Wizard Boot Camp, Part Six

Jerry Peek

This month, we'll continue the discussion of Linux processes with a look at two unrelated concepts that are both good to understand. First we'll write a simple daemon process and send signals to it. Then we'll see how to suspend a child shell — which gives some insight into how shells cope with signals sent to them.

What's a Daemon?

A *daemon* is typically a long-running job that doesn't have a controlling terminal (*tty*). It's often started when the system boots — for instance, from one of the files in `/etc/rc*` — and it runs until shut down. It could also run be run from *cron* or on demand.

The Linux Daemon Writing HOWTO has code examples and descriptions for GCC. The Unix Programming FAQ, at http://www.faqs.org/faqs/unix-faq/programmer/faq/ (among other places), has a useful section titled "How do I get my program to act like a daemon?" An easy way to do the most important parts of the job is with the `daemon` system call. It detaches the process from its parent, redirects all standard I/O to */dev/null*, and changes the current directory to the root, */*.

Many daemons have a configuration file that's read when the program starts and again if the daemon catches a signal — often `SIGHUP` (signal 1). This lets you update a daemon while it keeps running.

LISTING ONE: Watchload

```
 1  #!/bin/sh
 2  config=/tmp/watchload.config
 3  myname="${0##*/} (PID $$)"
 4
 5  # Function to set $maxload from $config:
 6  read_config()
 7  {
 8    if [ "$1" == -v ]
 9    then echo "$myname: reading $config file"
10    fi
11    . $config
12  }
13
14  read_config # set $maxloads array
15  trap 'read_config -v' HUP
# On SIGHUP, re-read $config
16
17  # Endless loop until killed by signal 15
(etc.):
18  while :
19  do
20    loads=$(< /proc/loadavg)  # get system
loads
21    load1=${loads%%.*}# get 1-minute load
(integer)
22    if [[ load -ge maxload ]]
23    then echo "$myname: 1-minute load is
$load1 at $(date)"
24    fi
25    sleep 60
26  done
```

Daemon in a Shell Script

One way to make simple daemons is by writing a daemon-like shell script. To simplify, our sample script will send messages to its standard output. (If it were a "real" daemon, it probably wouldn't *have* a standard output — or an associated terminal to show its *stdout*. Instead, it could use `mail` to send email messages or `logger` to send messages to the `syslog` facility.)

Listing One shows a Bash script named `watchload`. Although it's just an example (for instance, its configuration file is in the temporary-file directory */tmp*), it has important features of a more complete daemon:

- A function named `read_config` reads the configuration file. (The `.` operator, also named `source`, reads shell commands from the file as if they were part of the shell script. A more sophisticated daemon might read multiple variables from `$config` — and it should do some error checking.)
- When the `trap` (set in line 15) catches signal 1, it runs `read_config` to reset the value of `$maxload`.
- An endless `while` loop checks the system's one-minute load average every 60 seconds. The `:` (colon) operator simply returns a zero exit status, which is a convenient way to make an endless *while* loop. The loop runs until the shell is killed by an untrapped signal — such as signal 15, the default signal sent by `kill`. Some of the *bash* syntax in *watchload* may be new to you.
- The `${0##*/}` in line 1 edits the contents of `$0` (the pathname that was used to invoke the script) to remove everything from the start of the string to the last slash. The `$$` operator expands into the process ID number of the shell that's running the script.

- Line 20 uses the operator `$(< file)` reads the contents of */proc/loadavg* (more about that next month) into the `$loads` shell variable.

- The `${loads%%.*}` in line 21 removes everything starting from the first dot (the first decimal point) in `$loads`. This is a quick-and-dirty way to get the integer one-minute load average: by truncating a number like `5.67` to simply `5`, and throwing away the rest of the `$loads` string too.

We'll take `watchload` for a spin in *Listing Two*.

First we stop `vim` — which has been editing the *watchload* script — by typing CTRL-Z. As we saw last month, that suspends the editor process and leaves it just where it was. Sometime later, typing `fg %1` resumes editing where we left off.

The configuration file starts with the single line `maxload=5`. When the script reads this shell command (via the `read_config` function), it sets `$maxload` to `5`, which means the script won't output messages until the system's load average is at least 5.

Starting the script running in the background, the parent shell assigns it job number 2 and shows that the child shell's PID is 9864.

After a while, we overwrite the config file with the new value `maxload=0`, then use `kill -hup` to send SIGHUP to job number 2. (We could also have used the signal number and the PID, like `kill -1 9864`.)

The script outputs a message to say that it's re-read the configuration file. Because the load average can't be less than 0 (the value of `$maxload`), the script will begin printing a message every 60 seconds.

Sending signal 15 to the job terminates the endless loop — and the daemon exits.

LISTING TWO: Running the Watchload Script

```
[1]+  Stopped                 vim watchload
$ cat /tmp/watchload.config
maxload=5
$ ./watchload &
[2] 9864
(...time passes...)
$ echo "maxload=0" > /tmp/watchload.config
$ kill -hup %2
watchload (PID 9864): reading
/tmp/watchload.config file
$
watchload (PID 9864): 1-minute load is 3 at Thu
Jan 24 19:26:45 MST 2008
watchload (PID 9864): 1-minute load is 4 at Thu
Jan 24 19:27:45 MST 2008
$ kill %2
[2]-  Terminated              ./watchload
```

SUSPENDING SUBSHELLS VS SEPARATE WINDOWS

Listing Two and its description show a lot about using multiple subshells. Is it clearer to suspend subshells from a single terminal window than to open separate windows on your deskcop? That's for you to decide, of course. The concepts are worth understanding, at least.

