Previous parts on the DVD

The Mike Saunders SCHOOL of LINUX

Part 7: Learn how to handle processes like a pro, and get to grips with the famously minimal Vi editor.

Mike

Linux Professional Institute

Our expert

Mike Saunders has been writing about Linux for over a decade, and has installed more distros than he's had hot dinners.

We're coming towards the end of our LPI series of tutorials, with the final instalment due next month, so it's time to look at a few advanced topics that you might come across on your system administrator travels.

We're going to kick off with a look at processes, and how you can manipulate them to your liking. There's nothing worse than an errant process deleting important files and leaving you feeling helpless, so we're going to look at solutions to this problem. We'll also look at filesystems, not in terms of the contents as we covered that earlier, but in how to format partitions as new filesystem types and perform checks on them in case anything goes wrong.

Finally, we'll explore the *Vi* editor that's supplied with almost every Linux and Unix installation under the sun, and which is notoriously difficult to use at first but can be a godsend when you've got the essentials sorted.

Next issue we'll have a detailed set of LPI training questions, so once you've finished this tutorial, you have a few weeks to get revising. Good luck!

Section 1: Managing processes

Imagine you're sitting at home, and you fancy a nice cup of tea. Being the industrious sort that you are, you call out to a nearby family member to make one for you. Said family member heads off to the kitchen, and as you look over the back of your comfy armchair, you can see that he/she is making a cup of coffee instead. Now, in this situation you'd call out with a statement like "Gadzooks, I requested a fine cup of tea please!" or something similar. But how does it work in computing terms? What happens if you set a program running, and you want to stop it or change the way it works?

At the most basic level, the equivalent to yelling "stop" is pressing Ctrl+C. Try it with a command that generates vast amounts of output, like **ls -R /** to list the root directory and all subdirectories. As it spits out thousands of lines to your terminal, you can hit Ctrl+C to stop it mid flow. It's finished; there's nothing more that you can do. This is mightily important when you realise you've just entered something crazy, and you need to stop it before any damage is done.

There's an alternative to this, however. What if you want to merely pause the program's execution for later? Say, for instance, you've just entered **man gcc** to read the manual page for the GNU C Compiler. You've scrolled around and found an interesting point in the documentation – so you want to try out some things, without losing your position. Hit Ctrl+Z, and the manual page viewer will disappear into the background, putting you at the command prompt. You can do your work and then type **fg** (for foreground) to bring the manual page viewer to the front, exactly where you left it.

It's possible to start a program in non-viewing mode (suspended), so that you can switch to it when you're ready. This is done by appending an ampersand (**&**) character, like this:

```
man df &
```

» **Last month** Advanced tricks and techniques using the command line.

Nice to see you, to see you, nice

By default, a process doesn't have more rights to resources than any other process on the system. If process A and process B are started, and they're both taxing the CPU, the Linux kernel scheduler will split time evenly between them. However, this isn't always desirable, especially when you have many processes running in the background. For instance, you might have a cron job (periodic task) set up to compress old archive files on a desktop machine: if the user is doing something important, you don't want them to suddenly lose 50% of their processing power whenever that cron job comes up.

To combat this, there's a system of priorities. Each process has a **nice** value, which sets how the OS should treat it, with 19 the lowest priority, counting upwards to zero for the default, and -20 for the top priority. For instance, if you want to start a program with the lowest priority, use:

```
nice -n 19 programname
```

This will run the program, and if nothing else is happening on the system, it should complete in normal time. However, if the system gets under load from other processes, it will deal with them first. For **nice** values above zero, you have to be root:

```
sudo nice -n -10 programname
```

This is for multi-user systems, where the administrator wants to give his tasks priority (otherwise everyone else would be giving their processes maximum niceness!). You can change a process's nice value using the **renice** command – see its manual page for more information – and see the values in the output of **top**.

Here, we start the manual page viewer for the **df** (show disk free space) command, but in the background. We get a line of feedback on the screen:

```
[1] 3192
```

The second number here is the process ID (we'll come onto that in a minute). You can now go about doing your work, and when you're ready to join up with the program you started, just enter **fg**.

This system becomes especially useful when you combine multiple programs. For instance, enter:

```
nano &
man df &
```

Here we've started two programs in the background. If we enter **jobs**, we get a list of them, with numbers at the beginning and their command lines. We can resume specific programs using a number – for instance, **fg 1** will switch to the *Nano* editor, and **fg 2** to the manual page viewer.

Let's move on to processes. Ultimately, a process is an instance of execution by a program: most simple programs provide one process, which is the program itself. More complicated suites of software, such as a desktop environment, start many processes – file monitoring daemons, window managers and so forth. This helps with system maintenance (imagine if all of KDE was one gigantic, fat executable where everything stopped if one component crashed), and allows us to do some useful things as well.

