Process Control

Slackware systems often run hundreds or thousands of programs, each of which is referred to as a process. Managing these processes is an important part of system administration. So how exactly do we handle all of these seperate processes?

ps

The first step in managing processes is figuring out what processes are currently running. The most popular and powerful tool for this is ***ps***(1). Without any arguments, ***ps*** won't tell you much information. By default, it only tells you what processes are running in your currently active shell. If we want more information, we'll need to look deeper.

```
darkstar:~$ ps
  PID TTY          TIME CMD
12220 pts/4    00:00:00 bash
12236 pts/4    00:00:00 ps
```

Here you can see what processes you are running in your currently active shell or terminal and only some information is included. The PID is the *"Process ID"*; every process is assigned a unique number. The TTY tells you what terminal device the process is attached to. Naturally, CMD is the command that was run. You might be a little confused by TIME though, since it seems to move so slowly. This isn't the amount of real time the process has been running, but rather the amount of CPU time the process has consumed. An idle process uses virtually no CPU time, so this value may not increase quickly.

Viewing only our own processes isn't very much fun, so let's take a look at all the processes on the system with the *-e* argument.

```
darkstar:~$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 migration/0
    4 ?        00:00:00 ksoftirqd/0
    7 ?        00:00:11 events/0
    9 ?        00:00:01 work_on_cpu/0
   11 ?        00:00:00 khelper
  102 ?        00:00:02 kblockd/0
  105 ?        00:01:19 kacpid
  106 ?        00:00:01 kacpi_notify
... many more lines omitted ...
```

The above example uses the standard ***ps*** syntax, but much more information can be discovered if we use BSD syntax. In order to do so, we must use the *aux* argument.

This is distinct from the *-aux* argument, but in most cases the two arguments are equivalent. This is a

decades-old relic. For more information, see the man page for ***ps***.

```
darkstar:~$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   3928   632 ?        Ss   Apr05   0:00 init [3]
root         2  0.0  0.0      0     0 ?        S    Apr05   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    Apr05   0:00
[migration/0]
root         4  0.0  0.0      0     0 ?        S    Apr05   0:00
[ksoftirqd/0]
root         7  0.0  0.0      0     0 ?        S    Apr05   0:11 [events/0]
root         9  0.0  0.0      0     0 ?        S    Apr05   0:01
[work_on_cpu/0]
root        11  0.0  0.0      0     0 ?        S    Apr05   0:00 [khelper]
... many more lines omitted ....
```

As you can see, BSD syntax offers much more information, including what user controls the process and what percentage of RAM and CPU the process is consuming when ***ps*** is run.

To accomplish bits of this, on a per process basis, ***ps*** allows one or more process IDs (PIDs) to be provided in the command line, and has the '-o' flag to show a particular attribute of the PID.

```
darkstar:~$ ps -o cmd -o etime $$
CMD                          ELAPSED
/bin/bash                      12:22
```

What this is displaying, is the PID's command name (cmd), and its elapsed time (etime). The PID in this example, is a shell variable for the PID of the current shell. So you can see, in this example, the shell process has existed for 12 minutes, 22 seconds.

Using the ***pgrep***(1) command, this can get more automatable.

```
darkstar:~$ ps -o cmd -o rss -o vsz $(pgrep httpd)
CMD                           RSS    VSZ
/usr/sbin/httpd -k restart  33456  84816
/usr/sbin/httpd -k restart  33460  84716
/usr/sbin/httpd -k restart  33588  84472
/usr/sbin/httpd -k restart  30424  81608
/usr/sbin/httpd -k restart  33104  84900
/usr/sbin/httpd -k restart  33268  85112
/usr/sbin/httpd -k restart  30640  82724
/usr/sbin/httpd -k restart  15168  67396
/usr/sbin/httpd -k restart  33180  84416
/usr/sbin/httpd -k restart  33396  84592
/usr/sbin/httpd -k restart  32804  84232
```

In this example, a subshell execution, using ***pgrep***, is returning the PIDs of any process, whose command name includes 'httpd'. Then ***ps*** displaying the command name, resident memory size, and virtual memory size.

Finally, ***ps*** can also create a process tree. This shows you which processes have children processes.

