Text processing and filtering

TACKLING TEXT

Nicholas Piccillo, Fotolia

Enjoy a crash course on some of the text-processing and -filtering capabilities found in Linux. **BY HAL POMERANZ**

Unix-like operating systems have historically been very much about text processing. Really, the Unix design religion is: Make simple tools whose output can be manipulated by others with the use of pipes and other forms of output redirection. In this article, I'll look at the wealth of Linux command-line tools for combining, selecting, extracting, and otherwise manipulating text.

wc

The *wc* (word count) command is a simple filter that you can use to count the number of lines, characters (bytes), and, yes, even the number of words in a file. Whereas counting lines and bytes tends to be useful, I rarely find myself using *wc* to count words.

You can count lines in a file with *wc -l*:

```
$ wc -l kern.log
1026 kern.log
```

If you don't specify a file name, *wc* will also read the standard input. To exploit this feature, use the following useful idiom for counting the number of files in a directory:

```
$ ls | wc -l
138
```

To count the number of bytes in a file, use *wc -c*:

```
$ wc -c kern.log
106932 kern.log
```

On a single file, *wc -c* isn't necessarily that interesting because you could see the same information in the output of *ls -l*. However, if you combine *wc* with the *find* command, you get byte counts for all files in an entire directory tree:

```
$ find /var/log -type f ➥
  -exec wc -c {} \;
79666 /var/log/kern.log.6.gz
3781 /var/log/dpkg.log.4.gz
106932 /var/log/kern.log

...
```

After I examine a few more shell tricks in the sections that follow, I'll return to this example.

head and tail

Another pair of simple text-processing filters are *head* and *tail*, which extract the first 10 or the last 10 lines from their input, respectively. Also, you can specify a larger or smaller number of lines. For example, to obtain the name of the most recently modified file in a directory, use:

```
$ ls -t | head -1
kern.log
```

Then if you wanted to see the last few lines of that file, use:

```
$ tail -3 kern.log
Nov 21 09:00:19 elk kernel: ➥
   [11936.090452] [UFW BLOCK INPUT]: ➥
   IN=eth0 OUT=...
Nov 21 09:00:21 elk kernel: ➥
   [11938.083655] [UFW BLOCK INPUT]: ➥
   IN=eth0 OUT=...
Nov 21 09:00:25 elk kernel: ➥
   [11942.134431] [UFW BLOCK INPUT]: ➥
   IN=eth0 OUT=...
```

Here's a trick for extracting a particular line from a file by piping *head* into *tail*:

```
$ head -13 /etc/passwd | tail -1
www-data:x:33:33:www-data:➥
  /var/www:/bin/sh
```

In this case, I am extracting the 13th line of */etc/passwd*, but you could easily select any line just by changing the numeric argument that is passed in to the *head* command.

Another useful feature of the *tail* command is the *-f* option, which displays the last 10 lines of the file as usual, but then keeps the file open and displays any new lines that are appended onto the end of the file. This is particularly useful for keeping an eye on logfiles – for example, *tail -f kern.log*.

cut and awk

head and *tail* are useful for selecting particular sets of lines from your input, but sometimes you want to extract particular fields from each input line. The *cut* command is useful when your input has regular delimiters, such as the colons in */etc/passwd*:

```
$ cut -d: -f1,6 /etc/passwd
root:/root
daemon:/usr/sbin
bin:/bin
...
```

The *-d* option specifies the delimiter used to separate the fields on each line, and *-f* allows you to specify which fields you want to extract. In this case, I'm pulling out the usernames and the home directory for each user. *cut* also lets you pull out specific sequences of characters by using *-c* instead of *-f*. Here's an example that filters the output of *ls -l* so that you see just the permissions flags and the file name:

```
$ ls -l | cut -c2-10,52-
otal 1540
rwxr-xr-x acpi
rw-r--r-- adduser.conf
rw-r--r-- adjtime
...
```

Darn! The output contains the header line from *ls -l*. Happily, *tail* will help with this:

```
$ ls -l | tail -n +2 | cut -c2-10,52-
rwxr-xr-x acpi
rw-r--r-- adduser.conf
rw-r--r-- adjtime
...
```

