IBM

developerWorks®

Learn Linux, 101: Create, monitor, and kill processes

Keeping your eye on what's going on

Ian Shields
Senior Programmer
IBM

02 February 2010

Learn about process management on Linux®: how to shuffle processes between foreground and background, find out what's running, kill processes, and keep processes running after you've left for the day. You can use the material in this article to study for the LPI 101 exam for Linux system administrator certification, or just to learn for fun.

View more content in this series

Overview

This article grounds you in the basic Linux techniques for process management. Learn to:

- Manage foreground and background jobs
- Start processes that will run after you log out
- Monitor processes
- Select and sort processes for display
- Send signals to processes

This article helps you prepare for Objective 103.5 in Topic 103 of the Linux Professional Institute's Junior Level Administration (LPIC-1) exam 101. The objective has a weight of 4.

Prerequisites

Develop skills on this topic

This content is part of a progressive knowledge path for advancing your skills. See Basics of Linux system administration: Working at the console

To get the most from the articles in this series, you should have a basic knowledge of Linux and a working Linux system on which you can practice the commands covered in this article. Sometimes different versions of a program will format output differently, so your results may not always look exactly like the listings and figures shown here. The results in the examples shown here were obtained on an Ubuntu 9.10 (Karmic Koala) distribution.

© Copyright IBM Corporation 2010

Manage foreground and background jobs

Connect with Ian

Ian is one of our most popular and prolific authors. Browse all of Ian's articles on developerWorks. Check out Ian's profile and connect with him, other authors, and fellow readers in My developerWorks.

If you stop and reflect for a moment, it's pretty obvious that lots of things are running on your computer besides the terminal programs we've been discussing in earlier articles in this series. Indeed, if you are using a graphical desktop, you may have opened more than one terminal window at a time, or perhaps opened a file browser, Internet browser, game, spreadsheet, or other application. Previously our examples have shown commands entered at a terminal window. The command runs and you wait for it to complete before you do anything else. In this article, you will learn how to do more than one thing at a time using your terminal window.

About this series

This series of articles helps you learn Linux system administration tasks. You can also use the material in these articles to prepare for Linux Professional Institute Certification level 1 (LPIC-1) exams.

See our series roadmap for a description of and link to each article in this series. The roadmap is in progress and reflects the latest (April 2009) objectives for the LPIC-1 exams: as we complete articles, we add them to the roadmap. In the meantime, though, you can find earlier versions of similar material, supporting previous LPIC-1 objectives prior to April 2009, in our LPI certification exam prep tutorials.

When you run a command in your terminal window, you are running it in the *foreground*. Most such commands run quickly, but suppose you are running a graphical desktop and would like a digital clock displayed on the desktop. For now, let's ignore the fact that most graphical desktops already have one; we're just using this as an example.

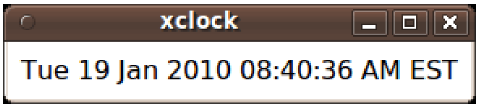
If you have the X Window System installed, you probably also have some utilities such as `xclock` or `xeyes`. You'll probably find these in a package named xorg-x11-apps or x11-apps if you don't have them installed already. Either works for this exercise, but we'll use xclock. The man page explains that you can launch a digital clock on your graphical desktop using the command:

```
xclock -d -update 1
```

The `-update 1` part requests updates every second; otherwise, the clock updates only every minute. So let's run this in a terminal window. You should see a clock like Figure 1, and your terminal window should look like Listing 1. If you don't have xclock or the X Window System, you'll see shortly how to create a poor man's digital clock with your terminal, so you might want to follow along for now and then retry these exercises with that clock.

Note: At the time of writing, there is a bug that affects xclock when desktop effects are enabled. The most noticeable effect is that the title bar does not change, even when given focus. If your xclock examples don't look like the ones in this article, you may want to try switching off desktop effects for a while.

Figure 1. A digital clock with xclock



Listing 1. Starting xclock

```
ian@attic4:~$ xclock -d -update 1
```

Unfortunately, your terminal window no longer has a prompt, so you really need to get control back. Fortunately, the Bash shell has a *suspend* key, Ctrl-z. Pressing this key combination gets you a terminal prompt again as shown in Listing 2.

