Wizard Boot Camp, Part Eight

Jerry Peek

If you're a heavy user of scripting languages, like Python, you may find it faster to write a few lines of Python code than to type a long command line (or a loop) at a shell prompt. But Linux systems have a long heritage — from Unix, and from other systems too — of power-packed utility programs that do a particular job particularly well. Even if Ruby or Perl code drips from your fingers, passing some data to a shell or a utility can sometimes make quick work of hacks that aren't part of Your Favorite Interpreter's feature set.

Some of the contents of */bin* and */usr/bin* can be surprising — even for the grey-bearded wizards who've run Linux systems for decades. Here's the first of a several "Boot Camp" columns to browse through and ask yourself whether you've used all of the non-obvious features of these obvious utilities. You long-time wizards might also want to read the series "What's GNU in Old Utilities?" — its first article is online at http://www.linux-mag.com/id/2048.

cat

What use is a utility that reads files, or standard input, and writes them to standard output?

One good reason is `cat`'s filtering options: `-b` and `-n` for line numbering; `-s` for removing multiple empty lines; and `-v` (and more) for showing "unprintable" characters. (See the `man` or `info` page.)

Or consider this little shell script named *foo*:

```
#!/bin/sh
{
 prog1
 cat - "$@"
 prog2
} > outfile
```

It's using `cat` as "plumbing" to insert text from the script's standard input, and from any files named on `foo`'s command line, into the proper sequence of output being written to `outfile`. So, if you run `foo` this way:

```
$ aprog | foo infile[12]
```

then *outfile* will hold the following data, in the following order:

1. The output of *prog1*
2. The output of *aprog*
3. The contents of *infile1*
4. The contents of *infile2*
5. The output of *prog2*

Like many standard Linux utilities, if you put a single dash (`-`) on its command line, `cat` will read its standard input at that point. Here, `cat` reads its standard input (which comes from `aprog`), then reads the files *infile1* and *infile2* (which the shell expanded from the wildcard pattern `infile[12]`).

date and touch

These tiny utilities show, and set, times of files and the time on the system clock.

By itself, `date` shows the time in your system's timezone. Setting the TZ environment variable tells `date` to use a timezone different than the system default; timezone names are typically listed under */usr/share/zoneinfo*. So, for instance, to check the times in your location, in Poland, and in western Australia, you could pass temporary values of TZ to `date` this way:

```
$ date
Wed Jan 30 11:54:12 CST 2008
$ TZ=Poland date
Wed Jan 30 18:55:09 CET 2008
$ TZ=Australia/Perth date
Thu Jan 31 02:55:31 WST 2008
```

Options include `-u` to show UTC (GMT) time and `-R` for RFC-2822 time (used in email and other messages). An argument starting with a plus sign lets you choose almost any output format. As an example:

```
$ date "+It's %A at %l:%M %p."
It's Wednesday at 12:27 PM.
```

Other uses are in setting file timestamps and using them for time-tracking. For instance, typing `touch last-report-printed` saves the current time as the last-modified date of the file *last-report-printed*. You can use that timestamp file later:

```
$ date -r last-report-printed
Wed Jan 30 12:01:31 CST 2008
$ find /reports -type f -newer last-
report-printed -print0 | xargs -0 lpr
```

The first command shows the timestamp. The second searches the directory */reports* for all files modified more recently than

the timestamp file, then sends those files to the printer with *lpr*. (The `-print0` and `-0` options keep "unusual" filenames from causing problems.)

The `touch` utility can also copy timestamps and set arbitrary timestamps. The first command below sets the timestamp of the file named *last-backup* to January 23 at 11:45 PM (2345) of the current year. The second command copies the timestamp of *last-report-printed* onto the file *previous-report-printed*:

```
$ touch -t 01232345 last-backup
$ touch -r last-report-printed previous-
report-printed
```

echo

This tiny utility — typically built into a shell — is useful for emitting bits (or bunches) of text. Combine it with a shell variable for storing text you need to use again. Here are a few examples:

- Send a test email message twice — repeated the second time via the shell history operator `!ec` (or just `!e` or maybe `!!`):

```
$ echo 'get this?' | mail joe@a.com
  ...later...
$ !ec
echo 'get this?' | mail joe@a.com
```

- Store a line of text in a shell variable, including escape sequences for TAB and a newline that *echo* will interpret using its `-e` option. Send five of them to the printer (plus a final empty line), without or with line numbers:

```
$ x='left column\tright column\n'
$ echo -e "$x$x$x$x$x" | lpr
$ echo -e "$x$x$x$x$x" | cat -n | lpr
```

- Not sure what a variable contains? Show it with `echo` and filter it with `cat-v` or `cat-A`:

```
$ echo "$IFS" | cat -A
 ^I$
$
```

Here, `$IFS` contains a space, a TAB, and a newline.

- A line with the date and time can help you track what happened when.

```
while sleep 60
do
  echo -e "\n===== $(date) ====="
  netstat
done > netstat.log
```

Then *netstat.log* will contain data like this:

```
==== Wed Jan 30 12:44:03 CST 2008 ====
Active Internet connections
tcp     0    0  somehost ...

==== Wed Jan 30 12:45:03 CST 2008 ====
Active Internet connections
tcp     0    0  somehost ...
```

We're redirecting the standard output of all `echo` and `netstat` calls within the `while` loop to the file *netstat.log*.

grep

This utility is invaluable for finding things, and most of its uses are so obvious that they don't belong in this column. A few possibly-non-obvious tips, though:

- If the files might not have readable text, pipe `grep`'s output through `cat-v` to protect your terminal.