By the way, if you do open separate windows, here's a tip. To replace the login shell in a window, use exec followed by the shell name. (We saw exec in the November 2007 issue, in part 3 of this series, which is available online.)

When you pass a program name to the shells' built-in exec command, that program replaces the current shell. (In this way, it's similar to the exec system call.)

For instance, if you want a Z Shell running in your terminal window, type exec zsh at a shell prompt. The previous shell is replaced, but the process PID stays the same. When you type *exit*, the *zsh* process terminates and the window closes.

Suspending a Subshell

In the previous section, we stopped the `vim` editor. That leaves its process in a steady state until, sometime later, it's put into the foreground again with `fg`.

Just as a shell runs `vim` in a subprocess, a shell can run another shell in a subprocess.

Why would you want to do this? Examples of child shells include:

- Using another shell, with features you want to have sometime, on your current login session, as yourself. For instance, you might like the powerful features of the Z shell in some cases, but not want (or not be allowed to) run it as your login shell (as the shell that starts when you log into a system and/or when you open a new X Window).

- An `ssh` session that starts an interactive shell on another host. (The subprocess is actually the `ssh` on your local host, but the effect is to suspend the remote shell.)

- An `su` command that runs a shell as another user. (After it checks your password, if needed, the `su` subprocess replaces itself with a shell running under the other user's effective UID.)

You can start the child shell, change its current directory, do whatever else you need to do — and, when you want to go back to your login shell, suspend the child shell.

The child shells command history, current directory, and all of its other attributes, will wait until you bring the child

See Power Tools, *pg. 52*

Power Tools, *from pg. 13*

shell back into the foreground. This is always extremely useful for testing.

Shells ignore SIGTSTP, but you can stop a shell with its built-in command `suspend`. A few shells, like `ash` and `dash`, don't have `suspend`. But, as we'll see, you can send `SIGSTOP` to those shells.

Note that you can't use `suspend` to stop a login shell. If you did — or if you send SIGSTOP to a login shell — that shell session will freeze, and you can only un-freeze it by sending SIGCONT to it from another shell.

LISTING THREE: Suspending Subshells

```
bash$ jobs
bash$ pwd
/home/jpeek
bash$ tcsh
tcsh> cd foo
tcsh> pwd
/home/jpeek/foo
tcsh> suspend
[ 1]+ Stopped tcsh
bash$ zsh
PS1='zsh%% '
zsh% cd bar
zsh% pwd
/home/jpeek/bar
zsh% suspend
[ 2]+ Stopped zsh
bash$ pwd
/home/jpeek
bash$ dash
$ suspend
dash: suspend: not found
$ kill- STOP $$
[ 3]+ Stopped dash
bash$ su web
Password:
<web@myhost> (1) ->suspend
[ 4]+ Stopped su web
bash$ ssh- 2C other@remhost
other@remhost's password:
Last login: Fri Nov 30 18:45:41 2007...
NetBSD 3.99.5...
other@remhost$
other@remhost$~ ^Z[ suspend ssh]
[ 5]+ Stopped ssh- 2C other@remhost
bash$ jobs- l
[ 1] 27125 Stopped tcsh
[ 2] 27144 Stopped zsh
[ 3] 27165 Stopped dash
[ 4]+ 27182 Stopped su web
[ 5]- 27204 Stopped ssh- 2C other@remhost
bash$%
su web
<web@myhost> (2) ->
```

The parent shell in *Listing Three* is `bash`, with the prompt `bash$`. We start by showing that there are no child processes in the shell's job table and that the current directory is *~jpeek*. Typing `tcsh` starts a child process that gives a `tcsh>` prompt. We change the child shell's directory to a subdirectory, then suspend the child.

Back in the parent shell, we do the same thing with `zsh`.

Next, we start a simple POSIX shell, `dash`, that doesn't have a built-in `suspend` command. That doesn't keep us from stopping the shell, though: passing the child shell's PID to `kill-STOP` sends the uncatchable SIGSTOP to `dash`. As in the first two cases, the parent `bash` sees that the child has stopped, prints a message to tell you the job number and PID, and prints a new `bash$` prompt.

And two more examples: changing to the user *web* using `su`, and starting a shell on a remote host with `ssh`. Because `su` starts a child shell, `suspend` does the job. However, the `ssh` process running on the local host isn't a shell, so we need a different method here. The `ssh` escape sequence of a newline character, which you get by pressing RETURN, followed immediately by a tilde character (`~`) and the control sequence CTRL-Z, tells `ssh` to suspend itself. (The network session to the remote host, and the remote shell, wait for the `ssh` process to resume.)

If you'd started an `ssh` session from `remhost` to yet another remote host, you would type two tildes (`~~^Z`) to suspend that session and return to a prompt on `remhost`.

Why? Because `~~`, when followed by a CTRL-Z character, escapes the second tilde — which sends a single tilde from your local host to the ssh subprocess running on `remhost`.

We take a look at the parent shell's job table. The `jobs` option `-l` (lower-case "L") shows the child processes' PIDs — and their current directory, if it's different. Why doesn't `jobs -l` report that jobs 1 and 2 are running in a different directory? That's because `jobs -l` only tracks what the parent shell's current directory was at the time that the child process was started! The parent shell doesn't have any easy way to find out what changes a child process has made to its environment.

Finally, we bring the `su` job into the foreground. Typing `fg %4` is the exact way to do this, but typing `%` by itself is a shortcut to bring the current job into the foreground.

Next Month…

The seventh wizard boot camp column will look into the */proc* pseudo-filesystem. We'll see some of what's there and some ways to use its process information.

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.