Listing 2. Suspending xclock with Ctrl-z

```
ian@attic4:~$ xclock -d -update 1
^Z
[1]+  Stopped                 xclock -d -update 1
```

The clock is still on your desktop, but it has stopped running. Suspending it did exactly that. In fact, if you drag another window over part of it, that part of the clock won't even redraw. Notice the terminal output message indicating "[1]+ Stopped". The 1 in this message is a *job number*. You can restart the clock by typing `fg %1`. You could also use the command name or part of it by typing `fg %xclock` or `fg %?clo`. Finally, if you just type `fg` with no parameters, you can restart the most recently stopped job, job 1 in this case. Restarting it with `fg` also brings the job right back to the foreground, and you no longer have a shell prompt. What you need to do is place the job in the *background*; a `bg` command takes the same type of job specification as the `fg` command and does exactly that.

Listing 3 shows how to bring the xclock job back to the foreground and suspend it using two forms of the `fg command.`. You can suspend it again and place it in the background; the clock continues to run while you do other work at your terminal.

Listing 3. Placing xclock in the foreground or background

```
ian@attic4:~$ fg %1
xclock -d -update 1
^Z
[1]+  Stopped                 xclock -d -update 1
ian@attic4:~$ fg %?clo
xclock -d -update 1
^Z
[1]+  Stopped                 xclock -d -update 1
ian@attic4:~$ bg
[1]+ xclock -d -update 1 &
```

Using "&"

You may have noticed that when you placed the xclock job in the background, the message no longer said "Stopped" and that it was terminated with an ampersand (&). In fact, you don't need to suspend the process to place it in the background at all; just append an ampersand to the command and the shell will start the command (or command list) in the background. Let's start an analog clock with a wheat background using this method. You should see a clock like that in Figure 2 and terminal output like Listing 4.

Figure 2. An analog clock with xclock

Listing 4. Starting analog xclock in background with &

```
ian@attic4:~$ xclock -bg wheat -update 1&
[2] 4320
```

Notice that the message is slightly different this time. It represents a job number and a process id (PID). We will cover PIDs and more about status in a moment. For now, let's use the `jobs` command to find out what jobs are running. Add the `-l` option to list PIDs, and you see that job 2 indeed has PID 4320 as shown in Listing 5. Note also that job 2 has a plus sign (+) beside the job number, indicating that it is the *current job*. This job will come to the foreground if no job specification is given with the `fg` command.

Listing 5. Displaying job and process information

```
ian@attic4:~$ jobs -l
[1]-  3878 Running                 xclock -d -update 1 &
[2]+  4320 Running                 xclock -bg wheat -update 1 &
```

Before we address some other issues related to background jobs, let's create a poor man's digital clock. We use the `sleep` command to cause a delay for two seconds, and use the `date` command to print the current date and time. We wrap these commands in a `while` loop with a `do/done` block to create an infinite loop. Finally, we put the whole lot in parentheses to make a command list and put the entire list in the background using an ampersand. You will learn more about how to build more complex commands using loops and scripting in later articles of this series. See our series roadmap for a description of and link to each article in the series.

Listing 6. Poor man's digital clock

```
ian@attic4:~$ (while sleep 2; do date;done)&
[2] 4856
ian@attic4:~$ Tue Jan 19 09:23:30 EST 2010
Tue Jan 19 09:23:32 EST 2010
Tue Jan 19 09:23:34 EST 2010
fTue Jan 19 09:23:36 EST 2010
Tue Jan 19 09:23:38 EST 2010
gTue Jan 19 09:23:40 EST 2010

( while sleep 2; do
    date;
done )
Tue Jan 19 09:23:42 EST 2010
Tue Jan 19 09:23:44 EST 2010
Tue Jan 19 09:23:46 EST 2010
^C
```

Our list is running as job 2 with PID 4856. Every two seconds, the date command runs, and a date and time are printed on the terminal. The input that you type is highlighted. A slow typist will have characters interspersed with several lines of output before a full command can be typed. In fact, notice how the 'f' 'g' that you type in to bring the command list to foreground are a couple of lines apart. When you finally get the `fg` command entered, bash displays the command that is now running in your shell, namely, the command list, which is still happily printing the time every two seconds.