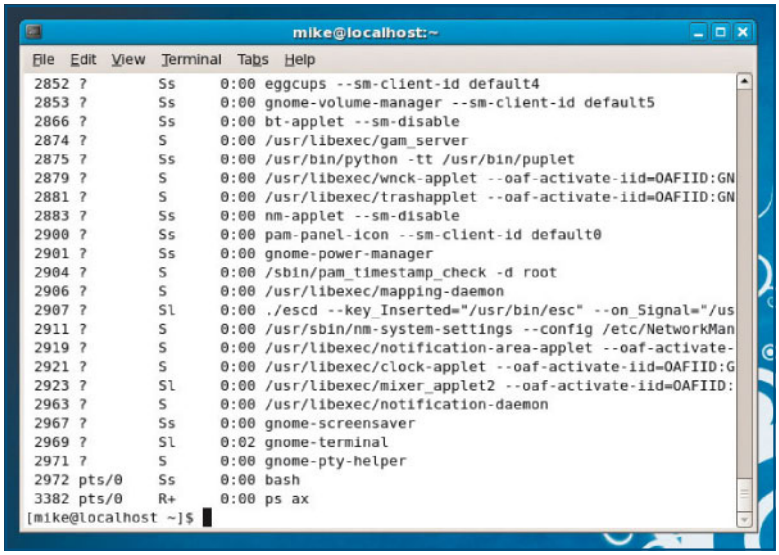
To show a list of processes, enter **ps**. You'll find the output rather uninspiring, as it's likely to be just a couple of lines long; this is because it's only showing processes being run by the current user. To view all processes running on the machine, enter **ps ax**. Typically, this will be very long, so you can pipe it to the **less** viewer as described last month:

```
ps ax | less
```

Exactly what you see will depend on the specific makeup of your Linux installation and currently running programs, but here's an example line:

```
2972 pts/0        Ss     0:00 bash
```

The **2972** here is the process ID (PID). Every process has a unique ID, starting from 1, which is the **/sbin/init** program that the kernel runs on boot. After that, the **pts/0** bit shows from which virtual terminal the command was run – if you see a question mark here, it's a process that was started outside of a terminal, eg by the kernel or a boot script. The **Ss** says that the process is sleeping (not doing any active processing), then there's a time indicator showing how much CPU time the process has consumed so far, followed by the command line used to start the process.



› Here's the output of ps ax, showing all running processes on the system.

An alternative way to get a list of processes, and one which is by default sorted by CPU usage, is by entering **top**. This is an interactive program that updates every few seconds, showing the most CPU-intensive tasks at the top of the list. It also provides headings (in the black bar) for the columns, so you can determine a process's PID, the user who started it and so forth. Note that while there are various columns for memory usage, the most important is RES (resident), which shows exactly how much real memory the process is currently using up. To exit **top**, press Q.

So, let's say you are happily running a program, and suddenly it goes haywire, gets stuck in a loop and is occupying all the CPU. You can't kill it with Ctrl+C – you might've started it from your window manager's menu. What can you do? The first option is to find out its PID using the previous methods, and then enter:

```
kill <pid>
```

Replace **<pid>** with the number. Although this command is named after the act of murdering something, it's actually rather soft in its default state. On its own, **kill** sends a friendly "Would you mind shutting down?" message to the process, which the process can then deal with (eg, cleaning up temporary files before shutting down). Sometimes this will

Quick tip

Key combinations to stop and pause processes, such as Ctrl+C and Ctrl+Z, work in most cases but not always. It's possible for programs to remap those keys to provide other functionality – or to stop employees from jumping out of their important programs to play *Nethack*!

»

Quick tip

You can get a quick overview of how your system is performing with the **uptime** command. This shows how much time has expired since the last (re)boot, how many users are logged in, and the load average for recent periods of time.

» stop a process, but if that process has its own **kill** signal handler and is too messed up to deal with it, you're stuck. This is when **kill** starts to justify its name. Enter:

```
kill -9 <pid>
```

This doesn't even bother asking the program if it's OK – it just stops it immediately. If the process is halfway through writing a file, the results could be very messy, so this should be used with extreme care, when no other option is available.

