



Taking Control of the System

By now, you should be starting to realize that the shell offers an enormous amount of power when it comes to administering your PC. The BASH shell commands give you quick and efficient control over most aspects of your Linux setup. However, the shell truly excels in one area: controlling the processes on your system.

Controlling processes is essential for administration of your system. You can tidy up crashed programs, for example, or even alter the priority of a program so that it runs with a little more consideration for other programs. Unlike with Windows, this degree of control is not considered out of bounds. This is just one more example of how Linux provides complete access to its inner workings and puts you in control.

Without further ado, let's take a look at what can be done.

Viewing Processes

A process is something that exists entirely behind the scenes. When the user runs a program, one or many processes might be started, but they're usually invisible unless the user specifically chooses to manipulate them. You might say that programs exist in the world of the user, but processes belong in the world of the system.

Processes can be started not only by the user, but also by the system itself to undertake tasks such as system maintenance, or even to provide basic functionality, such as the GUI system. Many processes are started when the computer boots up, and then they sit in the background, waiting until they're needed (such as programs that send mail). Other processes are designed to work periodically to accomplish certain tasks, such as ensuring system files are up-to-date.

You can see what processes are currently running on your computer by running the `top` program. Running `top` is simply a matter of typing the command at the shell prompt.

As you can see in Figure 16-1, `top` provides very comprehensive information and can be a bit overwhelming at first sight. However, the main area of interest is the list of processes (which `top` refers to as *tasks*).

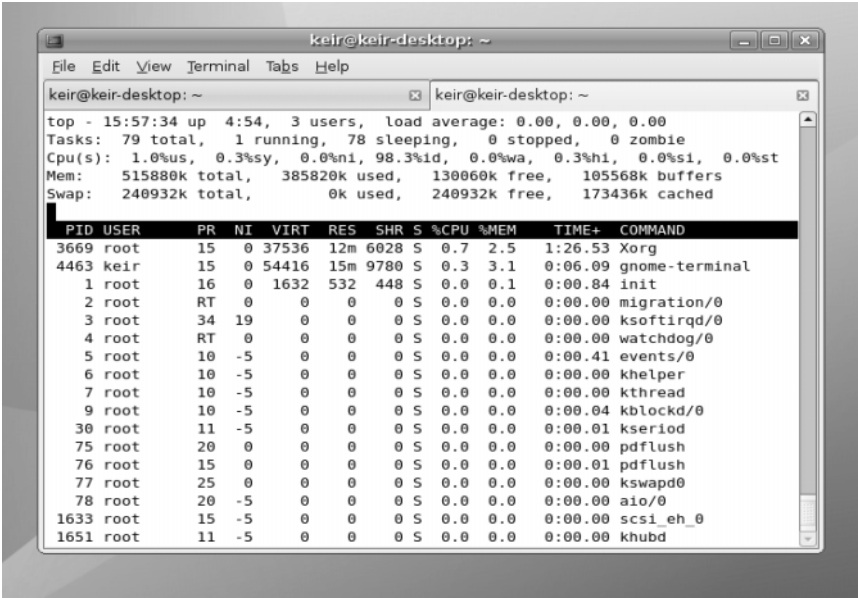


Figure 16-1. The *top* program gives you an eagle-eye view of the processes running on your system.

Here’s an example of a line taken from *top* on our test PC, shown with the column headings from the process list:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
5499	root	15	0	78052	25m	60m	S	2.3	5.0	6:11.72	Xorg

A lot of information is presented here, as described in Table 16-1.

Table 16-1. The *top* Program Process Information

Column	Description
PID	The first number is the process ID (PID). This is the unique number that the system uses to track the process. The PID comes in handy if you want to kill (terminate) the process (as explained in the next section of this chapter).
USER	This column lists the owner of the particular process. As with files, all processes must have an owner. A lot of processes will appear to be owned by the root user. Some of them are system processes that need to access the system hardware, which is something only the root user is allowed to do. Other processes are owned by root for protection; root ownership means that ordinary users cannot tamper with these processes.