Once you succeed in getting the job into the foreground, you can either terminate (or *kill*) it, or take some other action, In this case, let's use Ctrl-c to terminate our 'clock.'

You may wonder why this job is job 2. With the analog clock terminated, there was only one job running, which was job number 1. So the next available job number was assigned, and our poor man's clock became job 2.

Standard IO and background processes

The output from the `date` command in our previous example is interspersed with echoed characters for the `fg` command that we are trying to type. This raises an interesting issue. What happens to a background process if it needs input from stdin?

The terminal process under which we start a background application is called the *controlling terminal*. Unless redirected elsewhere, the stdout and stderr streams from the background process are directed to the controlling terminal. Similarly, the background task expects input from the controlling terminal, but the controlling terminal has no way of directing any characters you type to the stdin of a background process. In such a case, the Bash shell suspends the process, so that it is no longer executing. You may bring it to the foreground and supply the necessary input. Listing 7 illustrates a simple case where you can put a command list in the background. After a moment, press **Enter** and see the message that the process has stopped. Bring it to the foreground and provide a line of input followed by **Ctrl-d** to signal end of input file. The command list completes and you display the file we created.

Listing 7. Waiting for stdin

```
ian@attic4:~$ (date; cat - > bginput.txt;date)&
[2] 5070
ian@attic4:~$ Tue Jan 19 10:33:13 EST 2010


[2]+  Stopped                 ( date; cat - > bginput.txt; date )
ian@attic4:~$
ian@attic4:~$ fg
( date; cat - > bginput.txt; date )
some textmore text
Tue Jan 19 10:33:31 EST 2010
ian@attic4:~$ cat bginput.txt
some text
more text
```

Run a process after log out

In practice, you probably want to have standard IO streams for background processes redirected to or from a file. There is another related question: what happens to the process if the controlling terminal closes or the user logs off? The answer depends on the shell in use. If the shell sends a SIGHUP (or hangup) signal, then the application is likely to close. We cover signals shortly, but for now we'll consider another way around this problem.

nohup

The `nohup` command is used to start a command that will ignore hangup signals and will append stdout and stderr to a file. The default file is either nohup.out or $HOME/nohup.out. If the file cannot be written, then the command will not run. If you want output to go somewhere else, redirect stdout, or stderr as discussed in the article "Learn Linux 101: Streams, pipes and redirects."

The `nohup` command will not execute a pipeline or a command list. You can save a pipeline or list in a file and then run it using the `sh` (default shell) or the `bash` command. Another article in this series will show you how to make the script file executable, but for now we'll stick to running scripts by using the `sh` or the `bash` command. Listing 8 shows how we might do this for our poor man's digital clock. Needless to say, having the time written to a file isn't particularly useful, and the file will keep growing, so we'll set the clock to update every 30 seconds instead of every second.

Listing 8. Using nohup with a command list in a script

```
ian@attic4:~$ echo "while sleep 30; do date;done">pmc.sh
ian@attic4:~$ nohup sh pmc.sh&
[2] 5485
ian@attic4:~$ nohup: ignoring input and appending output to `nohup.out'

ian@attic4:~$ nohup bash pmc.sh&
[3] 5487
ian@attic4:~$ nohup: ignoring input and appending output to `nohup.out'
```

If we display the contents of nohup.out, we see lines, with each line approximately 30 seconds after the one that is two lines above it, as shown in Listing 9.

Listing 9. Output from nohup processes

```
ian@attic4:~$cat nohup.out
Tue Jan 19 15:01:12 EST 2010
Tue Jan 19 15:01:26 EST 2010
Tue Jan 19 15:01:44 EST 2010
Tue Jan 19 15:01:58 EST 2010
Tue Jan 19 15:02:14 EST 2010
Tue Jan 19 15:02:28 EST 2010
Tue Jan 19 15:02:44 EST 2010
Tue Jan 19 15:02:58 EST 2010
```

Older versions of nohup did not write a status message to the controlling terminal, so if you made a mistake, you might not immediately know. You can see the old behavior if you redirect both stdout and stderr to a file of your own choosing. Suppose you decided that it would be easier to source the command using `.` rather than typing `sh` or `bash`. Listing 10 shows what happens if you use nohup as we did before, but redirect both stdout and stderr. After you enter the command, you see the message indicating that job 4 has started with PID 5853. But press **Enter** again, and you see another message saying that the job has terminated with exit code 126.