Sometimes you might have multiple processes with the same name, or you just don't want to look up its PID. In this case, you can use the **killall** command. For instance, say you've compiled some super bleeding-edge *Apache* module, inserted it into *Apache*, and now your web server is going haywire. You can't stop *Apache* via its normal scripts, and there are loads of processes called **apache** running. Try this:

```
killall -9 apache
```

Another useful signal which isn't destructive but informs a program to restart itself or re-read its configuration files is **SIGHUP**, named after "hanging up" from dialup modem devices. Many programs will ignore this, but it works well for certain background daemons such as servers:

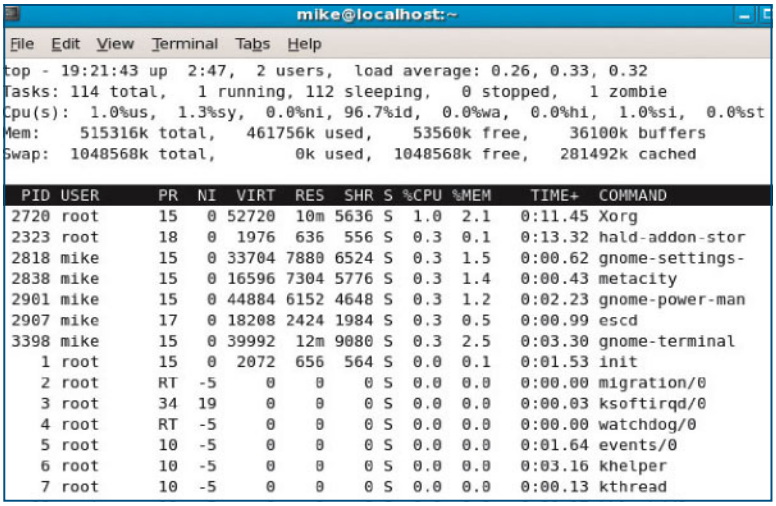
```
killall -HUP sendmail
```

This tells all **sendmail** processes running to slow down a second, re-read the config files and then carry on. This is very useful when you want to make a quick change to a config file, and not take down the whole program. If, for some reason, you don't want this signal to be processed, use the **nohup** tool to disable it – see **man nohup** for more information.

Section 2: Creating new filesystems

Let's move on to an advanced topic that, usually, you won't have to be concerned with as a system administrator: filesystems and partitioning. At least, you won't be dealing with it on a day-to-day basis. In most cases, you'll determine the partition setup of a machine at installation time, and that'll be it for months or even years. Graphical tools like *GParted*, used by many distro installers, make this process a cinch, although it's important to be aware of command line tools for emergency situations.

› Entering top shows a list of processes sorted by their CPU usage.



First, a recap: a hard drive is usually divided into multiple partitions. You might have a partition for Windows, for instance, and then a partition for Linux. Most Linux installations exist in two or more partitions: data partitions (such as **/** and **/home**) and then the swap partition for virtual memory. Data on these partitions has to be in some kind of order, or format, and historically in Linux that was the **ext2** filesystem format. Then came **ext3** (with journaling), and now we're on **ext4**. Other partition types from the Linux and UNIX world include XFS and ReiserFS. Most Windows machines use NTFS, but external storage devices such as USB keys tend to use FAT32 (also known as VFAT in Linux).

The most basic tool for partitioning from the command line is *fdisk*. Provide it with the device node for your hard drive, like this:

Testing filesystem integrity

Modern Linux filesystems, such as **ext4** (the default on most installations today), are robust and reliable. They can't perform miracles in the event of a power outage, but they can try to leave the filesystem in a reasonably consistent state, so that you don't lose all your data. However, nothing is invincible and if you've had a major problem with your hardware, you might suspect that something is wrong with your disk. Here's how an administrator would sort it out.

First, enter **dmesg** and see if there's anything funny in the logs – anything that stands out to do with data corruption, bad sectors and so forth. If you see anything like that, pop a USB thumb drive in the machine and copy all important data immediately, because you never know when the whole drive might fail. Next, Use the **df -h** command to see how much free space is on the drive; if it's much smaller than you expected, something might be wrong. Use **du -h** in directories to see disk usage there.

The next step is to perform a filesystem check. Reboot your distro into single-user mode (the method for this varies between distros, but usually involves editing the kernel line in the boot loader, and adding **S** to the end.) When you reach the command prompt, enter:

```
/sbin/fsck device-name
```

Change **device-name** for the device node for your root hard drive partition – eg **/dev/sda1**. You can also check other partitions too. **fsck** is actually a front-end to various filesystem checking tools, and on most Linux boxes runs **/sbin/e2fsck**, which handles **ext2/3/4** filesystems. During the checking process, if problems are found, **fsck** will ask you what you want to do. After checking, you can run **/sbin/dumpe2fs** followed by the device node to get more information about the partition, which helps if you need to report a problem in an online forum.