That looks better! Notice the syntax with *tail* here. The *-n* option is the alternative (POSIX-ly correct) way of specifying the number of lines *tail* should output. So *tail -10* and *tail -n 10* are equivalent. If you prefix the number of lines with *+*, as in the example above, it means *start with the specified line*. So here I'm telling *tail* to display all lines from the second line onward. The *+* syntax only works after *-n*.

cut is wonderful for lots of tasks, but the output of many commands is separated by white space and often irregular. The *awk* command is best for dealing with this kind of input:

```
$ ps -ef | awk ➥
  '{print $1 "\t" $2 "\t" $8}'
UID     PID     CMD
root    1       /sbin/init
root    2       [kthreadd]
root    3       [migration/0]
...
```

awk automatically breaks up each input line on white space and assigns each field to variables named *$1*, *$2*, and so on. *awk* is a fully functional scripting language with many different capabilities, but at its simplest, you can just use the *print* command to output particular input fields as I'm doing here.

awk also allows you to select specific lines from your input with the use of pattern matching or other conditional operators, which saves you from first having to filter your input with *grep* or some other tool. For example, suppose I wanted the filtered *ps* output above, but only for my own processes:

```
$ ps -ef | awk '/^hal / ➥
   {print $1 "\t" $2 "\t" $8}'
hal     7445    ➥
   /usr/bin/gnome-keyring-daemon
hal     7460    x-session-manager
hal     7566    ➥
  /usr/bin/dbus-launch
...
```

Here, I use the pattern match operator (*/…/*) to produce output only for lines that start with *hal* < *space* >. The command *ps -ef | awk '($1 == "hal") …'* would accomplish the same thing.

You can use the *-F* option with *awk* to specify a delimiter other than white space. This lets you use *awk* in places where you might normally use *cut*, but where you want to use *awk*'s conditional operators to match specific input lines.

Suppose you want to output usernames and home directories as in the first *cut* example, but only for users with directories under */home*:

```
$ awk -F: '($6 ~ /^\/home\//) ➥
   { print $1 ":" $6 }' /etc/passwd
sabayon:/home/sabayon
hal:/home/hal
laura:/home/laura
```

Rather than matching against the entire line, the command here uses the *~* operator pattern match against a specific field only.

sort

Sorting your output is often useful:

```
$ awk -F: '($6 ~ /^\/home\//) ➥
   { print $1 ":" $6 }' ➥
   /etc/passwd | sort
hal:/home/hal
laura:/home/laura
sabayon:/home/sabayon
```

By default, *sort* simply sorts alphabetically from the beginning of each line of input. Sometimes numeric sorting is what you want, and sometimes you want to sort on a specific field in each input line. Here's a classic example that shows how to sort your password file by the user ID field (useful for spotting duplicate UIDs and when somebody has added illicit UID 0 accounts):

```
$ sort -n -t: -k3 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
...
```

The *-n* option indicates a numeric sort, *-t* specifies the field delimiter (such as *cut -d* or *awk -F*), and *-k* specifies the field(s) to sort on (clearly they were running out of option letters).

Also, you can reverse the sort order with *-r* to get descending sorts:

```
$ ls /etc/rc3.d | sort -r
S99stop-readahead
```

```
S99rmnologin
S99rc.local
...
```

Remember the *find* command that I used *wc -c* with to get byte counts for all files under a given directory? Well, you can sort that output and then filter with *head* to get a count of the 10 largest files under your chosen directory:

```
$ find /var/log -type f -exec ➥
    wc -c {} \; | sort -nr | head
44962814 /var/log/vnetlib
24748291 /var/log/syslog
24708201 /var/log/mail.log
24708201 /var/log/mail.info
10243792 /var/log/ConsoleKit/history
3902994 /var/log/syslog.0
3782642 /var/log/mail.log.0
3782642 /var/log/mail.info.0
1039348 /var/log/vmware/hostd-7.log
804391 /var/log/installer/partman
```

uniq

When you're extracting fields with *cut* and *awk*, you sometimes want to output just the unique values. Although there's a *uniq* primitive for this, the trick is that *uniq* only suppresses duplicate lines that follow one right after the other. Therefore, you must typically sort the output before handing it off to *uniq*. For example, to get a list of all users with processes running on the current system, use the following command:

```
$ ps -ef | awk '{print $1}' ➥
   | sort | uniq
apache
dbus
dovecot
...
```

sort | *uniq* is such a common idiom that the *sort* command has a *-u* flag that does the same thing. Thus, you could rewrite the above example as *ps -ef* | *awk* '{*print $1*}' | *sort -u* instead.