Listing 10. Making mistakes with nohup

```
ian@attic4:~$ nohup . pmc.sh >mynohup.out 2>&1 &
[4] 5853
ian@attic4:~$
[4]+  Exit 126                nohup . pmc.sh > mynohup.out 2>&1
```

Listing 11 shows the contents of mynohup.out. Not surprising, really. You use **nohup** to run a command in the background, and you use **source** (**.**) to run read commands from a file and run them in the current shell. The important thing to remember about this is that you may have to press **Enter** to allow the shell to display the background job exit status, and you may have to look at nohup's output file to see what really went wrong.

Listing 11. Hidden message from nohup

```
ian@attic4:~$ cat mynohup.out
nohup: ignoring input
nohup: cannot run command `.': Permission denied
```

Now let's turn our attention to the status of our processes. If you are following along and planning to take a break at this point, please stay around as you now have two jobs that are creating ever larger files in your file system. You can use the `fg` command to bring each, in turn, to foreground, and then use Ctrl-c to terminate it, but if you let them run for a little longer, you'll see other ways to monitor and interact with them.

Monitor processes

Earlier, we had a brief introduction to the `jobs` command and saw how to use it to list the Process IDs (or PIDs) of our jobs.

ps

There is another command, the `ps` command, which we use to display various pieces of process status information. Remember "ps" as an acronym for "process status." The `ps` command accepts zero or more PIDs as arguments and displays the associated process status. If we use the `jobs` command with the `-p` option, the output is simply the PID of the *process group leader* for each job. We'll use this output as arguments to the `ps` command as shown in Listing 12.

Listing 12. Status of background processes

```
ian@attic4:~$ jobs -p
3878
5485
5487
ian@attic4:~$ ps $(jobs -p)
  PID TTY      STAT   TIME COMMAND
 3878 pts/1    S      0:06 xclock -d -update 1
 5485 pts/1    S      0:00 sh pmc.sh
 5487 pts/1    S      0:00 bash pmc.sh
```

If you use `ps` with no options, you see a list of processes that have your terminal as their controlling terminal as shown in Listing 13. Notice that the pmc.sh commands do not show up in this list. You'll see why in a moment.

Listing 13. Displaying status with ps

```
ian@attic4:~$ ps
  PID TTY          TIME CMD
 2643 pts/1    00:00:00 bash
 3878 pts/1    00:00:06 xclock
 5485 pts/1    00:00:00 sh
 5487 pts/1    00:00:00 bash
 6457 pts/1    00:00:00 sleep
 6467 pts/1    00:00:00 sleep
 6468 pts/1    00:00:00 ps
```

Several options, including `-f` (full), `-j` (jobs), and `-l` (long) give control of how much information is displayed. If you do not specify any PIDs, then another useful option is the `--forest` option, which displays the commands in a tree hierarchy, showing which process has which other process as a parent. In particular, you see that the `sleep` commands of the previous listing are children of the scripts you have running in background. If you happened to run the command at a different instant, you might see the `date` command listed in the process status instead, but the odds are very small with this script. We illustrate some of these options in Listing 14.

Listing 14. More status information

```
ian@attic4:~$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
ian       2643  2093  0 Jan18 pts/1    00:00:00 bash
ian       3878  2643  0 09:17 pts/1    00:00:06 xclock -d -update 1
ian       5485  2643  0 15:00 pts/1    00:00:00 sh pmc.sh
ian       5487  2643  0 15:01 pts/1    00:00:00 bash pmc.sh
ian       6635  5485  0 15:41 pts/1    00:00:00 sleep 30
ian       6645  5487  0 15:42 pts/1    00:00:00 sleep 30
ian       6647  2643  0 15:42 pts/1    00:00:00 ps -f
ian@attic4:~$ ps -j --forest
  PID  PGID   SID TTY          TIME CMD
 2643  2643  2643 pts/1    00:00:00 bash
 3878  3878  2643 pts/1    00:00:06  \_ xclock
 5485  5485  2643 pts/1    00:00:00  \_ sh
 6657  5485  2643 pts/1    00:00:00  |   \_ sleep
 5487  5487  2643 pts/1    00:00:00  \_ bash
 6651  5487  2643 pts/1    00:00:00  |   \_ sleep
 6658  6658  2643 pts/1    00:00:00  \_ ps
```

