Wizard Boot Camp, Part Nine

Jerry Peek

We're getting near the end of our series of tips that wizards should know. This month, the second in a sub-series about Linux utilities. There's more about *ln*, including some examples of how to use a data file and a combination of other utilities. And we'll see some examples of what the tiny editor *sed* can do — like editing email as it pours through a mail server.

More About ln (And the Power of Linux)

Most Linux users know about symbolic links, created by `ln -s`. The older "hard links" (without `-s`) are less common these days. Let's continue last month's final section with another linking example that shows what a hard link really is and why you'd want one.

Unlike a symbolic link, a hard link isn't a file; it's not stored in its own disk block. It's never broken (unless your filesystem is corrupt). A hard link is an entry in the directory file. (A directory file — the file named .), and one or more other names — is a list of filenames and pointers to the inodes that contain the file metadata.)

A file can have as many hard links — as many names — as you want, in any directory on that physical filesystem. (Hard links can't span filesystems.) When you remove a hard link — usually with `rm` — that doesn't remove the physical disk blocks. It simply removes the directory entry and decrements the *link count* (which you can see in `ls -l` listings). Once the link count goes to zero, the system harvests the inode and frees the disk blocks it was pointing to.

One situation where a hard link is useful is for renaming files on a live server. Say you have a directory full of image files and HTML files that display them. You need to rename the image files. Some users may have the old HTML files loaded into their browsers, so you can't simply update the HTML files and immediately rename the images. Also, there'll be time lags between the time you update the HTML files and the time you rename the image files; if there are hundreds of images, it'll be a tough change to make atomically. You might think about making symbolic links with the new filename that point to the old filename, or vice versa…but your server may be configured not to read symlinks, and you'd still have a bunch of symlink cruft to clean up.

A good answer is to give each file two simultaneous names, old and new, by making hard links with the new filenames. Since a second hard link is indistinguishable from the original filename, the server sees both names as valid. Once the dust has settled, you can remove the old links and the new ones will stand alone.

To do a huge renaming job like this, you might start with a file containing a TAB-separated list of old and new names, like this:

```
$ cat renaming_table
12345_01.gif    12345_02.gif
123456_01.gif   123457_01.gif
...
```

Test to see if any of the new names (in the second column) already exist. By throwing away *stderr*, you'll see any non-error output — that is, the names of any files that exist now. (The `2>` operator was covered in the second article of this series, "Standard I/O and open files," online at http://www.linux-mag.com/id/4705.)

```
$ ls $(cut -f2 renaming_table) 2>/dev/null
$
```

Maybe double-check for *any* duplicate names:

```
$ tr '\t' '\n' < renaming_table | sort |
uniq -d
```

Make the links:

```
$ while read old new
> do echo "=== ln $old $new ==="
> ln "$old" "$new"
> done < renaming_table
=== ln 12345_01.gif 12345_02.gif ===
=== ln 123456_01.gif 123457_01.gif ===
...
```

Any error messages should be easy to spot between the lines starting with equals signs. Make some redirects for your Apache *.htaccess* file:

```
$ sed 's@\(.*\)TAB\(.*\)@Redirect 301
/somedir/\1 http://somepath/\2@' \
  renaming_table > /tmp/htaccess.add
$ cat /tmp/htaccess.add
Redirect 301 /somedir/12345_01.gif
http://somepath/12345_02.gif
Redirect 301 /somedir/123456_01.gif
http://somepath/123457_01.gif
...
```

Then update the HTML files (possibly with a script for *sed*, *ed*, *vim*, or friends that changes the old filename to the new one). Append */tmp/htaccess.add* to the server's *.htaccess* file. Finally, remove the old links:

```
$ rm $(cut -f1 renaming_table)
```

You may never use a setup just like this one. Depending on your levels of experience and caution, you might also do more "sanity-checking" before running these commands on a live server. But the overall idea — using utilities to "slice and dice" textual data to check it, or to build other commands — is one of the things that makes Linux a power platform.

sed

We've seen *sed* the "stream editor," in examples this month and last. A common use for this utility is an application like:

```
$ sed 's/old/new/g' file.old > file.new
```

The editor reads text from its standard input, one or more editing commands from its command line or a script file, and writes the edited text to its standard output.

As terse as Perl (or more so!), with a much smaller set of operations, this tiny tool is even quite programmable — once you learn to "think *sed*." (If you like writing loops using GOTO statements, you'll love hacking *sed* scripts.) An editor like this is valuable when memory is tight — on forty-year-old PDP-11s or new embedded Linux systems. Once you know a few basics, it's also handy for operations like building commands from a list of names (as we did earlier), or even for using *sed* commands to build more sed commands (though building an `s` command from within another `s` command can make your head hurt…).

Let's look at a few more *sed* examples, from simple to somewhat complex.

1. Put a blank line between each line of *bigprog* output:

```
$ bigprog | sed G | less
```

sed reads each line of its standard input, applies the editing command (s) to that line, then writes the result to standard output. The `G` command appends a copy of *sed*'s internal "hold space" (which, in this case, is empty), followed by a newline character, to the current line (the "pattern space"). Then the pattern space is written to *stdout*. Since we haven't put anything in the hold space, this simply adds a blank line (the newline character) after each input line.

