Process and job control

FULL WORKLOAD

What is happening on your Linux machine? Various shell commands give you details about system processes and help you control them.

BY HEIKE JURZIK

Whenever you or an application launch a program, the operating system starts one or multiple processes. These processes can enter various states: They can be processed, stopped, restarted, and – under normal circumstances – stopped again. Linux has something similar to access controls for processes: Only the user that started a process can stop, restart, or terminate the process.

The only exception to this rule is the root user, who can control any process on a system. On top of this are processes that run on system user accounts, such as *nobody* or *lp* – again, only root has full access to them. In this issue, I will be looking at tools that help you find out more about, and control, processes.

One Big Family

Processes are never isolated and are always in good company. In fact, they are in a hierarchical structure, with process number 1, *init*, at the top. *init* is the first process that Linux launches after booting. All other processes share this common "ancestor" – *init* starts the operating system's basic programs. *pstree* shows the relationship between "parent" and "child" processes. This tree structure shows you at a glance who is descended from whom (Figure 1).

Additional Output

The tool gives you more detailed output if you set the *-a* flag. This tells pstree to show you, in addition, the parameters with which the programs are running.

If you use a terminal that supports different fonts and bold type, such as Gnome Terminal or KDE's Konsole, you might also want to try the *-h* parameter. This tells pstree to highlight its own process and its ancestors.

If you would like to use this practical feature for other processes, use *-H* with the process ID, and pstree will highlight the specified process and its family tree. Setting the *-p* option tells pstree to output the process ID (PID), and *-u* gives you the user.

All of these parameters can be combined – for example, *pstree -apu*.

Listing Processes with ps

The *ps* command gives you a list of the processes currently running on your system. If you do not specify any command-line parameters, the tool will restrict the list to the current shell. If you are interested in all of your processes, call *ps* with the *x* option (Listing 1).

The tabular output in the shell tells you the following characteristics:

- *PID*: The process identifier, a unique number that you can use to reference a process individually.
- *TTY*: The terminal or console on which the process was started. A question mark indicates that the process is not running on a terminal.
- *STAT*: The process status. The states can be *S* (sleeping), *R* (running), *D* (dead, the process cannot be restarted), or *Z* (zombie, a process that has terminated without correctly returning its return status).
- *TIME*: The computational time used.
- *COMMAND*: The full command with all of its command-line options.

The *ps* command offers a number of additional options for adding more information to the output. For example, *u*

shows the process owner and CPU cycles or memory percentage, and *a* gives you a list of all processes for all users. The *l* option is also practical – this lengthy output gives you additional information on the PPID (parent process identifier) and on the UID (user identification) of the user who launched the process.

To display what can be fairly lengthy command-line parameters in the *COMMAND* column, you might want to set *w* for wider output, and you can use the option multiple times. As shown in Figure 2, you can combine these parameters as needed.

Foreground and Background

In some cases, a program you launch in the shell might run for an extended period of time. Graphical programs that you launch in a terminal window block the shell, preventing any command input. In cases like this, you can run out and grab a coffee or open a second console and carry on working. As an alternative, you can move the process into the background when you start it, or at a later time.

To move the process into the background when you launch it, just add the ampersand character (&) to the command line (Listing 2, line 1). The Xpdf window launches, the shell tells you the process ID (5622), and bash can then accept more commands.

Besides the process ID, you can also see the job ID in square brackets. The job ID is allocated as a consecutive number by the shell. If you launch another program in the same session, you will see that bash assigns job ID 2 (Listing 2, line 3). The *jobs* command tells you which jobs are running in the current shell (Listing 2, line 6).

After a program has completed its task, the shell displays the job ID along with a status message (*Done*) and the program name:

```
[3]+  Done    ➥
xpdf article.pdf
```

The job ID is also useful if you need to move a background process into the foreground, or vice versa.

If you launch a program without appending an ampersand, you can press the keyboard shortcut Ctrl + Z to send it to sleep. The shell confirms this action as follows:

```
[1]+  Stopped    ➥
xpdf
```

If you now type *bg* (background), the process will continue to run in the background. The job ID is useful if you have stopped several processes in a shell. The *bg %3* command tells the process with the job ID *3* that it should start working again. In a similar way, the *fg* (foreground) program moves jobs into the foreground. Again, this program might need more details in the form of a job ID following a percent character.