Now that you have some basic tools for monitoring your processes using the `jobs` and `ps` commands, let's take a brief look at two other monitoring commands before moving on to other ways to select and sort processes for display.

free

The `free` command displays the amount of free and used memory in your system. By default the display is in kilobytes, but you can override this using `-b` for bytes, `-k` for kilobytes, `-m` for megabytes, or `-g` for gigabytes. The `-t` option displays a total line, and the `-s` option along with a value refreshes the info with the frequency specified. The number is in seconds but may be a floating point value. Listing 15 shows two examples.

Listing 15. Using the free command

```
ian@attic4:~$ free
             total       used       free     shared    buffers     cached
Mem:       4057976    1543164    2514812          0     198592     613488
-/+ buffers/cache:     731084    3326892
Swap:     10241428          0   10241428
ian@attic4:~$ free -mt
             total       used       free     shared    buffers     cached
Mem:          3962       1506       2456          0        193        599
-/+ buffers/cache:        713       3249
Swap:        10001          0      10001
Total:       13964       1506      12457
```

uptime

The `uptime` command shows you a one-line display that includes the current time, how long the system has been running, how many users are currently logged on, and the system load averages for the past 1, 5, and 15 minutes. Listing 16 shows an example.

Listing 16. Showing uptime information

```
ian@attic4:~$ uptime
 17:41:17 up 20:03,  5 users,  load average: 0.00, 0.00, 0.00
```

Select and sort processes for display

Using ps

The `ps` commands discussed so far only list processes that were started from your terminal session (note the SID, or session id, column in the second example of Listing 14). To see all the processes with controlling terminals, use the `-a` option. The `-x` option displays processes without a controlling terminal, and the `-e` option displays information for **every** process. Listing 17 shows the full format for all the processes with a controlling terminal.

Listing 17. Displaying other processes

```
ian@attic4:~$ ps -af
UID        PID  PPID  C STIME TTY          TIME CMD
ian       3878  2643  0 09:17 pts/1    00:00:06 xclock -d -update 1
ian       5485  2643  0 15:00 pts/1    00:00:00 sh pmc.sh
ian       5487  2643  0 15:01 pts/1    00:00:00 bash pmc.sh
ian       7192  5485  0 16:00 pts/1    00:00:00 sleep 30
ian       7201  5487  0 16:00 pts/1    00:00:00 sleep 30
ian       7202  2095  0 16:00 pts/0    00:00:00 ps -af
```

Note the controlling terminal listed in the TTY column. For this listing, I switched to the terminal window I opened originally (pts/0), so the `ps -af` command is running under pts/0, while the commands created for this article are running under pts/1.

There are many more options for `ps`, including a number that provide significant control over what fields are displayed and how they are displayed. Others provide control over the selection of processes for display, for example, by selecting those processes for a particular user (`-u`) or a particular command (`-C`). In Listing 18, all processes running the `getty` command are listed; we use the `-o` option to specify the columns that will be displayed. We've added the `user` option to the normal list that you get with just plain `ps`, so you can see which user runs `getty`.

Listing 18. Who is running the getty command?

```
ian@attic4:~$ ps -C getty -o user,pid,tty,time,comm
USER       PID TT           TIME COMMAND
root      1192 tty4     00:00:00 getty
root      1196 tty5     00:00:00 getty
root      1209 tty2     00:00:00 getty
root      1219 tty3     00:00:00 getty
root      1229 tty6     00:00:00 getty
root      1731 tty1     00:00:00 getty
```

Sometimes you will want to sort the output by particular fields, and you can do that too using the `--sort` option to specify the sort fields. The default is to sort in ascending order (`+`), but you can also specify descending order (`-`). Listing 19 shows the final `ps` example where all processes are listed using jobs format, and the output is sorted by session id and command name. For the first, we use the default sort order, and for the second, we specify both sorts orders explicitly.

