Wizard Boot Camp, Part Five

Jerry Peek

A true wizard doesn't just run processes. A true wizard knows how to communicate with those processes through *signals* to stop, restart, and even kill processes. This may sound like arcane and obscure knowledge, but if you follow along, you'll be managing your processes like an adept in no time.

The first articles in this series of things wizards should know (but books often don't cover) explained some Linux and shell fundamentals, discussed standard input/output and other open files, covered redirection of standard I/O and higher-numbered file descriptors, touched on FIFOs and a few other tips. Last month's column covered `ps` and some of what it can disclose about processes.

This month we'll dig into process control: signals sent to processes and how a shell handles processes, including some details on job control.

Introducing Signals

Since the early days of Unix, users have terminated processes by sending a signal with `kill`. Some signals don't actually "kill" a process: they notify the process (to, say, re-read its configuration file), suspend the process, or restart it. The first argument to `kill` is the signal you want to send; the other arguments are either the job numbers (see below) or PIDs of the process(es) that should get the signal. You can also send a signal to the foreground shell process by typing with key combinations like `CTRL-C` and `CTRL-Z` on your keyboard, as we'll see soon.

An "uncatchable" signal may not kill a process. For instance, a zombie is the remnant of a process that's waiting to exit.

Note that, in general, you can only send signals to your own processes. Only the root user is allowed to send a signal to any process running on the system.

Each signal has both a numeric value and a name. Some signals have the same value and name on almost every system. For instance, signal 15, named *SIGTERM*, is the default signal sent by the `kill` utility. Signal 2, *SIGINT*, is the "interrupt from keyboard" signal that's typically sent by pressing CTRL-C. (Use `stty` to change the default signals sent by various key combinations during your login session.)

The list of signals varies in different versions of Unix and Linux, and some signals are system-specific. See the man page for `signal` and the header file (*signal.h*), for specifics of signals on your system.

Some signals can be "caught" by a process; this can either delay or prevent the signal's default effect. For instance, a process can catch *SIGTERM* and "clean up" by, say, removing its temporary files before terminating itself. As another example, *SIGINT* doesn't terminate `vim` at all; instead, the editor leaves insert mode and cancels any pending command.

SIGHUP, signal 1, originated in the days when people connected to systems over dialup phone lines. *SIGHUP* was sent to any leftover processes when the user disconnected (hung up the phone). If you wanted to leave a process running after you logged out, you could start the process using `nohup`, which caught and ignored *SIGHUP* so that its child processes wouldn't terminate at logout.

Some signals, like *SIGQUIT* (sent by `CTRL-\`), cause a core dump before the process terminates. Don't use these signals unless you want to debug the process.

Certain signals can't be caught; they take effect immediately. The best-known example — signal 9, *SIGKILL* — terminates almost any process. A process can't "clean up" after *SIGKILL*, so you generally won't use it unless you've tried less-severe signals first. For instance, to stop a runaway process, try *SIGTERM* first and wait a few seconds before giving up and sending *SIGKILL*.

Even an "uncatchable" signal may not terminate a process. For instance, a *zombie* is the remnant of a process that's waiting to exit. (A zombie has `Z` in the *ps* STAT column.) A process that's being traced can also be unkillable.

Finally, there's a group of signals that suspend and resume a process. We'll look at those next as we introduce job control.

The Evolution of Job Control

Early Unix systems had fairly simple process control: to signal a process, users found its PID and used *kill*. When running programs from a shell prompt, you either ran the process *in the foreground* by typing the command line and waiting for it to finish or you'd start the process running *in the background* by putting an ampersand (`&`) after the command name and its arguments.

In those days, you typically had only one terminal. Once a process started running in the foreground, your only choices for getting a new shell prompt on your terminal were to wait for the process to finish or to terminate it with a signal. If you knew that a program would take some time to run, you'd think ahead and start it in the background so you could get another shell prompt and do other work in the foreground as the background process ran.

Though this setup was usually a lot better than batch processing (submitting a deck of cards to be run in order by the system operator, for instance) it left a lot to be desired. Eventually window systems came along and let users run as many foreground commands as they wanted to, one per window. In-between time, *job control* was developed to give users more control over their processes.

Job control involved changes to the kernel and the shell. (It was first available in the C shell.) You could send a signal to a *process group*, which is a collection of one or many processes. Also, a new set of signals — *SIGSTOP*, *SIGTSTP*, and *SIGCONT* — told individual processes, or process groups, to suspend and resume.

Job is a term used to describe one or more processes that are running under a shell process. There's at most one foreground job; the shell waits for the process or processes in the job to finish before it outputs another prompt. For example, if you type `cp */somedir`, the shell waits to print another prompt until `cp` finishes copying or you kill `cp` with a signal (for instance, `CTRL-C`).

Job control gives you another choice: suspending the foreground job by typing `CTRL-Z`. This sends *SIGTSTP*, "terminal stop," which can be caught by a process. This lets a text editor, for instance, restore your screen to a reasonable condition before the process is suspended. Once you've stopped the foreground job, you can manipulate the job with job control by using its job number. The first stopped or background job has job number 1 (written `%1`), the second is job 2, and so on. To send the "resume" signal, *SIGCONT*, to the process (es) in a job, you'd type `fg %1` to resume the job in the foreground (so the shell waits for it) or `bg %1` to run in the background. If you only have one job running, the job number is not necessary — so `fg` will return you to the suspended process if only one job is stopped.

Another "suspend" signal, *SIGSTOP*, can't be caught or handled. As we'll see in the next section, this is a good choice for background and daemon processes.