Listing 1: Command

```
01  $ ps x
02   PID TTY      STAT   TIME
    COMMAND
03  3011 ?        Ss     0:00 /
    usr/bin/gnome-session
04  3061 ?        S      0:00 /
    usr/bin/dbus-launch
    --exit-with-session /usr/bin/
    gnome-session
05 [...]
06  3086 ?        Ssl    0:02
    gnome-panel --sm-client-id
    default1
07  3088 ?        Ssl    0:02
    nautilus --no-default-window
    --sm-client-id default2
```

Listing 2: Jobs

```
01 $ xpdf article.pdf &
02 [1] 5622
03 $ audacity &
04 [2] 6559
05 [...]
06 $ jobs
07 [1]   Running   xpdf article.
   pdf &
08 [2]-  Running    audacity &
09 [3]+  Running   sleep 3600 &
```

Detached

The commands I just looked at move processes to the background and optionally let them go on running. If you close the shell in which you launched the program, this also terminates all the active processes.

The *nohup* program gives you a workaround by protecting the process against the shell's HUP signal (see the next section), thus allowing it to continue running after you close the terminal session. In other words, this cuts the ties between the child process and its parent. Simply call *nohup* with the program (and its options):

```
nohup find /scratch3/mp3 -name ➥
"*.ogg" > ogg_liste.txt
```

This approach does not automatically move the process to the background, but the methods I just described will take care of this.

Closing the shell means that you can't communicate with the process – or does it? Even if you do not have a direct terminal connection, you can still control the program using the signals discussed next.

An End to Everything?

Although the name might suggest otherwise, the *kill* program need not be fatal. On the contrary, you use it to send signals to processes, including polite requests to stop working.

As you might expect, non-privileged users are only allowed to talk to their own processes, whereas the root user can send signals to any process.

Instructions

Typing *kill -l* shows you the instructions that *kill* passes to a process. The following are the most relevant ones for your daily work:

- *SIGHUP*: This tells a process to restart immediately after terminating and is often used to tell servers to parse modified configuration files.
- *SIGTERM*: This request to terminate allows the process to clean up.
- *SIGKILL*: This signal forces a process to terminate come what may. But in

Passwords

ps displays the full set of command-line parameters in the *COMMAND* column. Some programs, such as the *wget* download manager, optionally accept passwords for authentication in the shell. The password also appears as a command in the process list; theoretically, any user on the system could sniff sensitive data.

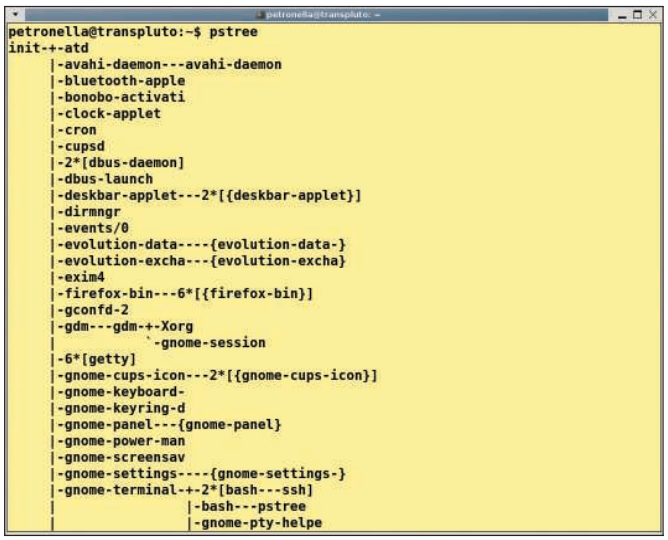

Figure 1: The pstree command shows you process relationships in the shell. All other processes are descended from the first process launched on the system (init).