Listing 19. Sorting the output from ps

```
ian@attic4:~$ ps -aj --sort -sid,+comm
  PID  PGID   SID TTY          TIME CMD
 5487  5487  2643 pts/1    00:00:00 bash
 9434  9434  2643 pts/1    00:00:00 ps
 5485  5485  2643 pts/1    00:00:00 sh
 9430  5485  2643 pts/1    00:00:00 sleep
 9433  5487  2643 pts/1    00:00:00 sleep
 3878  3878  2643 pts/1    00:00:10 xclock
 8019  8019  2095 pts/0    00:00:00 man
 8033  8019  2095 pts/0    00:00:00 pager
ian@attic4:~$ ps -aj --sort sid,comm
  PID  PGID   SID TTY          TIME CMD
 8019  8019  2095 pts/0    00:00:00 man
 8033  8019  2095 pts/0    00:00:00 pager
 5487  5487  2643 pts/1    00:00:00 bash
 9435  9435  2643 pts/1    00:00:00 ps
 5485  5485  2643 pts/1    00:00:00 sh
 9430  5485  2643 pts/1    00:00:00 sleep
 9433  5487  2643 pts/1    00:00:00 sleep
 3878  3878  2643 pts/1    00:00:10 xclock
```

As usual, see the man pages for `ps` for full details on the many options and fields you may specify, or get a brief summary by using `ps --help`.

Using top

If you run `ps` several times in a row to see what is changing, you probably need the `top` command instead. It displays a continuously updated process list, along with useful summary information. Listing 20 shows the first few lines of a `top` display. Use the **q** subcommand to quit **top**.

Listing 20. Displaying processes using top

```
top - 16:07:22 up 18:29,  5 users,  load average: 0.03, 0.02, 0.00
Tasks: 170 total,   1 running, 169 sleeping,   0 stopped,   0 zombie
Cpu(s):  2.1%us,  0.5%sy,  0.0%ni, 97.4%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   4057976k total,  1543616k used,  2514360k free,   194648k buffers
Swap: 10241428k total,        0k used, 10241428k free,   613000k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 6820 ian       20   0  506m  78m  26m S    1  2.0   0:23.97 firefox
 1381 root      20   0  634m  40m  18m S    1  1.0   2:06.74 Xorg
 2093 ian       20   0  212m  15m  10m S    1  0.4   0:13.53 gnome-terminal
 6925 ian       20   0 1118m 298m  19m S    1  7.5   1:07.04 java
 6855 ian       20   0 73416  11m 8808 S    1  0.3   0:05.01 npviewer.bin
 7351 ian       20   0 19132 1364  980 R    0  0.0   0:00.07 top
    1 root      20   0 19584 1888 1196 S    0  0.0   0:00.74 init
    2 root      15  -5     0    0    0 S    0  0.0   0:00.01 kthreadd
```

The `top` command has a number of subcommands, of which the most useful to start with are:

h
: gets you help

q
: quits the `top` command

f
: lets you add or remove fields from the display

o
: orders the display order

F
: selects fields to sort on

See the man pages for `top` for full details on options, including how to sort by memory usage or other criteria. Listing 21 shows an example of the output sorted by virtual memory usage in descending order.

