged part of the regular expression matched



› **The hold space lets you park text in what is really Sed's one-and-only 'variable'.**

Think that's tricksy? Well, maybe, but many of the Sed commands you find in administration scripts use far more spectacular regexes. Here's an example, taken from the file ***/etc/init/rc-sysinit.conf*** on Ubuntu:

```
sed -nre 's/^[^#][^:]*:([0-6sS]):initdefault:.*/DEFAULT_
RUNLEVEL="\1";/p' /etc/inittab
```

what this command does is extract the default run level from the **inittab** file.

Although it's conventional to use a forward slash to separate the pieces of a substitution command, it gets ugly if the old or new patterns themselves contain forward slashes. Suppose we want to change '***/home/chris/bin:***' to '***/opt/bin***'. We'd have to escape all those forward slashes so it would look like this:

```
sed 's/\/home\/chris\/bin\/\/opt\/bin/' foo.txt
```

Using a different separator (':' in this case) makes it a little easier:

```
sed 's:/home/chris/bin:/opt/bin:' foo.txt
```

Line selection

You can select single lines, or ranges of lines, that you want an editing command to operate on. The **1d** command we saw earlier is an example of this, selecting just line 1. Or we could select a range like this: **1,10d** to delete the first 10 lines, or **5,$d** to delete from line 5 to the end ($ is a shorthand for the last line of the file). We can also select lines based on a regular expression match, so

```
sed '/^#/d' /etc/fstab
```

will delete lines that begin with '#' (typically comment lines). This is like an inverse grep (print lines that don't match). To get an ordinary grep behaviour we need to make two changes. First we add the **-n** option which turns off the automatic printing of lines. That means we need to explicitly ask Sed to print the lines we want, like this:

```
sed -n '/^#/p' /etc/fstab
```

Notice that I've enclosed the command in single quotes to prevent the metacharacter wars that can so easily result in casualties on the Linux command line.

Here's a more interesting example. Suppose I have a shell script that has lots of function definitions scattered around, and I'd like to extract the functions into a separate file. For demonstration, let's imagine a toy script like this:

```
#!/bin/bash
echo hello
function foo() {
  echo this is foo
}
# call the first function
foo
function bar() {
  echo this is bar
}
# call the second function
bar
```

First, we'll create a script with the function definitions stripped out:

```
sed '/^function/,/^}/d' demo.sh > demo2.sh
```

Here, we specify a range of line numbers based on a regex match. Text between a line beginning with function and a line beginning with } is deleted. If there are several such blocks of text in the file, they are all deleted. Turning the logic round, we can extract just the function definitions:

```
sed -n '/^function/,/^}/p' demo.sh > funcs.sh
```

Now you can't do *that* with grep!

The pattern and hold space

Even with just the few commands we've seen so far, together with a canny use of regular expressions, there's a lot you can do with Sed. But in all our examples, the output lines will appear in the same order as the input lines. We can't re-order the material in the file. To do that we need to learn about the 'pattern space' and the 'hold space'. The pattern space is the text buffer that's used for the normal line-by-line editing. The substitute command, for example, operates on the pattern space, and the **p** command outputs the contents of the pattern space.

The hold space is basically a buffer where we can park text, allowing us to re-order the content of the input stream.

Three key commands (**h**, **H** and **x**) transfer text in and out of the hold space (these and the other commands are described in the summary table, bottom p59).

Now, using the hold space typically requires us to use two or more Sed commands in a single invocation, so before we go further, let's see how we can do that. The first way is to use the **-e** option on the command line. For example, the command:

```
sed -e 's/linux/windows/' -e 's/good/bad/' somefile.txt
```

will perform both substitutions on each line. Another way is to separate the commands with semi-colons, like this:

```
sed 's/linux/windows/;s/good/bad/' somefile.txt
```

These approaches work fine but get tedious if we have more than two or three commands. A better strategy is to put the commands into a file and then reference the file on the command line. Re-casting our example to use this approach, we could create a file called (for example) **script.sed** with content like so:

```
s/linux/windows/
s/good/bad/
```

and then tell Sed to take commands from this file like this:

```
sed -f script.sed somefile.txt
```

How old is Sed?

Sed is really, really old. It was originally written by Lee E McMahon around 1974, it appeared in the *Unix Programmer's Manual* for Seventh Edition Unix in 1979. Sed was an evolution of the interactive line editor ed, and its command syntax, which looks strange to modern eyes, would have felt much more comfortable to a seasoned ed user (or even to those of you who are comfortable with the bottom line commands in *Vi*). Even the GNU version is 15 years old, dating from 1998. Sed has in its turn influenced other languages that excel at processing text; notably Perl.