Ending the parent of a child process also ends the child. We do this with the *-ejH* argument.

```
darkstar:~$ ps -ejH
... many lines omitted ...
 3660  3660  3660 tty1     00:00:00   bash
29947 29947  3660 tty1     00:00:00     startx
29963 29947  3660 tty1     00:00:00       xinit
29964 29964 29964 tty7     00:27:11         X
29972 29972  3660 tty1     00:00:00         sh
29977 29972  3660 tty1     00:00:05           xscreensaver
29988 29972  3660 tty1     00:00:04           xfce4-session
29997 29972  3660 tty1     00:00:16             xfwm4
29999 29972  3660 tty1     00:00:02             Thunar
... many more lines omitted ...
```

As you can see, ***ps***(1) is an incredibly powerful tool for determining not only what processes are currently active on your system, but also for learning lots of important information about them.

As is the case with many of the applications, there is often several tools for the job. Similar to the ***ps -ejH*** output, but more terse, is ***pstree***(1). It displays the process tree, a bit more visually.

```
darkstar:~$ pstree
init-+-atd
     |-crond
     |-dbus-daemon
     |-httpd---10*[httpd]
     |-inetd
     |-klogd
     |-mysqld_safe---mysqld---8*[{mysqld}]
     |-screen-+-4*[bash]
     |        |-bash---pstree
     |        |-2*[bash---ssh]
     |        `-bash---irssi
     |-2*[sendmail]
     |-ssh-agent
     |-sshd---sshd---sshd---bash---screen
     `-syslogd
```

kill and killall

Managing processes isn't only about knowing which ones are running, but also about communicating with them to change their behavior. The most common way of managing a program is to terminate it. Thus, the tool for the job is named ***kill***(1). Despite the name, ***kill*** doesn't actually terminate processes, but sends signals to them. The most common signal is a SIGTERM, which tells the process to finish up what it is doing and terminate. There are a variety of other signals that can be sent, but the three most common are SIGTERM, SIGHUP, and SIGKILL.

What a process does when it receives a signal varies. Most programs will terminate (or attempt to terminate) whenever they receive any signal, but there are a few important differences. For starters,

the SIGTERM signal informs the process that it should terminate itself at its earliest convenience. This gives the process time to finish up any important activities, such as writing information to the disk, before it closes. In contrast, the SIGKILL signal tells the process to terminate itself immediately, no questions asked. This is most useful for killing processes that are not responding and is sometimes called the *"silver bullet"*. Some processes (particularly daemons) capture the SIGHUP signal and reload their configuration files whenever they receive it.

In order to signal a process, we first need to know it's PID. You can get this easily with ***ps*** as we discused. In order to send different signals to a running process, you simply pass the signal number and *-s* as an argument. The *-l* argument lists all the signals you can choose and their number. You can also send signals by their name with *-s*.

```
darkstar:~$ 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    16) SIGSTKFLT
... many more lines omitted ...
darkstar:~$ kill 1234 # SIGTERM
darkstar:~$ kill -s 9 1234 # SIGKILL
darkstar:~$ kill -s 1 1234 # SIGHUP
darkstar:~$ kill -s HUP 1234 # SIGHUP
```

Sometimes you may wish to terminate all running processes with a certain name. You can kill processes by name with ***killall***(1). Just pass the same arguments to ***killall*** that you would pass to ***kill***.

```
darkstar:~$ killall bash # SIGTERM
darkstar:~$ killall -s 9 bash # SIGKILL
darkstar:~$ killall -s 1 bash # SIGHUP
darkstar:~$ killall -s HUP bash # SIGHUP
```

