The Importance of Command Line Literacy

The Oldest Interface is Still One of the Best.

By Jerry Peek

Have you sat down with that grey-haired system administrator and watched what's happening as her (or his) fingers fly over the keyboard? Ever wondered why they're still Stuck In The Stone Age, typing text instead of using the obviously easier mouse and menus? Suspect that person is a geezer who can't learn new techniques?

Maybe they did learn Linux (actually, Unix) before everyone had windows on a monitor. But there's a reason why they keep a terminal window (or ten) open on their desktops: this is a power platform that can be the fastest way to work. It remembers what you've done (so it's easy to repeat and/or modify), has built-in programmability (need a loop? no worries!), and gives you full access to the library of hundreds of Linux utility programs that were designed for just this environment. A terminal and utilities do jobs that a GUI can't touch.

Less Power Than your Wristwatch

The Linux command line was born around the time Linus Torvalds drew his first breath: in the days of small Unix systems with tiny amounts of memory.

Back then, a *tt*y was a teletype — those clattering machines that also sent and received telegrams over 110-baud lines.

Using Unix meant typing on a keyboard and waiting for the (often-busy) system to respond. Every byte and CPU cycle were precious.

The Unix command line evolved to help people cope with this environment.

Not the Only Way

We aren't saying that you'll *only* want to use the command line. Understanding where it's useful and how to use it will let you choose whether to use your terminal window or a GUI.

So What?

Why would you want to type on a command line instead of (say) choosing commands from a GUI menu? There are more reasons than we can cover in three pages. Here are some of the best:

SIDEBAR ONE: Typing Ahead

In these days of speedy systems with enough power to burn that you can have 3-D multicolor displays which morph with your mouse, the typeahead feature of Linux device drivers can still be worth knowing about.

Typeahead means that the memory buffer which reads your keystrokes and passes them on to the kernel can read input and save it until the rest of the system is ready to receive it. This was very useful in the days of super-slow processors that could barely cope with the keyboard interrupts generated each time you pressed a key.

Once you're confident on the command line, you can still use typeahead today. For instance, if you're trying to coax work out of a remote system through a congested network link, you may not need to wait to see a response. If you're typing simple commands at a shell prompt, or using a keyboard-driven editor like *vim* that you know well enough to trust the commands you type, try typing ahead! It doesn't always work: for instance, keystrokes like CTRL-C or CTRL-Z flush the typeahead buffer. But typeahead is still available, ready to receive commands at molasses speed and execute them when the processor is ready.

SIDEBAR TWO: Typing Ahead

One pitfall for new command-line users is this: Utilities were born in the days when you wouldn't want to wait (and wait...) for a system to ask "Are you sure?" before, say, removing a file. So it's best to think before you type — or, at least, to double-check before you press the RETURN key to run a command.

Once you're used to decisiveness by default, the constant pop-up confirmations of a GUI can actually be frustrating.

- Linux *utility programs* — *sort*, *cut*, *join*, and hundreds of others — are powerful and flexible tools that are ready to run with a few keystrokes. *Options* let you fine-tune (or completely change) how utilities work.

- Powerful *I/O redirection* lets you save data to files, read data from files, and connect a series of utilities together in a *pipeline* that lets you custom-build exactly what you need …again, with just a few keystrokes. GUI applications — which have to pack all possible operations into a set of menus and checkboxes — can't be nearly as flexible or powerful.

- Linux commands can operate on *many* files at once. *Filename completion* lets you specify filenames by typing a few characters and the TAB key. Some GUIs have that feature as well. But most GUIs don't have *wildcards*, which let you specify one or many files with just a few keystrokes. The pattern `[A-Z]*.txt` matches any filename that starts with an uppercase letter and ends with `.txt`; that could be many files. (`[:upper:]*.txt` handles non-English filenames too.) Use `/home/*/ data0[0-5]` to match all files named *data00* through *data05* in all directories under */home* — for instance, */home/ amy/data01*, */home/randolph/data06*, and so on.

- A shell is actually an interpreter for a programming language: the language of command lines. It puts powerful loops, tests, variables, I/O manipulation, error-trapping, and more at your fingertips when you type a command line.

Put these same command lines in a script file, and you have a shell script — which lets you repeat those stored commands anytime you want to. It's the same language from the command line or from a script file. Learn the shell "language" and you can use it both places!

One of the authors of the Apress book "From Bash to Z Shell: Conquering the Command Line" gave a good example of this. "Earlier this week we had to rearrange a bunch of source files…arranging according to newer directory structure conventions. Lots of little jobs like renaming files to lower case would be really laborious from a GUI. From the shell I could do all the stuff on the fly as fast as my colleague explained what needed doing. He was quite impressed and it's convinced him to try to learn more about it all himself."

- *Command substitution* lets you use the output of one command as part of another command line. If you haven't used a shell before, this is hard to explain…but, trust us, it's very powerful. For instance, you can run a file-search program like *find*, then run another command on the files that *find* found.

This kind of flexibility is unheard-of in a GUI menu system.