Would you like to know more?

The official Sed manual is at **www.gnu.org/software/sed/manual**. Here, you'll find not only the full command reference, but some truly brain-curdling examples of scripts that emulate standard commands like *wc*, *cat*, *head*, *tail* and *uniq*. There's even a script that increments a number, proving that you really can do arithmetic in Sed (but probably don't want to.) And there's a great chapter on Sed in *Unix Power Tools* [See **LXF125**, p55).

There are a couple of benefits to putting your Sed commands in a separate script. First, we don't need to quote the commands any more because they are no longer subject to interpretation by the shell. Another benefit is that the script is a component that you can re-use.

With all these things in mind, let's return to our shell script example and give ourselves a slightly different task. Suppose we want to simply move all the function definitions up to the start of the file, with the rest of the script following. Here's the script; it's just three lines:

```
# sed script to separate out functions in a shell script
/^function/,/^}/!H
/^function/,/^}/p
$ { x; p; }
```

Function shift example

Some explanation is probably in order. The first line uses the same pair of regular expressions that we used before to identify the body of a function, except that we've added a ! to reverse the sense of the test. The **H** command adds the pattern space to the hold space. So, this line builds up in the hold buffer all those lines that are *outside* of a function definition. The second line prints out those lines that are *within* a function definition (so these will come out first, as required). Finally, the last line uses the line number shorthand $ meaning 'the last line of the input'; it swaps the hold space into the pattern space, and prints it out.

Let's run this and see what happens:

```
$ sed -n -f splitout.sed demoscript.sh
function foo() {
  echo this is foo
}
function bar() {
  echo this is bar
}
#!/bin/bash
echo hello
# call the first function
foo
# call the second function
bar
```

It's nearly right – the only problem is that the **#!** line should still be up at the top. That's not too hard to fix – I'll leave it as an exercise to the reader!

Sed in the real world

In case you're thinking that Sed is too obscure to be worth noticing, here's a statistic for you: I counted the number of uses of Sed in the system administration scripts on Ubuntu. Well actually I let the command:

```
$ find /etc -type f -exec grep -w sed {} \; 2> /dev/null | wc –l
```

count them for me. It turned up 259 examples.

The majority of these examples use Sed within a command substitution to set the value of a variable from the contents of a config file, something like this:

```
pid=$(sed 's/ //g' /var/spool/postfix/pid/master.pid)
```

All this example simply does is remove spaces from the input. The **g** switch on the end of the substitution tells Sed to make the change globally – that is, everywhere within that particular line.

Another common usage is to take the value of some existing variable and use Sed to munge it in some way. This example is taken from **/etc/network/if-pre-up.d/vlan** on Ubuntu:

```
VLANID=`echo $IFACE|sed "s/vlan0*//"`
```

Notice this uses the alternative back-quote notation for command substitution.

Here's another example, using both Awk and Sed in combination:

```
arch=`echo "$line" | awk '{print $4}' | sed 's/:$//'`
```

Here, Awk is selecting the fourth field of $line, and Sed is removing a trailing colon. Finally, this masterpiece is from **/etc/bash_completion.d/sysv-rc**:

```
valid_options=( $( \
  tr " " "\n" <<<"${COMP_WORDS[@]} ${options[@]}" \
  | sed -ne "/$( sed "s/ /\\|/g" <<<"${options[@]}" )/p" \
  | sort | uniq -u \
  ) )
```

What this impressive piece of scripting does is use Sed within a command substitution to generate a command for an *outer* Sed command. It makes my teeth curl up just thinking about it.

I am being a little unfair in presenting this example out of context. We don't know what the structure of the input data looks like, so it's hard to figure out what's happening. In my experience, the key to understanding all of these fancy text-processing pipelines is to have a very clear idea of the structure of the data that you're processing at each stage in the pipeline.

Next month we will take a good look at another of my favourite small languages – Awk. See you then. But for now, enough Sed! LXF

Arguments in Sed

Command	Description
s	Substitute text within pattern space
d	Delete contents of pattern space
p	Write pattern space to stdout
q	Quit
h	Copy the pattern space to the hold space
H	Append the pattern space to the hold space
g	Copy the hold space to the pattern space
G	Append the hold space to the pattern space
r	Read from a file into pattern space
w	Write pattern space to a file

› Sed has more commands but this is a great start.