2. You can separate sed commands with semicolons or with multiple `-e` options. So, to put three blank lines between each line of *bigprog* output (so you have room to write on the printout), you could use either:

```
$ bigprog | sed -e G -e G -e G | lpr
$ bigprog | sed 'G;G;G' | lpr
```

3. As we said, *sed* applies all editing commands to each line of input. You can apply a command only to certain lines to edit by preceding the command with an *address* that describes those lines. For instance, to delete (actually, to not output) the first ten lines of the input text, use a range of line numbers followed by the `d` command:

```
$ ... | sed '1,10d' | ...
```

You want to print a log file and add `NOTE!!` to the start of all lines containing the word "error". Use a line address that matches "error" anywhere in that line, and a substitute command that adds the text to the start (`^`) of those lines:

```
$ sed '/error/s/^/NOTE!! /' logfile | lpr
```

You're using *find* to search for all files whose name ends with `.c` from any subdirectory named *mysrc*. You'll use *sed* to create `mv -i` commands to move those files into a subdirectory of each *mysrc* named *OLD*. *sed*'s `-n` option tells it not to output("print") any lines without an explicit `p` command.

```
$ find . -type f -name '*.c' -print |
sed -n 's@\(.*\)\(/mysrc/\)\(.*\)@mv -i \
       "\1\2\3" "\1\2OLD/\3"@p'
mv -i "./somedir/mysrc/aprog.c" \
       "./somedir/mysrc/OLD/aprog.c"
mv -i "./otherdir/mysrc/bprog.c" \
       "./otherdir/mysrc/OLD/bprog.c"
...
```

The first character after the `s` command becomes the delimiter; we used `@` as the delimiter because the pattern contains `/` characters. Each pair of escaped parentheses `\(...\)` holds the matching text in a numbered buffer which can be "replayed" in the replacement with `\1` for the first pattern held, `\2` for the second, and so on. The double quotes help to be sure that the shell isn't confused by filenames containing spaces or most other special characters.

We didn't redirect *sed*'s output so you could see it. You could put the output into a file named *doit*, then feed it to a shell with `sh-v doit` — which will show each *mv* command before running it.

4. Although you can put editing commands on *sed*'s com-

mand line, that can be challenging when the commands span multiple lines. Let's wrap this up with a *sed* script that your columnist used years ago. Run from *procmail*, it edited and redistributed incoming email on-the-fly. (*sed* is good here because it's small enough that it won't overload a busy mail server.)

When *procmail* received an email message, this script would hide the message origin by removing most of the message header, add new header fields, and re-send it to other recipients. The *procmail* recipe looked like this (minus most comments):

```
# Redistribute trip reports:
:0 c
* PLUS ?? ^triprep$
* !^FROM_DAEMON
* !^X-Loop: triprep@jpeek.com
| formail -c | \
/usr/bin/sed -n -f clean_triprep \|
$SENDMAIL -oi -fjpeek@jpeek.com \
joe@foo.com liz@bar.com ...
```

Each incoming email message that matches all three `*` conditions is piped to `formail-c`, which concatenates multiline header fields.

If you aren't familiar with *procmail* or email formats, no problem: we'll cover the gist of it.

The *clean_triprep* script is applied to each line of text from *sed*'s standard input. The first line of the script, `1i\`, says "before line 1 of the input text, insert…" and the required backslash continues the command to the next line, where you see the first line of text to include — namely, the new `From:` field for the outgoing message. That line also ends with a backslash, so the script also inserts the following line, an `X-Loop:` header field that helps to stop mail loops. The last line inserted above line 1 is a generic `To:` field.

The second command in the script starts with the address `1,/^$/` and an opening curly-brace that surrounds commands to be performed on lines matching that address. The address means "all lines from line 1 through the first empty line"; this describes an email header. The commands between the braces match individual header fields that start with `Subject:`, `Date:`, and others, as well as a completely blank line. All of these line addresses end with the `p` command, which outputs the matching line into the new message that we're sending (with `$SENDMAIL`). Since *sed* was invoked with its `-n` ("don't print by default") option, header fields that don't match these `/..../p` commands are not output into the new email message. This removes most traces of the incoming message except the ones we've chosen, here, to be output as-is.

The last command inside the braces, `b`, is a *sed* "branch" command. It's a "GOTO" that skips all other commands in the script. So, after we've checked each header line, and possibly output it, we get the next line of input and start over. This is a simple *sed* loop. (There are also tests that can let you choose when to branch — and labels that let you choose where to branch, if not to the end of the script. Remember, this tiny language was written long, long ago…)

The script's second command ends with the closing curly-brace. The third command is `p`, which prints every line. But, as we just saw, this *p* is only reached when the input line does not match the address `1,/^$/}` — that is, when the input line is in the mail message body (which follows the header, after a blank line).

The script's second command ends with the closing curly-brace. The third command is `p`, which prints every line. But, as we just saw, this *p* is only reached when the input line does not match the address `1,/^$/}`. That is, when the input line is in the mail message body (which follows the header, after a blank line).

Whew. So, to review before the exam, let's look at a sample incoming mail message and the outgoing one that this script generates. Incoming:

```
Return-Path: ...
Received: from foo.com; ...
Message-ID: ....
Date: Mon, 4 Jan 2002 01:10:04 -0000
From: Jerry on the road <jerry@mail...>
To: jpeek+triprep@jpeek.com
Subject: Hello from Honolulu

Hi, everyone.  After a real fiasco on Molokai
...
```

Outgoing:

```
From:   "Jerry   Peek's   Trip   Reports"
&lt;jpeek@jpeek.com&gt;
X-Loop: triprep@jpeek.com
To: "Friends, c/o" <jpeek@jpeek.com>
Date: Mon, 4 Jan 2002 01:10:04 -0000
Subject: Hello from Honolulu

Hi, everyone.  After a real fiasco on Molokai
...
```

Though today's anti-spam technology means you'd want to do something a bit more sophisticated (sending a message to each instead of a single "blind-copy" to all), the ideas still apply.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for more than 25 years. He's happy to hear from readers; see http://www.jpeek.com/contact.html.