Behind the Scenes

If you're new to the command line, understanding some basics can help you fit the pieces together.

On modern systems, the old teletype device has been replaced by either a *console terminal* (which fills your screen — like the virtual terminals that appear when you press Alt+ Fn or Ctrl+ Alt+ Fn; see the *console* manual page for details) or a *terminal window* (which is like other GUI windows).

Running "inside" the terminal is a shell. A *shell* is a Linux program that's designed for running both *internal* "built-in" commands (like *cd* to change the shell's current directory) and *external* commands (like *grep*, *perl*, *vim*, and many more). There are several different shells; *bash*, *tcsh*, and *zsh*(" Z shell") are probably the most popular, but you can choose whatever shell you like.

A shell is a program like any other on Linux. So a shell can run another shell; that's the situation when you run a *shell script* (a text file with one or more command lines, which start with a command like *cd*, *grep*, *perl*, *vim*, …).

A shell does the following steps, over and over:

1. Unless it's reading from a script file, it outputs a prompt (like `$` — or anything you set it to) and waits for Step 2.

2. Reads a command line from the terminal (followed by the RETURN key) or from a script file.

3. Parses (interprets) the command line to find the command name (s), special characters like `*` and `?` (*wildcards* that let you specify one or many filenames without much typing), `|` and `>` (redirection symbols, which route programs' input and/or outputs).

4. Runs the command (s) and waits for them to finish.

5. Repeats the steps for the other commands until the script file ends, or the user types *exit* or CTRL-D. Then the shell

program terminates — and the terminal window closes, or virtual console prints another `login:` prompt.

Examples

We can't give many examples in a three-page column that also explains the basics. If you'd like to see of what you can do in this feature-rich environment, browse through some of the past six years' *Power Tools* columns at http://linux-mag.com/.

Being able to use any command in a loop — to do something repetitive, with just a few command lines — is one of the most obvious reasons to use a shell instead of a GUI. Here's a loop that finds every filename ending with *.c* in the current directory and makes a copy with `OLD_` preceding the filename. (The `-p` is an option that tells *cp* to give the copy the same last-modification date as the original file.) Linux utilities normally run silently unless they have errors…but, if you wanted to see what files *cp* is copying, you could add the `-v` or `—verbose` option:

```
$ for f in *.c
> do cp -p $f OLD_$f
> done
```

The `$` and `>` are prompts. These tell you that the shell is waiting for commands. You can customize prompts to show much more information: the name of the current directory, the computer name, your username, the date and time, and almost anything else.

> This is an example of the power of different shells: if you know how to use more than one, you can choose the best shell for a job

To repeat the previous loop at some later time, you could navigate up in the shell's history list by pressing the up-arrow key on your keyboard — and, when you see the loop you typed before, press RETURN to re-run it. It's also possible to edit the loop interactively at this point. Or, you could use one of the history commands available on old teletypes. This command repeats the previous *for* loop:

```
$ !for
```

Not all shells will let you repeat an entire loop with! for; they may only repeat the first line. The shell will show the loop commands, then re-execute them. This is an example of why Linux commands generally do what you ask without confirmation: in this case, *cp* will replace the previous backup files with the new ones. (If you didn't want that, you could have added a *cp* option like *-i*, " interactive," which asks before overwriting.)

Want to back up every filename ending in `.c` in the entire directory tree — which could be tens or thousands of directories? Replace the `*.c` in the previous loop with command substitution and the *find* utility. Here we'll change the *cp* command line to make copies whose names end with `.bak`:

```
$ for f in $(find . -name '*.c' -print)
> do cp -p $f $f.bak
> done
```

That loop has a couple of subtle problems that you'll understand once you learn more about how shells work. (You could add `-type f` to tell *find* to copy only files — not directories whose names happen to end with `.c`. Also, if any names have spaces or special characters, you should be sure they're handled correctly.) The Z Shell has a built-in `**` recursive-matching operator that's easier than *find*. Here's the previous loop using Z shell:

```
% for f in **/*.c
for> do cp -p $f $f.bak
for> done
```

This is an example of the power of different shells: if you know how to use more than one, you can choose the best shell for a job. (Many Linux users only learn *bash*…and that's plenty!)

Maybe your IDE (programming environment) has a function that can copy your source code. But can it copy any or all directories?

Can you choose the backup method, and whether the backups keep the original timestamps? Can you add conditional tests to choose which files are backed up — for instance, only files modified in the past day, or only files with more than 100 lines, or only files containing a call to the function *foobar()*, or…? If you can, that's a sophisticated IDE! But what if you want to make a *zip* or *tar* archive instead of a backup? Or if you don't want to copy the files; you want to rename them? Or….

Or maybe you should take the time to learn the fundamentals of shells and utilities. Once you learn the basics (like *for* loops) you can use those same fundamentals with the hundreds of utilities to do thousands of operations — with just a few keystrokes. Grab a good book (like the Apress book mentioned earlier; shameless plug: this author wrote part of it) and get ready to be a (grey-haired?) Linux guru yourself.

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.