10

PROCESSES

Modern operating systems are usually *multitasking*, meaning they create the illusion of doing more than one thing at once by rapidly switching from one executing program to another. The kernel manages this through the use of *processes*. Processes are how Linux organizes the different programs waiting for their turn at the CPU.

Sometimes a computer will become sluggish or an application will stop responding. In this chapter, we will look at some of the tools available at the command line that let us examine what programs are doing and how to terminate processes that are misbehaving.

This chapter will introduce the following commands:

`ps` Report a snapshot of current processes

`top` Display tasks

`jobs` List active jobs

`bg` Place a job in the background

`fg` Place a job in the foreground

`kill` Send a signal to a process

`killall` Kill processes by name

`shutdown` Shut down or reboot the system

How a Process Works

When a system starts up, the kernel initiates a few of its own activities as processes and launches a program called `init`. `init`, in turn, runs a series of shell scripts (located in */etc*) called *init scripts,* which start all the system services. Many of these services are implemented as *daemon programs,* programs that just sit in the background and do their thing without having any user interface. So, even if we are not logged in, the system is at least a little busy performing routine stuff.

The fact that a program can launch other programs is expressed in the process scheme as a *parent process* producing a *child process.*

The kernel maintains information about each process to help keep things organized. For example, each process is assigned a number called a *process ID* (*PID*). PIDs are assigned in ascending order, with `init` always getting PID 1. The kernel also keeps track of the memory assigned to each process, as well as the processes' readiness to resume execution. Like files, processes also have owners and user IDs, effective user IDs, and so on.

Viewing Processes

The most commonly used command to view processes (there are several) is `ps`. The `ps` program has a lot of options, but in its simplest form it is used like this:

```
[me@linuxbox ~]$ ps
  PID TTY          TIME CMD
 5198 pts/1    00:00:00 bash
10129 pts/1    00:00:00 ps
```

The result in this example lists two processes, process 5198 and process 10129, which are `bash` and `ps`, respectively. As we can see, by default, `ps` doesn't show us very much, just the processes associated with the current terminal session. To see more, we need to add some options; however, before we do that, let's look at the other fields produced by `ps`. TTY is short for "teletype" and refers to the *controlling terminal* for the process. Unix is showing its age here. The TIME field is the amount of CPU time consumed by the process. As we can see, neither process makes the computer work very hard.

If we add an option, we can get a bigger picture of what the system is doing.

```
[me@linuxbox ~]$ ps x
  PID TTY      STAT   TIME COMMAND
 2799 ?        Ssl    0:00 /usr/libexec/bonobo-activation-server –ac
 2820 ?        Sl     0:01 /usr/libexec/evolution-data-server-1.10 --
15647 ?        Ss     0:00 /bin/sh /usr/bin/startkde
15751 ?        Ss     0:00 /usr/bin/ssh-agent /usr/bin/dbus-launch --
15754 ?        S      0:00 /usr/bin/dbus-launch --exit-with-session
15755 ?        Ss     0:01 /bin/dbus-daemon --fork --print-pid 4 –pr
15774 ?        Ss     0:02 /usr/bin/gpg-agent -s –daemon
15793 ?        S      0:00 start_kdeinit --new-startup +kcminit_start
15794 ?        Ss     0:00 kdeinit Running...
15797 ?        S      0:00 dcopserver –nosid
--snip--
```

Adding the `x` option (note that there is no leading dash) tells `ps` to show all of our processes regardless of what terminal (if any) they are controlled by. The presence of a `?` in the TTY column indicates no controlling terminal. Using this option, we see a list of every process that we own.

Because the system is running a lot of processes, `ps` produces a long list. It is often helpful to pipe the output from `ps` into `less` for easier viewing. Some option combinations also produce long lines of output, so maximizing the terminal emulator window might be a good idea, too.

A new column titled sᴛᴀᴛ has been added to the output. sᴛᴀᴛ is short for *state* and reveals the current status of the process, as shown in Table 10-1.

Table 10-1: Process States

State	Meaning
R	Running. This means the process is running or ready to run.
S	Sleeping. The process is not running; rather, it is waiting for an event, such as a keystroke or network packet.
D	Uninterruptible sleep. The process is waiting for I/O such as a disk drive.
T	Stopped. The process has been instructed to stop. You'll learn more about this later in the chapter.
Z	A defunct or "zombie" process. This is a child process that has terminated but has not been cleaned up by its parent.
<	A high-priority process. It's possible to grant more importance to a process, giving it more time on the CPU. This property of a process is called *niceness*. A process with high priority is said to be less *nice* because it's taking more of the CPU's time, which leaves less for everybody else.
N	A low-priority process. A process with low priority (a nice process) will get processor time only after other processes with higher priority have been serviced.