Listing 21. Sorting the output of top

```
top - 16:21:48 up 18:43,  5 users,  load average: 0.16, 0.06, 0.01
Tasks: 170 total,   3 running, 167 sleeping,   0 stopped,   0 zombie
Cpu(s):  2.1%us,  0.8%sy,  0.0%ni, 96.6%id,  0.0%wa,  0.0%hi,  0.5%si,  0.0%st
Mem:   4057976k total,  1588940k used,  2469036k free,   195412k buffers
Swap: 10241428k total,        0k used, 10241428k free,   613056k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 6925 ian       20   0 1171m 338m  21m S    0  8.5   1:44.10 java
 1381 root      20   0  634m  40m  18m S    0  1.0   2:13.63 Xorg
 6820 ian       20   0  506m  83m  26m S    3  2.1   0:51.28 firefox
 2004 ian       20   0  436m  23m  15m S    0  0.6   0:01.55 nautilus
 2031 ian       20   0  419m  13m  10m S    0  0.3   0:00.11 evolution-alarm
 2118 ian       20   0  372m  10m 7856 S    0  0.3   0:00.06 evolution-data-
 2122 ian       20   0  344m  13m  10m S    0  0.3   0:00.10 evolution-excha
 2001 ian       20   0  331m  22m  14m S    0  0.6   0:13.61 gnome-panel
 1971 ian       20   0  299m 9.9m 7244 S    0  0.3   0:05.00 gnome-settings-
 1989 ian       20   0  288m  15m  11m S    0  0.4   0:11.95 metacity
 1954 ian       20   0  265m 5460 3412 S    0  0.1   0:00.28 pulseaudio
```

Send signals to processes

Let's now look at Linux *signals*, which are an asynchronous way to communicate with processes. We have already mentioned the SIGHUP signal, and we have used both Ctrl-c and Ctrl-z, which are other ways of sending a signal to processes. The general way to send a signal is with the `kill` command.

Sending signals using kill

The `kill` command sends a signal to a specified job or process. Listing 22 shows the use of the SIGTSTP and SIGCONT signals to stop and resume a background job. Using the SIGTSTP signal is equivalent to using the `fg` command to bring the job to the foreground and then Ctrl-z to suspend it. Using SIGCONT is like using the `bg` command.

Listing 22. Stopping and restarting background jobs

```
ian@attic4:~$ kill -s SIGTSTP %1

[1]+  Stopped                 xclock -d -update 1
ian@attic4:~$ jobs -l
[1]+  3878 Stopped                 xclock -d -update 1
[2]   5485 Running                 nohup sh pmc.sh &
[3]-  5487 Running                 nohup bash pmc.sh &
ian@attic4:~$ kill -s SIGCONT 3878
ian@attic4:~$ jobs -l
[1]   3878 Running                 xclock -d -update 1 &
[2]-  5485 Running                 nohup sh pmc.sh &
[3]+  5487 Running                 nohup bash pmc.sh &
```

We used the job specification (%1) to stop the xclock process in this example, and then the process id (PID) to restart (continue) it. If you stopped job %2 and then used `tail` with the `-f` option to follow it, you would see that only one process is updating the nohup.out file.

There are a number of other possible signals that you can display on your system using `kill -l`. Some are used to report errors such as illegal operation codes, floating point exceptions, or attempts to access memory that a process does not have access to. Notice that signals have both a number, such as 20, and a name, such as SIGTSTP. You may use either the number prefixed by a - sign, or the `-s` option and the signal name. On my system I could have used `kill -20` instead of `kill -s SIGTSTP`. You should always check the signal numbers on your system before assuming which number belongs to which signal.

Signal handlers and process termination

You have seen that Ctrl-c terminates a process. In fact, it sends a SIGINT (or interrupt) signal to the process. If you use `kill` without any signal name, it sends a SIGTERM signal. For most purposes, these two signals are equivalent.

You have seen that the `nohup` command makes a process immune to the SIGHUP signal. In general, a process can implement a *signal handler* to *catch* signals. So a process could implement a signal handler to catch either SIGINT or SIGTERM. Since the signal handler knows what signal was sent, it may choose to ignore SIGINT and only terminate when it receives SIGTERM, for example. Listing 23 shows how to send the SIGTERM signal to job %2. Notice that the process status shows as "Terminated" right after we send the signal. This would show as "Interrupt" if we used SIGINT instead. After a few moments, the process cleanup has occurred and the job no longer shows in the job list.

Listing 23. Terminating a process with SIGTERM

```
ian@attic4:~$ kill -s SIGTERM %2
ian@attic4:~$
[2]-  Terminated              nohup sh pmc.sh
ian@attic4:~$ jobs -l
[1]-  3878 Running                 xclock -d -update 1 &
[3]+  5487 Running                 nohup bash pmc.sh &
```

Signal handlers give a process great flexibility. A process can do its normal work and be interrupted by a signal for some special purpose. Besides allowing a process to catch termination requests and take possible action such as closing files or checkpointing transactions in progress, signals are often used to tell a daemon process to reread its configuration file and possibly restart operation. You might do this for the inetd process when you change network parameters, or the line printer daemon (lpd) when you add a new printer.