You can also send other signals with `kill`. For instance, `kill%1` sends *SIGTERM* to job 1. Use `kill -9 %2`, `kill -sigkill %2`, or `kill -kill %2` to send *SIGKILL* to job 2.

We won't cover job control in depth here; see your shell's manual page or another Linux reference. The important point is that job control lets you stop processes as a group, resume them in the foreground or background, or terminate them using a single job number.

Suspended Processes Remember

You can think of a process as a collection of attributes that are maintained until the process terminates. As we've discussed earlier, some attributes of a Linux process include its current directory, a list of environment variables, and open files.

The Linux kernel only runs a process when resources like a disk or a network are ready — and during its slices of CPU time. At other times, a process will wait or be suspended.

If a process is suspended — by the Linux kernel or by you at your keyboard — the process keeps its attributes. When the process resumes, it starts from the place it left off. For example, it has the same current directory. It reads and writes at the same points in the same files. This is one reason to suspend a text editor, for instance, instead of quitting and restarting it later.

Simply type CTRL-Z to the foreground job, or `kill -stop %`*n* to stop a background job. (The C and *tcsh* shells have a built-in *stop* command. If you use another shell, making an alias named *stop*, for `kill -stop`, can be handy.)

Once you've stopped the foreground job, you can manipulate the job with job control by using its job number

There are times you'll want to send SIGSTOP to a process that's not running in your shell session background. For instance, if you're a system administrator with a busy system, you might pick a long-running or CPU-intensive process, or a process that seems to be a runaway, and use `kill -STOP` to suspend it. Then you can wait for the system load to go down, or you can start looking for evidence of what might be wrong with the process (by reading its temporary files, talking to the user who started the process, and so on). Use `kill -CONT` to restart the process.

Catching Signals From the Shell

Bourne-type shells like *GNU Bash* have flexible signal handling that lets you catch or ignore signals. The built-in command `trap` does the job. Signal handling is much more limited in *tcsh*. *zsh*, as usual for that "super-shell," has even more sophisticated signal handling. You typically won't want to set a trap in a login shell (a shell that you use interactively from a shell prompt). You'll generally use `trap` from a shell script. The syntax of *trap* is:

```
trap todo signal(s)
```

The *signals* argument (s) are one or more signal numbers (2 for *SIGINT*, 15 for *SIGKILL*, and so on) that `trap` should handle; most shells also accept the signal names. *Do* is the action to take when the *signals* are received, and there are several possibilities:

LISTING ONE: Traptest shell script

```
#!/bin/sh
tmp=/tmp/afile  # dummy file
stat=1          # exit status
echo temp file for $0 > $tmp

trap 'echo ouch!; rm -f $tmp; exit' 2 15
trap 'echo bye; exit $stat' 0

echo "sleep #1..." 1>&2
sleep 5
stat=0
echo "sleep #2..." 1>&2
sleep 5
```

- If *todo* is empty (a pair of quotes, `''` or `""`), the signals will be ignored. (As mentioned earlier, some signals, like SIGKILL, can't be caught or ignored.)

- If *todo* has one or more commands, they'll be executed. Enter the commands as you would on a single line of a shell script: separate multiple commands with semicolons (`;`) or an operator like `&;&`.

- If you omit *todo*, handling for those *signals* is reset to the default.

One common use for *trap* is to clean up temporary files before exiting. *Listing One* shows a sample script that does this.

The script starts by creating a temporary file holding one line of text. Next it stores the script's exit status in the shell variable `$stat`; this value is passed to the `exit` command within the second `trap`.

Near the end of the script, `$stat` is changed to zero to indicate success.

The script has two `trap` s:

- The first catches signals 2 and 15; it echoes "ouch!", removes the temporary file, then calls `exit`. (Without the call to `exit`, the script would keep running after it performs the `trap` commands.)

- The other `trap` for "signal" 0 is executed just before the shell terminates — after the shell script has all been read, or when an explicit `exit` command is used. This `trap` echoes "bye", then exits with the value from `$stat`.

Note that the commands for both `trap` s are inside a pair of single quotes. Single quotes prevent expansion of any shell variables within the quoted text, so `$tmp` and `$stat` are stored literally. The shell variables' values are expanded at the time the `trap` is "sprung." This is important because we want the value of `$stat` to `edit` at the time the shell exits, not at the time that the `trap` is set.

(Enclosing the commands inside double quotes, `""`, would have expanded `$tmp` and `$stat` at the time the trap was set.)

After the `trap` s are set, the script echoes "sleep #1" and pauses for five seconds. Then it sets `$stat` to zero, echoes "sleep #2", and pauses for another five seconds. These pauses give you time to send a signal to the script.

Let's try the script in three different ways. First, we'll type `CTRL-C` during the first `sleep` to spring the first trap. As you can see from the "ouch!" and "bye", both traps were sprung. The script exits with status 1, as the `echo $?` command shows (because shell's `$?` parameter contains the exit status of the previous command):

```
$ ./traptest
sleep #1...
^C
ouch!
bye
$ echo $?
1
```

Next, we'll send signal 2 during the second `sleep`. The script's exit status will be zero because `$stat` had been set to 0:

```
$ ./traptest
sleep #1...
sleep #2...
^C
ouch!
bye
$ echo $?
0
```

Finally, if we let the script run to completion, only the second `trap` is sprung:

```
$ ./traptest
sleep #1...
sleep #2...
bye
$ echo $?
0
```

We'll see another example of a *trap* next month in a simple "daemon" shell script that re-reads its configuration file and continues running when it gets a certain signal.

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.