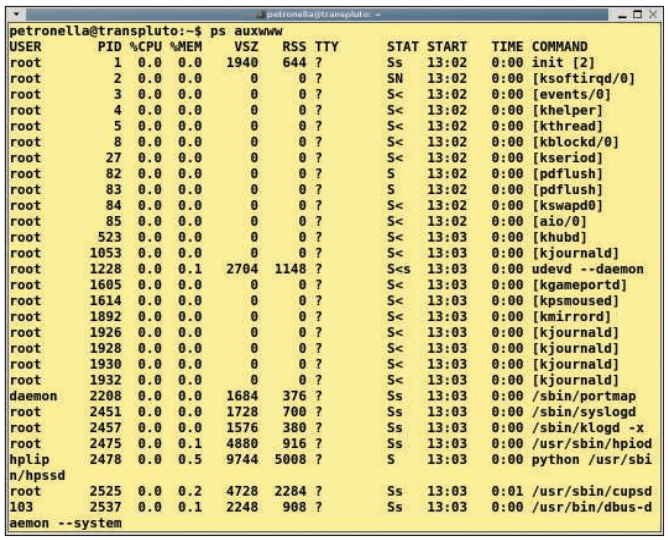

Figure 2: The ps command shows you what is happening on your Linux machine. As you can see here, you can combine parameters as needed.

some cases, it takes more to get rid of the process. After waiting in vain for a timeout, you have no alternative but to reboot.

- *SIGSTOP*: Interrupts the process until you enter *SIGCONT* to continue.

To send a signal to a process, you can enter either the signal name or number followed by the process ID – for example, *kill -19 9201*. Also, you can specify multiple process IDs. If you call *kill* without any parameters but with the PID, it will send the *SIGTERM* signal to the process.

Seek and Ye Shall Find

To find the right process ID, you can run *ps* as described previously. The shell command can be combined with other tools, such as *grep*, in the normal way. For example, you could do this (Listing 3) to find processes with *ssh* in their names.

Besides the SSH server (*sshd*), the list includes all of your SSH connections. To send the same signal to all of these processes, you would normally list the PIDs in the *kill* command line, which can be tricky if the list is too long.

The *killall* gives you a workaround – the tool understands all of the kill signals but expects process names instead of IDs.

The *killall -19 ssh* command sends all your SSH connections to sleep (*SIGSTOP*). If you do not specify the signal, killall assumes you mean *SIGTERM*, just like kill.

Because killall really does remove the processes in one fell swoop, it is a good idea to switch to interactive mode (*-i* option). For each process, the tool prompts you to decide whether to terminate.

Listing 3: grep ssh

```
01 $ ps aux | grep ssh
02 root    2816 ... 0:00 /usr/
   sbin/sshd
03 chicken 3992 ... 0:00 ssh -X
   chicken@asteroid
04 chicken 4249 ... 0:00 ssh
   chicken@nugget
```

More Detective Work

If you are looking for process IDs, a combination of *ps* and *grep* is a good idea, but you can save some typing by running *pgrep* instead.

To find all processes with *ssh* in their names, do the following:

```
$ pgrep ssh
2816
3992
4249
```

Listing 4: -u flag

```
01 # pgrep -lfu petrosilie
02 7682 sleep 4000000000
03 7792 bash
04 [...]
05 # pkill -19 -u petrosilie
```

If you need more context, add the *-l* parameter and pgrep will reveal the names. To discover the full command line, including all arguments, combine *-l* and *-f*:

```
$ pgrep -lf ssh
2816 /usr/sbin/sshd
3992 ssh -X chicken@asteroid
4249 ssh chicken@nugget
```

Hit Squad

The *pkill* command, which is an abbreviation for the Linux "hit squad," understands the same options as *pgrep*, and is run against processes by specifying a signal in the same way as kill:

```
pkill -19 ssh
```

Another practical aspect is that system administrators can target another user's processes by setting the *-u* flag (see Listing 4). To do so, root simply passes in the username as an option.

THE AUTHOR

Heike Jurzik studied German, Computer Science and English at the University of Cologne, Germany. She discovered Linux in 1996 and has been fascinated with the scope of the Linux command line ever since. In her leisure time you might find Heike hanging out at Irish folk sessions or visiting Ireland.