Terminating processes unconditionally

Some signals cannot be caught, such as some hardware exceptions. SIGKILL, the most likely one you will use, cannot be caught by a signal handler and unconditionally terminates a process. In general, you should need this only if all other means of terminating the process have failed.

Logout and nohup

Remember you saw that using `nohup` would allow your processes to keep running after you log out. Well, let's do that and then log back in again. After you log back in, check your remaining poor man's clock process using `jobs` and `ps` as we have done above. The output is shown in Listing 24.

Listing 24. Logging back in

```
ian@attic4:~$ jobs -l
ian@attic4:~$ ps -a
  PID TTY          TIME CMD
10995 pts/0    00:00:00 ps
```

We are running on pts/0 this time, but there is no sign of our jobs, just the `ps` command. Not perhaps what we were expecting. However, all is not lost. Suppose you can't remember whether you terminated the nohup job that you started with bash or the one you started with bash. You saw above how to find the processes that were running the `getty` command, so you can use the same trick to display just the SID, PID, PPID, and command string. Then you can use the `-js` option to display all the processes in the session. Listing 25 shows the result. Think about other ways you might have found these processes, such as searching by username and then filtering using `grep`.

Listing 25. Finding our lost commands

```
ian@attic4:~$ ps -C bash -C sh -o pid,sid,tname,cmd
  PID   SID TTY      CMD
 5487  2643 ?        bash pmc.sh
 7050  7050 pts/3    -bash
10851 10851 pts/0    bash
ian@attic4:~$ ps -js 2643
  PID  PGID   SID TTY          TIME CMD
 5487  5487  2643 ?        00:00:00 bash
11197  5487  2643 ?        00:00:00 sleep
```

Note that the pmc.sh is still running but now it has a question mark (?) for the controlling TTY.

Given what you have now learned about killing processes, you should be able to kill the remaining poor man's clock process using its PID and the `kill` command.

Resources

Learn

- Develop and deploy your next app on the IBM Bluemix cloud platform.
- Use the developerWorks roadmap for LPIC-1 to find the developerWorks articles to help you study for LPIC-1 certification based on the April 2009 objectives.
- At the LPIC Program site, find detailed objectives, task lists, and sample questions for the three levels of the Linux Professional Institute's Linux system administration certification. In particular, see their April 2009 objectives for LPI exam 101 and LPI exam 102. Always refer to the LPIC Program site for the latest objectives.
- Review the entire LPI exam prep series on developerWorks to learn Linux fundamentals and prepare for system administrator certification based on earlier LPI exam objectives prior to April 2009.
- In "Basic tasks for new Linux developers" (developerWorks, March 2005), learn how to open a terminal window or shell prompt and much more.
- The Linux Documentation Project has a variety of useful documents, especially its HOWTOs.
- Read more of Ian's articles on developerWorks, and connect with him through his profile in My developerWorks.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tutorials and Linux tips on developerWorks.
- Stay current with developerWorks technical events and Webcasts.
- Follow developerWorks on Twitter.

Get products and technologies

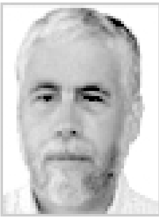
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Participate in the discussion forum for this content.
- Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Ian Shields



Ian Shields works on a multitude of Linux projects for the developerWorks Linux zone. He is a Senior Programmer at IBM at the Research Triangle Park, NC. He joined IBM in Canberra, Australia, as a Systems Engineer in 1973, and has since worked on communications systems and pervasive computing in Montreal, Canada, and RTP, NC. He has several patents and has published several papers. His undergraduate degree is in pure mathematics and philosophy from the Australian National University. He has an M.S. and Ph.D. in computer science from North Carolina State University. Learn more about Ian in Ian's profile on developerWorks Community.

© Copyright IBM Corporation 2010
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)