Job Control

In the previous lesson, we looked at some of the implications of Linux being a multi-user operating system. In this lesson, we will examine the multitasking nature of Linux, and how this is manipulated with the command line interface.

As with any multitasking operating system, Linux executes multiple, simultaneous processes. Well, they appear simultaneous, anyway. Actually, a single processor computer can only execute one process at time but the Linux kernel manages to give each process its turn at the processor and each appears to be running at the same time.

There are several commands that can be used to control processes. They are:

- **ps** - list the processes running on the system
- **kill** - send a signal to one or more processes (usually to "kill" a process)
- **jobs** - an alternate way of listing your own processes
- **bg** - put a process in the background
- **fg** - put a process in the forground

A practical example

While it may seem that this subject is rather obscure, it can be very practical for the average user who mostly works with the graphical user interface. You might not know this, but most (if not all) of the graphical programs can be launched from the command line. Here's an example: there is a small program supplied with the X Windows system called **xload** which displays a graph representing system load. You can excute this program by typing the following:

```
[me@linuxbox me]$ xload
```

Notice that the small **xload** window appears and begins to display the system load graph. Notice also that your prompt did not reappear after the program launched. The shell is waiting for the program to finish before control returns to you. If you close the **xload** window, the **xload** program terminates and the prompt returns.

Putting a program in the background

Now, in order to make life a little easier, we are going to launch the **xload** program again, but this time we will put it in the background so that the prompt will return. To do this, we execute xload like this:

```
[me@linuxbox me]$ xload &
[1] 1223

[me@linuxbox me]$
```

In this case, the prompt returned because the process was put in the background.

Now imagine that you forgot to use the "&" symbol to put the program into the background. There is still hope. You can type control-z and the process will be suspended. The process still exists, but is idle. To resume the process in the background, type the **bg** command (short for background). Here is an example:

```
[me@linuxbox me]$ xload
[2]+ Stopped xload

[me@linuxbox me]$ bg
[2]+ xload &
```

Listing your processes

Now that we have a process in the background, it would be helpful to display a list of the processes we have launched. To do this, we can use either the **jobs** command or the more powerful **ps** command.

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

[me@linuxbox me]$ ps
PID TTY TIME CMD
1211 pts/4 00:00:00 bash
1246 pts/4 00:00:00 xload
1247 pts/4 00:00:00 ps

[me@linuxbox me]$
```

Killing a process

Suppose that you have a program that becomes unresponsive (hmmm...Netscape comes to mind ;-); how do you get rid of it? You use the **kill** command, of course. Let's try this out on xload. First, you need to identify the process you want to kill. You can use either **jobs** or **ps**, to do this. If you use **jobs** you will get back a job number. With **ps**, you are

given a process id (PID). We will do it both ways:

```
[me@linuxbox me]$ xload &
[1] 1292

[me@linuxbox me]$ jobs
[1]+ Running xload &

[me@linuxbox me]$ kill %1

[me@linuxbox me]$ xload &
[2] 1293
[1] Terminated xload

[me@linuxbox me]$ ps
PID TTY TIME CMD
1280 pts/5 00:00:00 bash
1293 pts/5 00:00:00 xload
1294 pts/5 00:00:00 ps

[me@linuxbox me]$ kill 1293
[2]+ Terminated xload

[me@linuxbox me]$
```

A little more about kill

While the **kill** command is used to "kill" processes, its real purpose is to send *signals* to processes. Most of the time the signal is intended to tell the process to go away, but there is more to it than that. Programs (if they are properly written) listen for signals from the operating system and respond to them, most often to allow some graceful method of terminating. For example, a text editor might listen for any signal that indicates that the user is logging off, or that the computer is shutting down. When it receives this signal, it saves the work in progress before it exits. The **kill** command can send a variety of signals to processes. Typing:

```
kill -l
```

will give you a list of the signals it supports. Most are rather obscure, but several are useful to know:

Signal #	*Name*	*Description*
1	*SIGHUP*	Hang up signal. Programs can listen for this signal and act (or not act) upon it.

2	*SIGINT*	Interrupt signal. This signal is given to processes to interrupt them. Programs can process this signal and act upon it. You can also issue this signal directly by typing control-c in the terminal window where the program is running.
15	*SIGTERM*	Termination signal. This signal is given to processes to terminate them. Again, programs can process this signal and act upon it. You can also issue this signal directly by typing control-c in the terminal window where the program is running. This is the default signal sent by the **kill** command if no signal is specified.
9	*SIGKILL*	Kill signal. This signal causes the immediate termination of the process by the Linux kernel. Programs cannot listen for this signal.

Now let's suppose that you have a program that is hopelessly hung (Netscape, maybe) and you want to get rid of it. Here's what you do:

1. Use the **ps** command to get the process id (PID) of the process you want to terminate.
2. Issue a **kill** command for that PID.
3. If the process refuses to terminate (i.e., it is ignoring the signal), send increasingly harsh signals until it does terminate.

```
[me@linuxbox me]$ ps x
PID TTY STAT TIME COMMAND
2931 pts/5 SN 0:00 netscape

[me@linuxbox me]$ kill -SIGTERM 2931

[me@linuxbox me]$ kill -SIGKILL 2931
```

In the example above I used the **kill** command in the formal way. In actual practice, it is more common to do it in the following way since the default signal sent by **kill** is SIGTERM and **kill** can also use the signal number instead of the signal name:

```
[me@linuxbox me]$ kill 2931
```

Then, if the process does not terminate, force it with the SIGKILL signal:

```
[me@linuxbox me]$ kill -9 2931
```

That's it!

This concludes the "Learning the shell" series of lessons. In the next series, "Writing shell scripts," we will look at how to automate tasks with the shell.
