Unknown Bash Tips and Tricks For Linux by Carla Schroder

Familiarity breeds ennui, and even though Bash is the default Linux command shell used daily by hordes of contented users, it contains a wealth of interesting and useful features that don't get much attention. Today we shall learn about Bash builtins and killing potential.

Bash Builtins

Bash has a bunch of built-in commands, and some of them are stripped-down versions of their external GNU coreutils cousins. So why use them? You probably already do, because of the order of command execution in Bash:

1. Bash aliases
2. Bash keywords
3. Bash functions
4. Bash builtins
5. Scripts and executable programs that are in your PATH

So when you run `echo, kill, printf, pwd,` or `test` most likely you're using the Bash builtins rather than the GNU coreutils commands. How do you know? By using one of the Bash builtins to tell you, the `command` command:

```
$ command -V echo
echo is a shell builtin

$ command -V ping
ping is /bin/ping
```

There are no man pages for the Bash builtins, but there is a backwards `help` builtin command that displays syntax and options:

```
$ help echo
echo: echo [-neE] [arg ...]
    Write arguments to the standard output.

    Display the ARGs on the standard output followed by a newline.

    Options:
      -n        do not append a newline
      -e        enable interpretation of the following backslash escapes
[...]
```

I call it backwards because most Linux commands use a syntax of `commandname --help`, where `help` is a command option instead of a command.

The `type` command looks a lot like the `command` builtin, but it does more:

```
$ type -a cat
cat is /bin/cat

$ type -t cat
file

$ type ll
ll is aliased to `ls -alF'

$ type -a echo
echo is a shell builtin
echo is /bin/echo

$ type -t grep
alias
```

`type` identifies builtin commands, functions, aliases, keywords (also called reserved words), and also binary executables and scripts, which it calls *file*. At this point, if you are like me, you are grumbling "How about showing me a LIST of the darned things." I hear and obey, for you can find these delightfully documented in the The GNU Bash Reference Manual indexes. Don't be afraid, because unlike most software documention this isn't a scary mythical creature like Sasquatch, but a real live complete command reference.

The point of this little exercise is so you know what you're really using when you type a command into the Bash shell, and so you know how it looks to Bash. There is one more overlapping Bash builtin, and that is the `time` keyword:

```
$ type -t time
keyword
```

So why would you want to use Bash builtins instead of their GNU cousins? Builtins may execute a little faster than the external commands, because external commands have to fork an extra process. I doubt this is much of an issue on modern computers because we have horsepower to burn, unlike the olden days when all we had were tiny little nanohertzes, but when you're tweaking performance it's one thing to look at. When you want to use the GNU command instead of the Bash builtin use its whole path, which you can find with `command`, `type`, or the good old not-Bash command `which`:

```
$ which echo
/bin/echo

$ which which
/usr/bin/which
```

Bash Functions

Run `declare -F` to see a list of Bash's builtin function names. `declare -f` prints out the complete functions, and `declare -f [function-name]` prints the named function. `type` won't find list functions, but once you know a function name it will also print it:

```
$ type quote
quote is a function
quote ()
{
    echo \'${1//\'/\'\\\'\'}\'
}
```

This even works for your own functions that you create, like this simple example `testfunc` that does one

thing: changes to the /etc directory:

```
$ function testfunc
> {
> cd /etc
> }
```

Now you can use `declare` and `type` to list and view your new function just like the builtins.

Bash's Violent Side

Don't be fooled by Bash's calm, obedient exterior, because it is capable of killing. There have been a lot of changes to how Linux manages processes, in some cases making them more difficult to stop, so knowing how to kill runaway processes is still an important bit of knowledge. Fortunately, despite all this newfangled "progress" the reliable old killers still work.

I've had some troubles with bleeding-edge releases of KMail; it hangs and doesn't want to close by normal means. It spawns a single process, which we can see with the `ps` command:

```
ps axf|grep kmail
 2489 ?     Sl  1:44 /usr/bin/kmail -caption KMail
```

You can start out gently and try this:

```
$ kill 2489
```

This sends the default SIGTERM (signal terminate) signal, which is similar to the SIGINT (signal interrupt) sent from the keyboard with Ctrl+c. So what if this doesn't work? Then you amp up your stopping power and use SIGKILL, like this:

```
$ kill -9 2489
```

This is the nuclear option and it will work. As the relevant section of the GNU C manual says: "The SIGKILL signal is used to cause immediate program termination. It cannot be handled or ignored, and is therefore always fatal. It is also not possible to block this signal." This is different from SIGTERM and SIGINT and other signals that politely ask processes to terminate. They can be trapped and handled in different ways, and even blocked, so the response you get to a SIGTERM depends on how the program you're trying to kill has been programmed to handle signals. In an ideal world a program responds to SIGTERM by tidying up before exiting, like finishing disk writes and deleting temporary files. SIGKILL knocks it out and doesn't give it a chance to do any cleanup. (See `man 7 signal` for a complete description of all signals.)

So what's special about Bash `kill` over GNU `/bin/kill`? My favorite is how it looks when you invoke the online help summary:

```
$ help kill
```

Another advantage is it can use job control numbers in addition to PIDs. In this modern era of tabbed terminal emulators job control isn't the big deal it used to be, but the option is there if you want it. The biggest advantage is you can kill processes even if they have gone berserk and maxed out your system's process number limit, which would prevent you from launching `/bin/kill`. Yes, there is a limit, and you can see what it is by querying /proc:

```
$ cat /proc/sys/kernel/threads-max
61985
```

With Bash `kill` there are several ways to specify which signal you want to use. These are all the same:

```
$ kill 2489
$ kill -s TERM 2489
$ kill -s SIGTERM 2489
$ kill -n 1 2489
```

`kill -l` lists all supported signals.

If you spend a little quality time with `man bash` and the GNU Bash Manual I daresay you will learn more valuable tasks that Bash can do for you.