The process state may be followed by other characters. These indicate various exotic process characteristics. See the ᴘs man page for more details.

Another popular set of options is `aux` (without a leading dash). This gives us even more information.

```
[me@linuxbox ~]$ ps aux
USER   PID %CPU %MEM  VSZ  RSS TTY    STAT START  TIME
COMMAND
root    1 0.0 0.0 2136  644 ?     Ss  Mar05  0:31 init
root    2 0.0 0.0   0    0 ?     S<  Mar05  0:00 [kt]
root    3 0.0 0.0   0    0 ?     S<  Mar05  0:00 [mi]
root    4 0.0 0.0   0    0 ?     S<  Mar05  0:00 [ks]
root    5 0.0 0.0   0    0 ?     S<  Mar05  0:06 [wa]
root    6 0.0 0.0   0    0 ?     S<  Mar05  0:36 [ev]
root    7 0.0 0.0   0    0 ?     S<  Mar05  0:00 [kh]
--snip--
```

This set of options displays the processes belonging to every user. Using the options without the leading dash invokes the command with "BSD-style" behavior. The Linux version of `ps` can emulate the behavior of the `ps` program found in several different Unix implementations. With these options, we get the additional columns shown in Table 10-2.

Table 10-2: BSD Style `ps` Column Headers

Header	Meaning
USER	User ID. This is the owner of the process.
%CPU	CPU usage in percent.
%MEM	Memory usage in percent.
VSZ	Virtual memory size.
RSS	Resident set size. This is the amount of physical memory (RAM) the process is using in kilobytes.

Header	Meaning
START	Time when the process started. For values over 24 hours, a date is used.

Viewing Processes Dynamically with top

While the ps command can reveal a lot about what the machine is doing, it provides only a snapshot of the machine's state at the moment the ps command is executed. To see a more dynamic view of the machine's activity, we use the top command.

```
[me@linuxbox ~]$ top
```

The top program displays a continuously updating (by default, every three seconds) display of the system processes listed in order of process activity. The name *top* comes from the fact that the top program is used to see the "top" processes on the system. The top display consists of two parts: a system summary at the top of the display, followed by a table of processes sorted by CPU activity.

```
top - 14:59:20 up  6:30,  2 users,  load average: 0.07, 0.02, 0.00
Tasks: 109 total,   1 running, 106 sleeping,   0 stopped,   2 zombie
Cpu(s):  0.7%us,  1.0%sy,  0.0%ni, 98.3%id,  0.0%wa,  0.0%hi,  0.0%si
Mem:   319496k total,  314860k used,    4636k free,   19392k buff
Swap:  875500k total,  149128k used,  726372k free,  114676k cach


  PID USER       PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 6244 me         39  19 31752 3124 2188 S  6.3  1.0  16:24.42 trackerd
11071 me         20   0  2304 1092  840 R  1.3  0.3   0:00.14 top
 6180 me         20   0  2700 1100  772 S  0.7  0.3   0:03.66 dbus-dae
 6321 me         20   0 20944 7248 6560 S  0.7  2.3   2:51.38 multiloa
 4955 root       20   0  104m 9668 5776 S  0.3  3.0   2:19.39 Xorg
    1 root       20   0  2976  528  476 S  0.0  0.2   0:03.14 init
    2 root       15  -5     0    0    0 S  0.0  0.0   0:00.00 kthreadd
```

```
  3 root    RT -5   0  0   0 S 0.0 0.0  0:00.00 migratio
  4 root    15 -5   0  0   0 S 0.0 0.0  0:00.72 ksoftirq
  5 root    RT -5   0  0   0 S 0.0 0.0  0:00.04 watchdog
  6 root    15 -5   0  0   0 S 0.0 0.0  0:00.42 events/0
  7 root    15 -5   0  0   0 S 0.0 0.0  0:00.06 khelper
 41 root    15 -5   0  0   0 S 0.0 0.0  0:01.08 kblockd/
 67 root    15 -5   0  0   0 S 0.0 0.0  0:00.00 kseriod
114 root    20  0   0  0   0 S 0.0 0.0  0:01.62 pdflush
116 root    15 -5   0  0   0 S 0.0 0.0  0:02.44 kswapd0
```

The system summary contains a lot of good stuff. Table 10-3 provides a rundown.