You may notice that many Linux distributions automatically run **fsck** filesystem checks after every 30 boots, or every 100 days, or a combination. You can change how this works with the **tune2fs** tool and its **-c** and **-C** options. There are also settings for how the kernel should treat a filesystem if it spots errors, and many other features. It's well worth reading the manual page, especially the information about the first five options. A similar utility, albeit extremely technical, is **debugfs** – but you really need to know the internals of filesystem design to make good use of it.

```
/sbin/fdisk /dev/sda
```

(Note that this has to be run as root, and if you're not sure which device node your hard drive is, look in the output of **dmesg**.) Also note here that we're just using **/dev/sda**, and not **/dev/sda1** – the latter is the number for a specific partition, whereas we just want the whole disk. In most cases, **sda** is the first hard drive, **sdb** is the second, and so on.

fdisk is a rather bare program; there are no menus or wizards to automate things. Enter **p** to get a list of partitions on your hard drive, and **m** to get help. There you'll see which commands delete partitions, create new partitions and so forth. Any changes you make aren't actually committed until you enter **w** to write them to disk. Some distros include *cfdisk*, a curses-based version of *fdisk* that makes things a bit easier: there are simple menus and you move around with the cursor keys.

After you've made a partition, you need to format it. This is where the *mkfs* tools come into play. Type **/sbin/mkfs** and then hit tab to show possible options – you'll see there's **mkfs.ext3** (for most Linux partitions), **mkfs.vfat** (for FAT32 partitions) and more. To format a partition, just provide its device node:

```
/sbin/mkfs.ext3 /dev/sda2
```

For swap partitions, use the **mkswap** command. You can then enable and deactivate swap space with the **swapon** and **swapoff** commands.

Quick tip

To get an indication of how much memory is available, enter **free -m**. This shows statistics in megabytes. The first line might shock you, and make you think there's hardly any memory left, even if you're just running *Fluxbox*. But that includes disk caches – so the second line, **-/+ buffers/cache**, is the one to look at.

Section 3: A quick tour of the Vi editor

Finally this month, we're going to take a brief look at *Vi*, the "visual" editor. Yes, it might sound ridiculous that an editor is described as "visual" – surely they all are? But back when *Vi* was developed for Unix OSes in the 1970s, some people were still using teletype machines.

The concept of a full-screen editor was rather novel, as people were used to working on individual lines of a text file. Still, *Vi* is very terse and basic, but because of its low requirements, it's installed by default on virtually every Unix machine. In the Linux world, most distros include *Vim* (*Vi Improved*), a much more advanced and capable version of the editor.

Getting started

To start, run **vi filename.txt**. Before you press a key, note that *Vi* operates in two modes: normal (for commands) and insert (for editing text). This is in contrast to most other editors where you just begin typing straight away.

In *Vi*, you have to press **i** to insert text at the current location, which then lets you type what you want. When you're finished, press Esc to return to normal mode, ready for commands.

There are many commands, and if you want to become a *Vi* guru then there are plenty of books available. But some essentials: in normal mode, entering **dd** deletes a line, **yy** yanks (copies) a line to the clipboard, and **p** pastes that line back out.

There are some operations which require that you enter a colon first. For instance **:w** writes the file to disk, while **:q** quits the editor. (If you've made edits and haven't saved, and then try to quit, *Vi* might complain – you can tell it to quit without saving using **:q!**. You can even combine actions, like writing and saving with **:wq**.

Many people find *Vi* and *Vim* extremely uncomfortable to work with, and yearn for modeless editors such as *Emacs* or *Nano*. Others love its minimalism and hate the Ctrl key obsession of those two other editors. It's a war that'll run on and on, but regardless of the outcome, all good admins know some basic *Vi*, as you can pretty much guarantee it'll be on every box you encounter. LXF

```
                VIM - Vi IMproved

                 version 7.0.237
              by Bram Moolenaar et al.
        Modified by <bugzilla@redhat.com>
    Vim is open source and freely distributable

           Help poor children in Uganda!
type  :help iccf<Enter>       for information

type  :q<Enter>               to exit
type  :help<Enter>  or  <F1>  for on-line help
type  :help version7<Enter>   for version info
```

Helpfully *Vim*, unlike regular *Vi*, gives you some help text when you start it without a filename.

Test yourself!

As you progress through this series of tutorials, you may want to assess your knowledge along the way. After all, if you go full-on for LPI certification at the end, you'll need to be able to use your knowledge on the spot, without consulting the guides. So, make sure you've read, internalised and tried out everything in this tutorial, and then see if you can answer the questions below.

1. How would you run the command **exim -q** in the background?
2. You have several programs running in the background. How do you bring up a list of them?
3. How do you generate a list of all processes?
4. *Exim* has gone haywire, and you need to completely terminate all instances of it. What's the command?
5. You've just created a new partition, **/dev/sda2**, and you want to format it as FAT32. How?
6. Provide a way to start **myprog** with the lowest priority.

1 – exim -q &. 2 – jobs. 3 – ps ax. 4 – killall -9 exim. 5 – /sbin/mkfs.vfat /dev/sda2. 6 – nice -n 19 myprog.

Next month The last few bits and pieces, and sample test questions galore!