Table 16-1. *The top Program Process Information*

Column	Description
PR	This column shows the priority of the process. This is a dynamic number, showing where the particular process is in the CPU queue at the present time.
NI	This column shows the “nice” value of the process. This refers to how charitable a process is in its desire for CPU time. A high figure here (up to 19) indicates that the process is willing to be interrupted for the sake of other processes. A negative value means the opposite: the process is more aggressive than others in its desire for CPU time. Some programs need to operate in this way, and this is not necessarily a bad thing.
VIRT	This column shows the amount of virtual memory used by the process. ¹
RES	This column shows the total amount of physical memory used. ¹
SHR	This column shows the amount of shared memory used. This refers to memory that contains code that is relied on by other processes and programs.
S	This column shows the current status of the task. Generally, the status will either be sleeping, in which case an <i>S</i> will appear, or running, in which case an <i>R</i> will appear. Most processes will be sleeping, even ones that appear to be active. Don’t worry about this; it just reflects the way the Linux kernel works. A <i>Z</i> in this column indicates a zombie process (a child of a process that has been terminated).
%CPU	This column shows the CPU use, expressed as a percentage. ²
%MEM	This column shows the memory use, again expressed as a percentage. ²
TIME+	This column shows a measure of how long the process has been up and running.
COMMAND	This shows the actual name of the process itself.

¹ Both *VIRT* and *RES* are measured in kilobytes unless an *m* appears alongside the number; in which case, you should read the figure as megabytes.

² The *%CPU* and *%MEM* entries tell you in easy-to-understand terms how much of the system resources a process is taking up.

This list will probably be longer than the screen has space to display, so *top* orders the list of processes by the amount of CPU time the processes are using. Every few seconds, it updates the list. You can test this quite easily. Let your PC rest for a few seconds, without touching the mouse or typing. Then move the mouse around for a few seconds. You’ll see that the process called *Xorg* leaps to the top of the list (or appears very near the top). *Xorg* is the program that provides the graphical subsystem for Linux, and making the mouse cursor appear to move around the screen requires CPU time. When nothing else is going on, moving the mouse causes *Xorg* to appear as the number one user of CPU time on your system.

Tip Typing *d* while *top* is running lets you alter the update interval, which is the time between screen updates. The default is three seconds, but you can reduce that to one second or even less if you wish. However, a constantly updating *top* program starts to consume system resources and can therefore skew the diagnostic results you’re investigating. Because of this, a longer, rather than shorter, interval is preferable.

It's possible to alter the ordering of the process list according to other criteria. For example, you can list the processes by the quantity of memory they're using, by typing `M` while `top` is up and running. You can switch back to CPU ordering by typing `P`.

RENICING A PROCESS

You can set how much CPU time a process receives while it's actually running. This is done by *renicing* the process. This isn't something you should do on a regular basis, but it can prove very handy if you start a program that then uses a lot of system resources and makes the system unbearably slow.

The first thing to do is to use `top` to spot the process that needs to be restrained and find out its PID number. This will be listed on the left of the program's entry on the list. Once you know this, type `r`, and then type in the PID number. You'll then be asked to specify a renice value. The scale goes from `-20`, which is considered the highest priority, to `19`, which is considered the lowest. Therefore, you should type `19`. After this, you should find some responsiveness has returned to the system, although how much (if any) depends on the nature of the programs you're running.

You might be tempted to bump up the priority of a process to make it run faster, but this may not work because of complexities in the Linux kernel. In fact, it might cause serious problems. Therefore, you should renice with care and only when you must.

Renicing can also be carried out via the `renice` command at the prompt, avoiding the need to use `top`. Also useful is the `nice` command, which can be used to set the initial priority of a process before it starts to run. To learn more, see the man pages for `renice` and `nice`.

Controlling Processes

Despite the fact that processes running on your computer are usually hidden away, Linux offers complete, unrestricted, and unapologetic control over them. You can terminate processes, change their properties, and learn every item of information there is to know about them.