The *uniq* program has lots of useful options. For example, *uniq -c* counts the total number of lines merged, and you could use this to report the number of processes running as each user, as in the following command:

```
$ ps -ef | awk '{print $1}' ➥
   | sort | uniq -c
      8 apache
      1 dbus
      8 dovecot
...
```

And with the use of another *sort* command, you could sort that output by the number of processes:

```
$ ps -ef | awk '{print $1}' ➥
  | sort | uniq -c | sort -nr
    121 root
     11 hal
      8 dovecot
      8 apache
...
```

Another useful trick is *uniq -d*, which only shows lines that are repeated (duplicated) and doesn't show unique lines. For example, if you want to detect duplicate UIDs in your password file, you could do this:

```
$ cut -d: -f3 /etc/passwd ➥
   | sort -n | uniq -d
```

In this case, I didn't get any output – no duplicate UIDs – which is exactly what I want to see.

By the way, a *uniq -u* command will output only the unique (non-duplicated) lines in your output, but I don't find myself using this option often.

paste and join

Sometimes you want to glue multiple input files together. The *paste* command simply combines two files on a line-by-line basis, with tab as the delimiter by default. For example, suppose you had a file, *capitals*, containing capital letters and another file, *lowers*, containing the letters in lower case. To paste these files together, use:

```
$ paste capitals lowers
A       a
B       b
C       c
...
```

Or if you wanted to use something other than tab as the delimiter:

```
$ paste -d, capitals lowers
A,a
B,b
C,c
...
```

But it's not really that common to want to glue files together on a line-by-line basis. More often you want to match up lines on some particular field, which is what the *join* command is for. The *join* command can get pretty complicated, so I'll provide a simple example that uses files of letters.

To put line numbers at the beginning of each line in the files, use the *nl* program:

```
$ nl capitals
     1  A
     2  B
     3  C
...
```

The *join* command could then stitch together the resulting files by using the line numbers as the common field:

```
$ join <(nl capitals) ➥
<(nl lowers)
1 A a
2 B b
3 C c
...
```

Notice the clever *<(…)* Bash syntax, which means, *substitute the output of a command in this place where a file name would normally be used*.

For some reason, when I'm using *join*, life is never this easy. Some crazy combination of fields and delimiters always seems to be the result. For example, suppose I had one CSV file that listed the top 20 most populous countries along with their populations:

```
1,China,1330044544
2,India,1147995904
3,United States,303824640
...
```

And suppose my other file listed the capital cities of all the countries in the world:

```
Afghanistan,Kabul
Albania,Tirane
Algeria,Algiers
...
```

What if my task were to connect the capital city information with each of the 20 most populous countries? In other words, I want to glue the information in

the two files together with the use of field 2 from the first file and field 1 from the second file. The complicated thing about *join* is that it only works if both files are sorted in the same order on the fields you're going to be joining the files on. Normally I end up doing some presorting on the input files before giving them to *join*:

```
$ join -t, -1 2 -2 1 <(sort -t, ➥
   -k2 most-populous) <(sort cities)
Bangladesh,7,153546896,Dhaka
Brazil,5,196342592,Brasilia
China,1,1330044544,Beijing
...
```

The options to the *join* command specify the delimiter I'm using (*-t,*) and the fields that control the join for the first (*-1 2*) and second (*-2 1*) files. Once again, I'm using the *<(…)* Bash syntax, this time to sort the two input files appropriately before processing them with *join*.