top

So far we've learned how to look at the active processes for a moment in time, but what if we want to monitor them for an extended period? ***top***(1) allows us to do just that. It displays an ordered list of the processes on your system, along with vital information about them, and updates periodically. By default, processes are ordered by their CPU percentage and updates occur every three seconds.

```
darkstar:~$ top
top - 16:44:15 up 26 days,  5:53,  5 users,  load average: 0.08, 0.03, 0.03
Tasks: 122 total,   1 running, 119 sleeping,   0 stopped,   2 zombie
Cpu(s):  3.4%us,  0.7%sy,  0.0%ni, 95.5%id,  0.1%wa,  0.0%hi,  0.2%si,
0.0%st
Mem:   3058360k total,  2853780k used,   204580k free,   154956k buffers
Swap:        0k total,        0k used,        0k free,  2082652k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    1 root      20   0  3928  632  544 S    0  0.0   0:00.99 init
    2 root      15  -5     0    0    0 S    0  0.0   0:00.00 kthreadd
```

```
    3 root      RT  -5     0    0    0 S    0  0.0   0:00.82 migration/0
    4 root      15  -5     0    0    0 S    0  0.0   0:00.01 ksoftirqd/0
    7 root      15  -5     0    0    0 S    0  0.0   0:11.22 events/0
    9 root      15  -5     0    0    0 S    0  0.0   0:01.19 work_on_cpu/0
   11 root      15  -5     0    0    0 S    0  0.0   0:00.01 khelper
  102 root      15  -5     0    0    0 S    0  0.0   0:02.04 kblockd/0
  105 root      15  -5     0    0    0 S    0  0.0   1:20.08 kacpid
  106 root      15  -5     0    0    0 S    0  0.0   0:01.92 kacpi_notify
  175 root      15  -5     0    0    0 S    0  0.0   0:00.00 ata/0
  177 root      15  -5     0    0    0 S    0  0.0   0:00.00 ata_aux
  178 root      15  -5     0    0    0 S    0  0.0   0:00.00 ksuspend_usbd
  184 root      15  -5     0    0    0 S    0  0.0   0:00.02 khubd
  187 root      15  -5     0    0    0 S    0  0.0   0:00.00 kseriod
  242 root      20   0     0    0    0 S    0  0.0   0:03.37 pdflush
  243 root      15  -5     0    0    0 S    0  0.0   0:02.65 kswapd0
```

The man page has helpful details on how to interact with ***top*** such as changing its delay interval, the order processes are displayed, and even how to terminate processes right from within ***top*** itself.

cron

Ok, so we've learned many different ways of viewing the active processes on our system and means of signalling them, but what if we want to run a process periodically? Fortunately, Slackware includes just the thing, ***crond***(8). cron runs processes for every user on the schedule that user demands. This makes it very useful for processes that need to be run periodically, but don't require full daemonization, such as backup scripts. Every user gets their own entry in the cron database, so non-root users can periodically run processes too.

In order to run programs from cron, you'll need to use the ***crontab***(1). The man page lists a variety of ways to do this, but the most common method is to pass the *-e* argument. This will lock the user's entry in the cron database (to prevent it from being overwritten by another program), then open that entry with whatever text editor is specified by the VISUAL environment variable. On Slackware systems, this is typically the ***vi*** editor. You may need to refer to the chapter on ***vi*** before continuing.

The cron database entries may seem a little archaic at first, but they are highly flexible. Each uncommented line is processed by ***crond*** and the command specified is run if all the time conditions match.

```
darkstar:~$ crontab -e
# Keep current with slackware
30 02 * * * /usr/local/bin/rsync-slackware64.sh 1>/dev/null 2>&1
```

As mentioned before, the syntax for cron entries is a little difficult to understand at first, so let's look at each part individually. From left to right, the different sections are: Minute, Hour, Day, Month, Week Day, and Command. Any asterisk * entry matches every minute, hour, day, and so on. So from the example above, the command is *"/usr/local/bin/rsync-slackware64.sh 1>/dev/null 2>&1"*, and it runs every weekday or every week of every month at 2:30 a.m.

crond will also e-mail the local user with any output the command generates. For this reason, many

tasks have their output redirected to `/dev/null`, a special device file that immediately discards everything it receives. In order to make it easier for you to remember these rules, you might wish to paste the following commented text at the top of your own cron entries.

```
# Redirect everything to /dev/null with:
#   1>/dev/null 2>&1
#
# MIN HOUR DAY MONTH WEEKDAY COMMAND
```

By default, Slackware includes a number of entries and comments in root's crontab. These entries make it easier to setup periodic system tasks by creating a number of directories in `/etc` corresponding to how often the tasks should run. Any script placed within these directories will be run hourly, daily, weekly, or monthly. The names should be self-explanatory: `/etc/cron.hourly`, `/etc/cron.daily`, `/etc/cron.weekly`, and `/etc/cron.monthly`.

Chapter Navigation

Previous Chapter: The Bourne Again Shell

Next Chapter: The X Window System

Sources

* Original source: http://www.slackbook.org/beta
* Originally written by Alan Hicks, Chris Lumens, David Cantrell, Logan Johnson

slackbook, process control, ps, kill, killall, top, cron