This provides ample scope for damaging the currently running system but, in spite of this, even standard users have complete control over processes that they personally started (one exception is zombie processes, described a bit later in this section). As you might expect, the root user (or any user who adopts superuser powers) has control over all processes that were created by ordinary users, as well as those processes started by the system itself.

The user is given this degree of control over processes in order to enact repairs when something goes wrong, such as when a program crashes and won't terminate cleanly. It's impossible for standard users to damage the currently running system by undertaking such work, although they can cause themselves a number of problems.

Note This control over processes is what makes Linux so reliable. Because any user can delve into the workings of the kernel and terminate individual processes, crashed programs can be cleaned up with negligible impact on the rest of the system.

Killing Processes

Whenever you quit a program or, in some cases, when it completes the task you've asked of it, it will terminate itself. This means ending its own process and also that of any other processes it created in order to run. The main process is called the *parent*, and the ones it creates are referred to as *child* processes.

Tip You can see a nice graphical display of which parent owns which child process by typing `pstree` at the command-line shell. It's worth piping this into the `less` command so you can scroll through it: `type pstree | less`. We explain what piping is in the next chapter.

While this should mean your system runs smoothly, badly behaved programs sometimes don't go away. They stick around in the process list. Alternatively, you might find that a program crashes and so isn't able to terminate itself. In very rare cases, some programs that appear otherwise healthy might get carried away and start consuming a lot of system resources. You can tell when this happens because your system will start slowing down for no reason, as less and less memory and/or CPU time is available to run actual programs.

In all of these cases, the user usually must kill the process in order to terminate it manually. This is easily done using `top`.

The first task is to track down the crashed or otherwise problematic process. In `top`, look for a process that matches the name of the program, as shown in Figure 16-2. For example, the Firefox web browser generally runs as a process called `firefox-bin`.

MEM	TIME+	COMMAND
3.0	14:09.19	Xorg
1.7	0:01.16	firefox-bin
1.9	0:00.73	metacity
1.8	0:00.54	wnck-applet
1.0	0:07.73	gnome-terminal
1.5	0:00.70	gnome-settings-
1.0	0:01.22	kblockd/0
1.0	0:00.12	kswapd0
1.7	0:00.71	gnome-panel
1.0	0:00.96	nautilus
1.7	0:00.33	update-notifier

Figure 16-2. You can normally identify a program by its name in the process list.

Caution You should be absolutely sure that you know the correct process before killing it. If you get it wrong, you could cause other programs to stop running.

Because `top` doesn't show every single process on its screen, tracking down the trouble-causing process can be difficult. A handy tip is to make `top` show only the processes created by the user you're logged in under. This will remove the background processes started by `root`. You can do this within `top` by typing `u` and then entering your username.

Once you've spotted the crashed process, make a note of its PID number, which will be at the very left of its entry in the list. Then type `k`. You'll be asked to enter the PID number. Enter that number, and then press `Enter` once again (this will accept the default signal value of 15, which will tell the program to terminate).

With any luck, the process (and the program in question) will disappear. If it doesn't, the process you've killed might be the child of another process that also must be killed. To track down the parent process, you need to configure `top` to add the PPID field, for the parent process ID, to its display. To add this field, type `f`, and then `b`. Press `Enter` to return to the process list. The PPID column will appear next to the process name on the right of the window. It simply shows the PID of the parent process. You can use this information to look for the parent process within the main list of processes.

The trick here is to make sure that the parent process isn't something that's vital to the running of the system. If it isn't, you can safely kill it. This should have the result of killing the child process you uncovered prior to this.

Caution In both the PPID and PID fields, you should always watch out for low numbers, particularly one-, two- or three-digit numbers. These are usually processes that started early on when Linux booted and that are essential to the system.