The output isn't very pretty. *join* outputs the joined field first (the country name), followed by the remaining fields from the first file (the ranking and the population), followed by the remaining fields from the second file (the capital city). The *cut* and *sort* commands can pretty things up a little bit:

```
$ join -t, -1 2 -2 1 <(sort -t, ➥
   -k2 most-populous) <(sort cities) ➥
   | cut -d, -f1,3,4 | sort -nr -t, -k2
China,1330044544,Beijing
India,1147995904,New Delhi
United States,303824640,Washington D.C.
...
```

Examples like this are where you really start to get a sense of just how powerful the text-processing capabilities of the operating system are.

split

Joining files together is all well and good, but sometimes you want to split them up. For example, I might split my password-cracking dictionary into smaller chunks so that I can farm out the processing across multiple systems:

```
$ split -d -l 1000 dictionary ➥
   dictionary.
$ wc -l *
  98569 dictionary
   1000 dictionary.00
   1000 dictionary.01
   1000 dictionary.02
...
```

Here, I'm splitting the file called *dictionary* into 1000-line chunks (*-l 1000*, is actually the default) and assigning *dictionary* as the base name of the resulting files. Then I want *split* to use numeric suffixes (*-d*) rather than letters, and I use *wc -l* to count the number of lines in each file and confirm that I got what I wanted.

Note that you can also specify -, meaning the standard input, instead of a file name. This can be useful when you want to split the output of a very verbose command into manageable chunks (e.g., *tcpdump | split -d -l 100000 - packet-info*).

tr

The *tr* command allows you to transform one set of characters into another. The classic example is mapping uppercase letters to lowercase. For this example, to transform the *capitals* file I used previously, I'll use:

```
$ tr A-Z a-z < capitals
a
b
c
...
```

But this is a rather silly example. A more useful task for *tr* is this little hack for looking at data under */proc*:

```
$ cd /proc/self
$ cat environ
GNOME_KEYRING_SOCKET=/tmp/➥
   keyring-lFz8t4/socketLOGNAME➥
   =halGDMSESSION=default...
$ tr \\000 \\n <environ
GNOME_KEYRING_SOCKET=/tmp/➥
   keyring-lFz8t4/socket
LOGNAME=hal
GDMSESSION=default
...
```

Typically */proc* data are delimited with nulls (ASCII zero), so when you dump */proc* to the terminal, everything just runs together, as shown in the output of the *cat* command above. By converting the nulls (*\000*) to newlines (*\n*), everything becomes much more readable. (The extra backwhacks (\) in the *tr* command here are necessary because the shell normally interprets the backslash as a special character. Doubling them up indicates that the backslash should be taken literally.)

Instead of converting one set of characters to another, you can use the *-d* option simply to delete a particular set of characters from your input. For example, if you don't happen to have a copy of the *dos2unix* command handy, you can always use *tr* to remove those annoying carriage returns:

```
$ tr -d \\r <dos.txt >unix.txt
```

Or, for a sillier example, here's a way for all you fans of *The Matrix* to get a spew of random characters in your terminal:

```
$ tr -d -c [:print:] </dev/urandom
```

Here I'm using *[:print:]* to specify the set of printable characters, but I'm also employing the *-c* (compliment) option, which means *all characters not in this set*. Thus, I end up deleting everything except the printable characters.

Conclusion

This has been a high-speed introduction to some of the text-processing and -filtering capabilities in Linux, but of course it really only just scratches the surface. Lots of sites on the Internet have more examples and ideas for you to study, including shelldorado.com, commandlinefu.com, and the weekly blog I co-author with several friends at blog.commandlinekungfu.com.

The online manual pages can help a lot too – and don't forget *man -k* for keyword searches if you've forgotten a command name or just aren't sure where to start! But really, the best teachers are practice, practice, and practice. I've been using Unix and Linux systems for more than 20 years, and I'm still learning things about the shell command line. ■

THE AUTHOR

Hal Pomeranz is the Founder and Technical Lead of Deer Run Associates, an IT and Information Security consulting firm. He is also a Faculty Fellow of the SANS Institute and the course developer and primary instructor for their Linux/Unix Security certification track (GCUX). And, yes, he could replace you with a very small shell script.