Table 10-3: top Information Fields

Row	Field	Meaning
1	top	This is the name of the program.
	14:59:20	This is the current time of day.
	up 6:30	This is called *uptime*. It is the amount of time since the machine was last booted. In this example, the system has been up for six and a half hours.
	2 users	There are two users logged in.
	load average:	*Load average* refers to the number of processes that are waiting to run; that is, the number of processes that are in a runnable state and are sharing the CPU. Three values are shown, each for a different period of time. The first is the average for the last 60 seconds, the next the previous 5 minutes, and finally the previous 15 minutes. Values less than 1.0 indicate that the machine is not busy.

Row	Field	Meaning
2	Tasks:	This summarizes the number of processes and their various process states.
3	Cpu(s):	This row describes the character of the activities that the CPU is performing.
	0.7%us	0.7 percent of the CPU is being used for *user processes*. This means processes outside the kernel.
	1.0%sy	1.0 percent of the CPU is being used for *system* (kernel) processes.
	0.0%ni	0.0 percent of the CPU is being used by "nice" (low-priority) processes.
	98.3%id	98.3 percent of the CPU is idle.
	0.0%wa	0.0 percent of the CPU is waiting for I/O.
4	Mem:	This shows how physical RAM is being used.
5	Swap:	This shows how swap space (virtual memory) is being used.

The top program accepts a number of keyboard commands. The two most interesting are h, which displays the program's help screen, and q, which quits top.

Both major desktop environments provide graphical applications that display information similar to top (in much the same way that Task Manager in Windows works), but top is better than the graphical versions because it is faster and it consumes far fewer system resources. After all,

our system monitor program shouldn't be the source of the system slow-down that we are trying to track.

Controlling Processes

Now that we can see and monitor processes, let's gain some control over them. For our experiments, we're going to use a little program called `xlogo` as our guinea pig. The `xlogo` program is a sample program supplied with the X Window System (the underlying engine that makes the graphics on our display go), which simply displays a resizable window containing the X logo. First, we'll get to know our test subject.

```
[me@linuxbox ~]$ xlogo
```

After entering the command, a small window containing the logo should appear somewhere on the screen. On some systems, `xlogo` might print a warning message, but it can be safely ignored.

NOTE

If your system does not include the `xlogo` program, try using `gedit` or `kwrite` instead.

We can verify that `xlogo` is running by resizing its window. If the logo is redrawn in the new size, the program is running.

Notice how our shell prompt has not returned? This is because the shell is waiting for the program to finish, just like all the other programs we have used so far. If we close the `xlogo` window (see Figure 10-1), the prompt returns.

Figure 10-1: The `xlogo` *program*

Interrupting a Process

Let's observe what happens when we run `xlogo` again. First, enter the `xlogo` command and verify that the program is running. Next, return to the terminal window and press CTRL-C.

```
[me@linuxbox ~]$ xlogo
[me@linuxbox ~]$
```

In a terminal, pressing CTRL-C *interrupts* a program. This means we are politely asking the program to terminate. After we pressed CTRL-C, the `xlogo` window closed, and the shell prompt returned.

Many (but not all) command line programs can be interrupted by using this technique.

Putting a Process in the Background

Let's say we wanted to get the shell prompt back without terminating the `xlogo` program. We can do this by placing the program in the background. Think of the terminal as having a *foreground* (with stuff visible on the surface like the shell prompt) and a *background* (with stuff hidden behind the surface). To launch a program so that it is immediately placed in the background, we follow the command with an ampersand (`&`) character.

```
[me@linuxbox ~]$ xlogo &
[1] 28236
[me@linuxbox ~]$
```

After entering the command, the `xlogo` window appeared, and the shell prompt returned, but some funny numbers were printed too. This message is part of a shell feature called *job control*. With this message, the shell is telling us that we have started job number 1 (`[1]`) and that it has PID 28236. If we run `ps`, we can see our process.

```
[me@linuxbox ~]$ ps
  PID TTY       TIME CMD
10603 pts/1   00:00:00 bash
28236 pts/1   00:00:00 xlogo
28239 pts/1   00:00:00 ps
```

The shell's job control facility also gives us a way to list the jobs that have been launched from our terminal. Using the `jobs` command, we can see this list:

```
[me@linuxbox ~]$ jobs
[1]+  Running           xlogo &
```

The results show that we have one job, numbered 1; that it is running; and that the command was `xlogo &`.

