How To Use ps, kill, and nice to Manage Processes in Linux

Authored by: ***ASPHostServer Administrator*** *[asphostserver@gmail.com]*
Saved From: *http://faq.asphosthelpdesk.com/article.php?id=177*

A Linux server, like any other computer you may be familiar with, runs applications. To the computer, these are considered "processes".

While Linux will handle the low-level, behind-the-scenes management in a process's life-cycle, you will need a way of interacting with the operating system to manage it from a higher-level.

In this guide, we will discuss some simple aspects of process management. Linux provides an abundant collection of tools for this purpose.

We will explore these ideas on an Ubuntu 12.04, but any modern Linux distribution will operate in a similar way.

How To View Running Processes in Linux

top

The easiest way to find out what processes are running on your server is to run the `top` command:

```
top
```

```
top - 15:14:40 up 46 min,  1 user,  load average: 0.00, 0.01, 0.05
Tasks:  56 total,   1 running,  55 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1019600k total,   316576k used,   703024k free,     7652k buffers
Swap:        0k total,        0k used,        0k free,   258976k cached
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    1 root      20   0 24188 2120 1300 S  0.0  0.2   0:00.56 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:00.07 ksoftirqd/0
    6 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    7 root      RT   0     0    0    0 S  0.0  0.0   0:00.03 watchdog/0
    8 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 cpuset
    9 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 khelper
   10 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kdevtmpfs
```

The top chunk of information give system statistics, such as system load and the total number of tasks.

You can easily see that there is 1 running process, and 55 processes are sleeping (aka idle/not using CPU resources).

The bottom portion has the running processes and their usage statistics.

htop

An improved version of `top`, called `htop`, is available in the repositories. Install it with this command:

```
sudo apt-get install htop
```

If we run the `htop` command, we will see that there is a more user-friendly display:

```
htop
```

```
  Mem[||||||||||||           49/995MB]     Load average: 0.00 0.03 0.05
  CPU[                           0.0%]     Tasks: 21, 3 thr; 1 running
  Swp[                          0/0MB]     Uptime: 00:58:11
  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 1259 root       20   0 25660  1880  1368 R  0.0  0.2  0:00.06 htop
    1 root       20   0 24188  2120  1300 S  0.0  0.2  0:00.56 /sbin/init
  311 root       20   0 17224   636   440 S  0.0  0.1  0:00.07 upstart-udev-brid
  314 root       20   0 21592  1280   760 S  0.0  0.1  0:00.06 /sbin/udevd --dae
  389 messagebu  20   0 23808   688   444 S  0.0  0.1  0:00.01 dbus-daemon --sys
  407 syslog     20   0  243M  1404  1080 S  0.0  0.1  0:00.02 rsyslogd -c5
  408 syslog     20   0  243M  1404  1080 S  0.0  0.1  0:00.00 rsyslogd -c5
  409 syslog     20   0  243M  1404  1080 S  0.0  0.1  0:00.00 rsyslogd -c5
  406 syslog     20   0  243M  1404  1080 S  0.0  0.1  0:00.04 rsyslogd -c5
  553 root       20   0 15180   400   204 S  0.0  0.0  0:00.01 upstart-socket-br
```

How To Use ps to List Processes

Both `top` and `htop` provide a nice interface to view running processes similar to a graphical task manager.

However, these tools are not always flexible enough to adequately cover all scenarios. A powerful command called `ps` is often the answer to these problems.

When called without arguments, the output can be a bit lack-luster:

```
ps
```

```
  PID TTY          TIME CMD
 1017 pts/0    00:00:00 bash
 1262 pts/0    00:00:00 ps
```

This output shows all of the processes associated with the current user and terminal session. This makes sense because we are only running `bash` and `ps` with this terminal currently.

To get a more complete picture of the processes on this system, we can run the following:

```
ps aux
```

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.2  24188  2120 ?        Ss   14:28   0:00 /sbin/init
```

```
root         2  0.0  0.0      0     0 ?        S    14:28   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    14:28   0:00 [ksoftirqd/0]
root         6  0.0  0.0      0     0 ?        S    14:28   0:00 [migration/0]
root         7  0.0  0.0      0     0 ?        S    14:28   0:00 [watchdog/0]
root         8  0.0  0.0      0     0 ?        S<   14:28   0:00 [cpuset]
root         9  0.0  0.0      0     0 ?        S<   14:28   0:00 [khelper]
. . .
```

These options tell `ps` to show processes owned by all users (regardless of their terminal association) in a user-friendly format.

To see a *tree* view, where hierarchal relationships are illustrated, we can run the command with these options:

```
ps axjf
```

```
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
    0     2     0     0 ?           -1 S        0   0:00 [kthreadd]
    2     3     0     0 ?           -1 S        0   0:00  \_ [ksoftirqd/0]
    2     6     0     0 ?           -1 S        0   0:00  \_ [migration/0]
    2     7     0     0 ?           -1 S        0   0:00  \_ [watchdog/0]
    2     8     0     0 ?           -1 S<       0   0:00  \_ [cpuset]
    2     9     0     0 ?           -1 S<       0   0:00  \_ [khelper]
    2    10     0     0 ?           -1 S        0   0:00  \_ [kdevtmpfs]
    2    11     0     0 ?           -1 S<       0   0:00  \_ [netns]
. . .
```

As you can see, the process `kthreadd` is shown to be a parent of the `ksoftirqd/0` process and the others.

A Note About Process IDs

In Linux and Unix-like systems, each process is assigned a **process ID**, or **PID**. This is how the operating system identifies and keeps track of processes.

A quick way of getting the PID of a process is with the `pgrep` command:

```
pgrep bash
```

```
1017
```

This will simply query the process ID and return it.

The first process spawned at boot, called *init*, is given the PID of "1".

```
pgrep init
```

```
1
```

This process is then responsible for spawning every other process on the system. The later processes are given larger PID numbers.

A process's *parent* is the process that was responsible for spawning it. If a process's parent is killed, then the child processes also die. The parent process's PID is referred to as the **PPID**.

You can see PID and PPID in the column headers in many process management applications, including `top`, `htop` and `ps`.

Any communication between the user and the operating system about processes involves translating between process names and PIDs at some point during the operation. This is why utilities tell you the PID.

How To Send Processes Signals in Linux

All processes in Linux respond to *signals*. Signals are an os-level way of telling programs to terminate or modify their behavior.

How To Send Processes Signals by PID

The most common way of passing signals to a program is with the `kill` command.

As you might expect, the default functionality of this utility is to attempt to kill a process:

```
kill PID_of_target_process
```

This sends the **TERM** signal to the process. The TERM signal tells the process to please terminate. This allows the program to perform clean-up operations and exit smoothly.

If the program is misbehaving and does not exit when given the TERM signal, we can escalate the signal by passing the `KILL` signal:

```
kill -KILL PID_of_target_process
```

This is a special signal that is not sent to the program.

Instead, it is given to the operating system kernel, which shuts down the process. This is used to bypass programs that ignore the signals sent to them.

Each signal has an associated number that can be passed instead of the name. For instance, You can pass "-15" instead of "-TERM", and "-9" instead of "-KILL".

How To Use Signals For Other Purposes

Signals are not only used to shut down programs. They can also be used to perform other actions.

For instance, many daemons will restart when they are given the `HUP`, or hang-up signal. Apache is one program that operates like this.

```
sudo kill -HUP pid_of_apache
```

The above command will cause Apache to reload its configuration file and resume serving content.

You can list all of the signals that are possible to send with kill by typing:

```
kill -l
```

```
1) SIGHUP    2) SIGINT   3) SIGQUIT  4) SIGILL   5) SIGTRAP
 6) SIGABRT  7) SIGBUS   8) SIGFPE   9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
. . .
```

How To Send Processes Signals by Name

Although the conventional way of sending signals is through the use of PIDs, there are also methods of doing this with regular process names.

The `pkill` command works in almost exactly the same way as `kill`, but it operates on a process name instead:

```
pkill -9 ping
```

The above command is the equivalent of:

```
kill -9 `pgrep ping`
```

If you would like to send a signal to every instance of a certain process, you can use the `killall` command:

```
killall firefox
```

The above command will send the TERM signal to every instance of firefox running on the computer.

How To Adjust Process Priorities

Often, you will want to adjust which processes are given priority in a server environment.

Some processes might be considered mission critical for your situation, while others may be executed whenever there might be leftover resources.

Linux controls priority through a value called **niceness**.

High priority tasks are considered less *nice*, because they don't share resources as well. Low priority processes, on the other hand, are *nice* because they insist on only taking minimal resources.

When we ran `top` at the beginning of the article, there was a column marked "NI". This is the *nice* value of the process:

```
top
```

```
 Tasks:  56 total,   1 running,  55 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.3%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
```

```
Mem:   1019600k total,   324496k used,   695104k free,     8512k buffers
Swap:        0k total,        0k used,        0k free,   264812k cached
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1635 root      20   0 17300 1200  920 R  0.3  0.1   0:00.01 top
    1 root      20   0 24188 2120 1300 S  0.0  0.2   0:00.56 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:00.11 ksoftirqd/0
```

Nice values can range between "-19/-20" (highest priority) and "19/20" (lowest priority) depending on the system.

To run a program with a certain nice value, we can use the `nice` command:

```
nice -n 15 command_to_execute
```

This only works when beginning a new program.

To alter the nice value of a program that is already executing, we use a tool called `renice`:

```
renice 0 PID_to_prioritize
```

Note: While nice operates with a command name by necessity, renice operates by calling the process PID

Conclusion

Process management is a topic that is sometimes difficult for new users to grasp because the tools used are different from their graphical counterparts.

However, the ideas are familiar and intuitive, and with a little practice, will become natural. Because processes are involved in everything you do with a computer system, learning how to effectively control them is an essential skill.