- The GNU color-highlighting option `—color` makes it much easier to spot a match in a mass of output text. But if you pipe colored text through a filter like `cat -v`, `cat` will change colored areas into escape-sequence gibberish. Try using two `grep`s — with the highlighting done after the filtering:

```
grep 'xyz$' */* | cat -v | grep —color
'xyz$'
```

- If *grep* finds lots of matching text, piping the output to the *less* pager can help. Using `sed G` (as we'll see in a future column) will double-space the matching lines, separating them for clarity:

```
grep 'xyz$' */* | sed G | less
```

If you also use a `less` search command (here, type `/xyz$` after `less` starts running), then `less` will highlight the matching text. (Another useful `less` option here is `-a`, which skips all matches shown on the current screen when you type `n` to repeat the search.

- The Swiss-Army-Knife `find` is another utility that's so obvious we shouldn't even need to mention it. Still, it's handy with `grep` for searching a filesystem tree. As we saw earlier (in the `date` section), use its `-print0` option with `xargs-0`. Adding the `grep` option `-H` makes sure that ev-

ery filename is output. (Without `-H`, in case `xargs` passes just a single filename to `grep`, `grep` wouldn't show the filename.)

gzip and bzip2

Compressing files is a no-brainer when you need more disk space, and `gzip` / `gunzip` / `zcat` are often-used. `bzip2` / `bunzip2` / `bzcat` often compresses more effectively, though the compression phase may be slower. Here are some things you might not know about these power tools:

Not all data is worth compressing. Data that has enough redundancy, or gaps of "filler" (like the NUL bytes in a *tar* archive), can be reduced in size by compression. For instance, a TIFF photo with big areas of a single color, or a text file with lots of contiguous whitespace, will compress. A JPEG photo or an already-compressed file usually won't compress well.

Using `gzip` on an already-compressed file usually won't hurt, although the file size may increase a little. (By the way, `gzip` refuses to compress files with names ending in *.gz*.) Still, to avoid needless *gzip* ping, you can do "trial compression" with the `-v` option (which shows compression percentage) and a temporary file:

```
$ gzip -cv photo.tif > gz.gz
photo.tif:     96.5%
```

The `-c` option writes the compressed data to *stdout*. (The output of `-v` goes to *stderr*.) If the compression ratio was high enough, overwrite the original with `mv gz.gz photo.tif`; otherwise, use `rm gz.gz`.

When `-c` isn't used, `gzip` makes the compressed archive the same file mode (access permissions) as the original. Our two-step method above doesn't do that, but you can — by transferring the original file mode onto the compressed copy before you run `mv`:

```
$ chmod —reference=photo.tif gz.gz
```

Yup, this all is a likely candidate for a shell script that does conditional compression, testing to see whether `gzip` or `bzip` is better, etc. The `bash` operator `2>` redirects *stderr* into a file. You might pre-check the file type with *file* so your script won't try to compress already-compressed data types.

When sending compressible data across a network, you can compress data on-the-fly to reduce transmission time. (This is typically only useful if the network bandwidth is low, and/or if the systems on both ends of the link aren't so overloaded that they bog down the compression.)

If you're using *ssh*, and if both systems support it, the `ssh -C` option will compress network traffic. Otherwise, use built-in compression in utilities (like *tar*) that have it — or pipe data through `gzip` and `zcat`. Here are two examples:

Let's say you're using a portable Linux laptop without a printer and you want to print some text files on your office printer. You could compress their data in one of these two ways:

```
$ pr report*.txt | ssh -C foo.com lpr
$ pr report*.txt | gzip | ssh foo.com
'zcat | lpr'
```

Although *scp* has a `-r` option to copy directory trees, using *tar* preserves more of the metadata. If the versions of *tar* on both systems support gzip compression, use the first version below. Or, if the remote system doesn't support `tar z`, try the second:

```
$ tar czf - adir | ssh foo.com 'cd bar tar
xzvf -'
$ tar czf - adir | ssh foo.com 'cd bar
zcat | tar xvf -'
```

ln

Links are commonly used to point from one filesystem location to another. They also have a more surprising use: to make a program that does several things, deciding what to do by checking the name it was called with. Here's an example: a single program in */usr/bin* that sets, shows, and removes *at* jobs:

```
$ ls -l at*
-rwsr-sr-x 1 ... at
lrwxrwxrwx 1 ... atq -> at
lrwxrwxrwx 1 ... atrm -> at
```

These are simple to write as shell scripts. The script simply needs to test its name:

```
myname=${0##*/}  # Program name without path
case "$myname" in
at) ... ;;
atq) ... ;;
atrm) ... ;;
*) echo "$0 ERROR: ..." 1>&amp;2; exit 1
;;
esac
```

Before you reach for Python, Perl, Ruby, or another scripting language of your choice, make sure you're making the most of the shell utilities that you have at your fingertips!

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.