Returning a Process to the Foreground

A process in the background is immune from terminal keyboard input, including any attempt to interrupt it with CTRL-C. To return a process to the foreground, use the `fg` command in this way:

```
[me@linuxbox ~]$ jobs
[1]+  Running           xlogo &
[me@linuxbox ~]$ fg %1
xlogo
```

The `fg` command followed by a percent sign and the job number (called a *jobspec*) does the trick. If we have only one background job, the jobspec is optional. To terminate `xlogo`, press CTRL-C.

Stopping (Pausing) a Process

Sometimes we'll want to stop a process without terminating it. This is often done to allow a foreground process to be moved to the background. To stop a foreground process and place it in the background, press CTRL-Z. Let's try it. At the command prompt, type `xlogo`, press ENTER, and then press CTRL-Z.

```
[me@linuxbox ~]$ xlogo
[1]+  Stopped         xlogo
[me@linuxbox ~]$
```

After stopping `xlogo`, we can verify that the program has stopped by attempting to resize the `xlogo` window. We will see that it appears quite dead. We can either continue the program's execution in the foreground, using the `fg` command, or resume the program's execution in the background with the `bg` command.

```
[me@linuxbox ~]$ bg %1
[1]+ xlogo &
[me@linuxbox ~]$
```

As with the `fg` command, the jobspec is optional if there is only one job.

Moving a process from the foreground to the background is handy if we launch a graphical program from the command line but forget to place it in the background by appending the trailing `&`.

Why would we want to launch a graphical program from the command line? There are two reasons.

- The program you want to run might not be listed on the window manager's menus (such as `xlogo`).
- By launching a program from the command line, you might be able to see error messages that would otherwise be invisible if the program were launched graphically. Sometimes, a program will fail to start up when launched from the graphical menu. By launching it from the

command line instead, we may see an error message that will reveal the problem. Also, some graphical programs have interesting and useful command line options.

Signals

The `kill` command is used to "kill" processes. This allows us to terminate programs that need killing (that is, some kind of pausing or termination). Here's an example:

```
[me@linuxbox ~]$ xlogo &
[1] 28401
[me@linuxbox ~]$ kill 28401
[1]+  Terminated              xlogo
```

We first launch `xlogo` in the background. The shell prints the jobspec and the PID of the background process. Next, we use the `kill` command and specify the PID of the process we want to terminate. We could have also specified the process using a jobspec (for example, `%1`) instead of a PID.

While this is all very straightforward, there is more to it than that. The `kill` command doesn't exactly "kill" processes; rather, it sends them *signals*. Signals are one of several ways that the operating system communicates with programs. We have already seen signals in action with the use of CTRL-C and CTRL-Z. When the terminal receives one of these keystrokes, it sends a signal to the program in the foreground. In the case of CTRL-C, a signal called INT (interrupt) is sent; with CTRL-Z, a signal called TSTP (terminal stop) is sent. Programs, in turn, "listen" for signals and may act upon them as they are received. The fact that a program can listen and act upon signals allows a program to do things such as save work in progress when it is sent a termination signal.

Sending Signals to Processes with kill

The `kill` command is used to send signals to programs. Its most common syntax looks like this:

```
kill -signal PID...
```

If no signal is specified on the command line, then the TERM (terminate) signal is sent by default. The kill command is most often used to send the signals listed in Table 10-4.

Table 10-4: Common Signals

Number	Name	Meaning
1	HUP	Hang up. This is a vestige of the good old days when terminals were attached to remote computers with phone lines and modems. The signal is used to indicate to programs that the controlling terminal has "hung up." The effect of this signal can be demonstrated by closing a terminal session. The foreground program running on the terminal will be sent the signal and will terminate. This signal is also used by many daemon programs to cause a reinitialization. This means that when a daemon is sent this signal, it will restart and reread its configuration file. The Apache web server is an example of a daemon that uses the HUP signal in this way.
2	INT	Interrupt. This performs the same function as CTRL-C sent from the terminal. It will usually terminate a program.

Number	Name	Meaning
9	KILL	Kill. This signal is special. Whereas programs may choose to handle signals sent to them in different ways, including ignoring them all together, the KILL signal is never actually sent to the target program. Rather, the kernel immediately terminates the process. When a process is terminated in this manner, it is given no opportunity to "clean up" after itself or save its work. For this reason, the KILL signal should be used only as a last resort when other termination signals fail.
15	TERM	Terminate. This is the default signal sent by the kill command. If a program is still "alive" enough to receive signals, it will terminate.
18	CONT	Continue. This will restore a process after a STOP or TSTP signal. This signal is sent by the bg and fg commands.
19	STOP	Stop. This signal causes a process to pause without terminating. Like the KILL signal, it is not sent to the target process, and thus it cannot be ignored.
20	TSTP	Terminal stop. This is the signal sent by the terminal when CTRL-Z is pressed. Unlike the STOP signal, the TSTP signal is received by the program, but the program may choose to ignore it.

