Process Management

Romain Lerallut

rom1@users.sourceforge.net

A documentation about how to manage processes under Linux

1. Introduction

One of the strengths of Linux is its excellent process management. A crashing process cannot bring down the system with it, and a lot of work has been done on multitasking so that each process gets its share.

1.1. Credits

Thanks to the NewbieDoc team, to Ken Thomson and Dennis Ritchie for Unix, and to the Linux developers for Linux !

1.2. Feedback

As usual, comments, corrections, flames, money are all equally welcome !

2. Unix and processes

Most of what will be said in this doc is not Linux-specific, but should apply to most Unix systems

A process running on a Unix system can be managed using five numbers:

- **Process ID**: a number defined by the system when the process starts. It is unique: you may have many xterms running but only one will have 123 as its PID .

  Note:each process has a parent (except init, being The First) and keeps the parent's PID stored somewhere called PPID.

- **User ID**: usually the UID of the user that started the process, it is used to manage permissions and accounting.
- **Priority**: a number ranging from -20 to +20, negative numbers being reserved for the Superuser.

The lower a job's priority number, the more important it is, go figure... Use nice (renice) to define (modify) a process's priority when starting it (any time).

3. Process Management

3.1. Finding info about a process

There are two tools used to find info about running processes: **ps** and **top**

3.1.1. Static Process Management: ps

ps is a tool found on most if not all flavours of UNIX. Its syntax varies a lot, and for the sake of clearness I will speak only about the GNU-ps' syntax.

At the shell prompt, type

```
ps
```

you should see something like:

```
PID TTY          TIME CMD
10923 pts/3    00:00:00 bash
12494 pts/3    00:00:00 ps
```

which are the currently running process that are attached to your shell.

To see all your processes, type

```
ps xu
```

with **x** for "all processes even those that are not launched from a terminal" and **u** for slightly more detailed info

To see all processes, including those belonging to others type

```
ps axu
```

Note: some older versions of ps use the

```
ps -axu
```

syntax, however it is deprecated.

See the **ps manpage** of the column headers of ps' output to find which column refers to what field.

Tip: use ps and grep together to find a specific process:

```
ps axu | grep my_process_name
```

3.1.2. Dynamic Process Management: top

"top" is a nice useful tool that acts like ps but with an auto-refresh.

just type

```
top
```

and get my meaning. You'll see all your running processes ordered by decreasing CPU usage.

Common interactive options are:

- **p**: sort by CPU usage (default)
- **m**: sort by memory usage
- **s**: adjust time between refreshes.
- **q**: quit

"top" displays first general info about your system, such as uptime, number of users ,etc.

Then it reports global CPU and memory usage

Then come the list of running processes. Interesting fields are:

- **PID**: PID of the process
- **user**: owner of the process
- **PRI**: priority, note that top's priority floats around 19
- **SIZE**: size of the process, RSS is the size of the physical memory allocate, SHARE being the size of shared memory
- **STAT**: state of the process: R for running, S for sleeping, Z for zombies.

The remaining columns should be easy to understand :-)...

3.2. Signals

3.2.1. What are signals ?

In Unix the owner of a process (or the superuser) can send a *signal* to a process he owns (any process for the SU).

Some of these signals can be recognized by processes which can act accordingly.

Example: say you have a laptop, and a battey-monitoring tool. If the battery runs low, your monitoring program can send signals (SIGPWR, I guess) to your running programs so that they shut down correctly and then poweroff the computer.

3.2.2. Which signals should I know of ?

- **SIGKILL (IMPORTANT)**: kills a process using deadly force. The process doesn't stand a chance of survival, and exits *without* performing any action, such as save-and-exit. USE CAUTION !
- **SIGTERM (IMPORTANT)**: asks strongly to a program to nicely stop. Most evolved programs "trap this signal" , and run some save-and-exit code. It might not work if the program is stone-dead, since it won't execute any code at all.
- **SIGINT**: it's one you already know about ! That's what Ctrl-c sends. It's a nice way of saying goodbye to a process.
- **SIGQUIT**: you already know this one also, it's usually what Ctrl-d sends when typed in a terminal.
- **SIGSTOP et SIGCONT**: to temporarily halt a process or to ask it to continue. (Ctrl-z sends SIGSTOP whereas fg and bg send SIGCONT).

3.2.3. How do I send a signal ?

It is simple. You must know the PID of a process and then:

```
kill -signal_number PID
```

for example:

```
kill -9 123
```

or

```
kill -SIGKILL 123
```

will terminate with maximum prejudice process #123.

You can find the conversion table between signals' names and numbers by doing

```
kill -l
```

3.3. Nice and priorities

If you want to run a process without hogging 99% of the CPU and upset other users, **nice** is for you.

A process is usually run with the priority of its parent. To know your shell's priority type:

```
nice
```

To run a niced command type:

```
nice my_command
```

It will (by default) add 10 to your shell's priority and define it as the process' priority. If your shell's priority is 0, your new process will have priority of 10.

To specify the nice-level (relative to current shell's priority):

```
nice -n 14
```

so the priority of the new process will be current_shell_priority+14.

3.4. Renice a job's priorities

To renice a job's priority, use **renice**.

```
renice +5 my_pid
```

You can also specify "all the processes of a user":

```
renice +20 -u patient_guy
```

This will add 20 to the priority of all patient_guy's processes. (you usually need to be root to do this, and to be able to run fast :-)

4. Tips and Tricks

4.1. Killing processes

Let's face it: the only time we use kill to send a signal is to kill a process. And that's why it's called kill.

I usually have netscape crashes, so I have a small shell script that does:

```
#!/bin/sh
ps xu | grep netscape | grep -v grep | awk '{ print $2 }' | xargs kill -9
```

This is what is does:

1. prints all my processes
2. takes only the lines containing 'netscape"

3. discards the line about 'grep netscape'
4. selects the second column of ps xu (the one showing the PID)
5. xargs builds the command "kill -9 PID" by taking everyword coming from the pipe and launching kill -9 on it. If I had several PIDs coming down the pipe, xargs would launch a kill -9 on each of those PIDs.