DAVE TAYLOR

Dealing with Signals

By handling signals in your bash scripts, you can provide features that are otherwise difficult, such as telling your script to reread its configuration file after it's already been started.

This month, I thought it would be interesting to take a bit of a detour from my usual multi-month programming projects and instead focus on a specific topic that is of great importance to people writing longer scripts: signal management.

Signals are numeric messages sent to running applications from the operating system, other applications or the user, and they generally invoke a specific response like "shut down gracefully", "stop running so I can put you in the background" or "die!"

Most likely, you've used the kill command to send signals to different programs, but if you've ever pressed Ctrl-C or Ctrl-Z to stop a running app, you've also sent signals to a running application.

A signal is managed in a cascading manner. It's sent to the application or script, then if the application doesn't have a specific handler (signal management or response function), it's pushed back to the shell or operating system. Some signals can't be managed within individual apps, like SIGKILL, which is caught by the operating system and immediately kills the running application (including the shell: SIGKILL your login shell and you just logged out).

Most likely, you've used the kill command to send signals to different programs, but if you've ever pressed Ctrl-C or Ctrl-Z to stop a running app, you've also sent signals to a running application.

To start this journey, let's find out what signals your version of Linux can handle. Do this by typing `kill -l` (that's a lowercase L, not the digit 1):

```
$ kill -l
1) SIGHUP     2) SIGINT      3) SIGQUIT    4) SIGILL
5) SIGTRAP    6) SIGABRT     7) SIGEMT     8) SIGFPE
9) SIGKILL   10) SIGBUS     11) SIGSEGV   12) SIGSYS
13) SIGPIPE  14) SIGALRM    15) SIGTERM   16) SIGURG
17) SIGSTOP  18) SIGTSTP    19) SIGCONT   20) SIGCHLD
21) SIGTTIN  22) SIGTTOU    23) SIGIO     24) SIGXCPU
25) SIGXFSZ  26) SIGVTALRM  27) SIGPROF   28) SIGWINCH
29) SIGINFO  30) SIGUSR1    31) SIGUSR2
```

Most of these are uninteresting. The cool ones are SIGHUP, which is sent on a "hangup" or the user logging out; SIGINT, which is a simple interrupt (Ctrl-C, usually); SIGKILL, the "terminate with extreme prejudice" of signals; SIGTSTP, which is Ctrl-Z; SIGCONT, which is what the application gets from the shell commands fg and bg subsequent to a SIGTSTP; SIGWINCH, which is for window system events like a window resize; and SIGUSR1 and SIGUSR2, which are intended for interprocess communication.

Let's write some code to see what happens, shall we? Signals are caught with the "trap" built in, and the general format of these signal mapping commands is exemplified with:

```
trap 'echo "Ctrl-C Ignored" ' INT
```

How do we play with that as a shell script? Here's an easy way:

```
#!/bin/bash

trap 'echo " - Ctrl-C ignored" ' INT
while /usr/bin/true ; do
  sleep 30
done

exit 0
```

Did you catch the infinite loop there? It's barely using any resources because most of its time is spent in "sleep", but if you don't do something to end it, this script will run forever or until the Mayans are proven right two years from now—one of the two.

Let's look at a more flexible way to manage signals by creating shell script functions:

```
sigquit()
{
   echo "signal QUIT received"
}

sigint()
{
   echo "signal INT received, script ending"
   exit 0
}
```

```
trap 'sigquit' QUIT
trap 'sigint'  INT
trap ':'       HUP      # ignore the specified signals

echo "test script started. My PID is $$"
while /usr/bin/true ; do
  sleep 30
done
```

Run this then from another terminal window and shoot some signals at it.

Now, let's get that script started and watch what happens when we send a few different signals:

```
$ ./test.sh
test script started. My PID is 25309
signal QUIT received
signal INT received, script ending
$
```

Perfect! To send the signals, execute the following commands from a different terminal window:

```
$ kill -HUP  25309
$ kill -QUIT 25309
$ kill -INT  25309
```

Armed with this useful script, let's have a look at how to handle a more complex signal like Ctrl-Z within a shell script.

Stop! Don't Stop!

I'm going to create a scenario here rather than just going through the intellectual exercise. In a complex script, you realize that you have certain passages where you need to ignore the TSTP signal (SIGTSTP or Ctrl-Z or signal number 18) and other spots where it's fine to stop and restart. Can it be done?

To start working out a solution, I'll create a function that not only handles the specified signal, but also disables itself after a single invocation:

```
sigtstp()
{
  echo "SIGTSTP received" > /dev/tty
  trap - TSTP
  echo "SIGTSTP standard handling restored"
}
```

Invoke `trap - signal` somewhere else in the script,

and you've reset that signal handler, so if I have the line:

```
trap 'sigtstp' TSTP
```

right before the section where I don't want the Ctrl-Z to work, it'll ignore that first Ctrl-Z, then reset the signal handler and work as expected the second time you press that key.

More useful is to ignore all Ctrl-Z stop signals until you're ready to deal with them, and that's quite easily done with the minimalist:

```
trap : TSTP  # ignore Ctrl-Z requests
```

And, then when you're ready to receive them again:

```
trap - TSTP  # allow Ctrl-Z requests
```

Experimentation will show that there are some weird terminal buffering issues associated with SIGTSTP, however, so don't be surprised if you have a signal handler that has output. In this particular instance, it won't show up until the script quits.

Experimentation will show that there are some weird terminal buffering issues associated with SIGTSTP, however, so don't be surprised if you have a signal handler that has output.

Reading a Configuration File

Let's look at a more practical example. Say you have an admin script that is always supposed to be running as a dæmon, but occasionally you want to tweak its configuration file and have it reread its setup (a lot faster than killing and restarting it).

Further, let's use SIGUSR1 for this task, as that is its intended use, so we're using the kernel's signal handling subsystem in the manner it was intended.

Reading a configuration file might be something as simple as:

```
. $config
```

(Recall that using . means that any variables set in the secondary file affect the current shell, not a subshell. The source command does the same thing as the . command.)

Here's our script to experiment with this feature:

```
#!/bin/bash

config="our.config.file"

sigusr1()
{
  echo "(SIGUSR1: re-reading config file)"
  . $config
}

trap sigusr1 USR1        # catch -USR1 signal

echo "Daemon started. Assigned PID is $$"

. $config                # read it first time

while /usr/bin/true; do
  echo "Target number = $number"
  sleep 5
done

trap - USR1              # reset to be elegant

exit 0
```

We'll start with the configuration file containing number=5, then after 10–15 seconds, change it to number=1. Until we send the actual USR1 signal, however, the script just plugs along without a clue that it has changed:

```
$ ./test2.sh
Daemon started. Assigned PID is 25843
Target number = 5
Target number = 5
Target number = 5
```

Meanwhile, in another window, I've already edited the file, so I type in this command:

```
$ kill -USR1 25843
```

And, here's what happens in the main script window:

```
(SIGUSR1: re-reading config file)
Target number = 1
Target number = 1
```

Cool, eh?

I hope this exploration of signal handling in shell scripts is useful. I actually learned quite a bit about advanced handling as I researched the code here. I'm still a bit stymied about how to reset the output stream after catching a SIGTSTP, but I bet that some sharp *Linux Journal* reader will have an answer.■

Dave Taylor has been hacking shell scripts for a really long time, 30 years. He's the author of the popular *Wicked Cool Shell Scripts* and can be found on Twitter as @DaveTaylor and more generally at www.DaveTaylorOnline.com.