Let's try the kill command.

```
[me@linuxbox ~]$ xlogo &
[1] 13546
[me@linuxbox ~]$ kill -1 13546
[1]+  Hangup              xlogo
```

In this example, we start the xlogo program in the background and then send it a HUP signal with kill. The xlogo program terminates, and the shell indicates that the background process has received a hang-up signal. We may need to press ENTER a couple of times before the message appears. Note that signals may be specified either by number or by name, including the name prefixed with the letters *SIG*.

```
[me@linuxbox ~]$ xlogo &
[1] 13601
[me@linuxbox ~]$ kill -INT 13601
[1]+  Interrupt           xlogo
[me@linuxbox ~]$ xlogo &
[1] 13608
[me@linuxbox ~]$ kill -SIGINT 13608
[1]+  Interrupt           xlogo
```

Repeat the preceding example and try the other signals. Remember, we can also use jobspecs in place of PIDs.

Processes, like files, have owners, and you must be the owner of a process (or the superuser) to send it signals with kill.

In addition to the list of signals covered so far, which are most often used with kill, there are other signals frequently used by the system, as listed in Table 10-5.

Table 10-5: Other Common Signals

Number	Name	Meaning

Number	Name	Meaning
3	QUIT	Quit.
11	SEGV	Segmentation violation. This signal is sent if a program makes illegal use of memory; that is, if it tried to write somewhere it was not allowed to write.
28	WINCH	Window change. This is the signal sent by the system when a window changes size. Some programs, such as `top` and `less`, will respond to this signal by redrawing themselves to fit the new window dimensions.

For the curious, you can display a complete list of signals with the following command:

```
[me@linuxbox ~]$ kill -l
```

Sending Signals to Multiple Processes with killall

It's also possible to send signals to multiple processes matching a specified program or username by using the `killall` command. Here is the syntax:

```
killall [-u user] [-signal] name...
```

To demonstrate, we will start a couple of instances of the `xlogo` program and then terminate them.

```
[me@linuxbox ~]$ xlogo &
[1] 18801
[me@linuxbox ~]$ xlogo &
[2] 18802
```

```
[me@linuxbox ~]$ killall xlogo
[1]- Terminated            xlogo
[2]+ Terminated            xlogo
```

Remember, as with `kill`, you must have superuser privileges to send signals to processes that do not belong to you.

Shutting Down the System

The process of shutting down the system involves the orderly termination of all the processes on the system, as well as performing some vital house-keeping chores (such as syncing all of the mounted file systems) before the system powers off. Four commands can perform this function:

- `halt`
- `poweroff`
- `reboot`
- `shutdown`

The first three are pretty self-explanatory and are generally used without any command line options. Here's an example:

```
[me@linuxbox ~]$ sudo reboot
```

The `shutdown` command is a bit more interesting. With it, we can specify which of the actions to perform (halt, power down, or reboot) and provide a time delay to the shutdown event. Most often it is used like this to halt the system:

```
[me@linuxbox ~]$ sudo shutdown -h now
```

or like this to reboot the system:

```
[me@linuxbox ~]$ sudo shutdown -r now
```

The delay can be specified in a variety of ways. See the `shutdown` man page for details. Once the `shutdown` command is executed, a message is

"broadcast" to all logged-in users warning them of the impending event.

More Process-Related Commands

Because monitoring processes is an important system administration task, there are a lot of commands for it. Table 10-6 lists some to play with.

Table 10-6: Other Process-Related Commands

Command	Description
pstree	Outputs a process list arranged in a tree-like pattern showing the parent-child relationships between processes.
vmstat	Outputs a snapshot of system resource usage including memory, swap, and disk I/O. To see a continuous display, follow the command with a time delay (in seconds) for updates. Here's an example: vmstat 5. Terminate the output with CTRL-C.
xload	A graphical program that draws a graph showing system load over time.
tload	Similar to the xload program but draws the graph in the terminal. Terminate the output with CTRL-C.

Summing Up

Most modern systems feature a mechanism for managing multiple processes. Linux provides a rich set of tools for this purpose. Given that Linux is the world's most deployed server operating system, this makes a lot of sense. Unlike some other systems, however, Linux relies primarily on command line tools for process management. Though there are graphical process tools for Linux, the command line tools are greatly preferred because of their speed and light footprint. While the GUI tools may look

pretty, they often create a lot of system load themselves, which somewhat defeats the purpose.