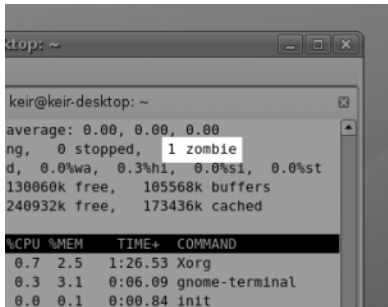
Controlling Zombie Processes

Zombie processes are those that are children of processes that have terminated. However, for some reason, they failed to take their child processes with them. Zombie processes are rare on most Linux systems.

Despite their name, zombie processes are harmless. They're not actually running and don't take up system resources. However, if you want your system to be spick-and-span, you can attempt to kill them.

In the top-right area of `top`, you can see a display that shows how many zombie processes are running on your system, as shown in Figure 16-3. Zombie processes are easily identified because they have a `Z` in the status (`S`) column within `top`'s process list. To kill a zombie

process, type `k`, and then type its PID. Then type 9, rather than accept the default signal of 15.



```

top: ~
keir@keir-desktop: ~
average: 0.00, 0.00, 0.00
ng, 0 stopped, 1 zombie
d, 0.0%wa, 0.3%hi, 0.0%si, 0.0%st
130060k free, 105568k buffers
240932k free, 173436k cached

%CPU %MEM  TIME+  COMMAND
 0.7  2.5   1:26.53 Xorg
 0.3  3.1   0:06.09 gnome-terminal
 0.0  0.1   0:00.84 init

```

Figure 16-3. You can see at a glance how many zombie processes are on your system by looking at the top right of `top`'s display.

Note No magic is involved in killing processes. All that happens is that `top` sends them a “terminate” signal. In other words, it contacts them and asks them to terminate. By default, all processes are designed to listen for commands such as this; it’s part and parcel of how programs work under Linux. When a program is described as *crashed*, it means that the user is unable to use the program itself to issue the terminate command (such as `Quit`). A crashed program might not be taking input, but its *processes* will probably still be running.

In many cases, zombie processes simply won’t go away. When this happens, you have two options. The first is to restart the program that is likely to be the zombie’s owner, in the hope that it will reattach with the zombie, and then quit the program. With any luck, it will take the zombie child with it this time. Alternatively, you can simply reboot your PC. But it’s important to note that zombie processes are harmless and can be left in peace on your system!

Using Other Commands to Control Processes

You don’t always need to use `top` to control processes. A range of quick and cheerful shell commands can diagnose and treat process problems.

The first of these is the `ps` command. This stands for Process Status and will report a list of currently running processes on your system. This command is normally used with the `-aux` options:

```
ps -aux
```

This will return a list something like what you see when you run `top`.

If you can spot the problematic process, look for its PID and issue the following command:

```
kill <PID number>
```

For example, to kill a process with a PID of 5122, you would type this:

```
kill 5122
```

If, after this, you find the process isn't killed, then you should use the `top` program, as described in the previous sections, because it allows for a more in-depth investigation.

Another handy process-killing command lets you use the actual process name. The `killall` command is handy if you already know from past experience what a program's process is called. For example, to kill the process called `firefox-bin`, which is the chief process of the Firefox web browser, you would use the following command:

```
killall firefox-bin
```

Caution Make sure you're as specific as possible when using the `killall` command. Issuing a command like `killall bin` will kill all processes that might have the word `bin` in them!

CLEARING UP CRASHES

Sometimes, a crashed process can cause all kinds of problems. The shell you're working at may stop working, or the GUI itself might stop working properly.

In cases like this, it's important to remember that you can have more than one instance of the command-line shell up and running at any one time. For example, if a process crashes and locks up GNOME Terminal, simply start a new instance of GNOME Terminal (Applications ► Accessories ► Terminal). Then use `top` within the new window to kill the process that is causing trouble for the other terminal window.

If the crashed program affects the entire GUI, you can switch to a virtual console by pressing `Ctrl+Alt+F1`. Although the GUI disappears, you will not have killed it, and no programs will stop running. Instead, you've simply moved the GUI to the background while a shell console takes over the screen. Then you can use the virtual console to run `top` and attempt to kill the process that is causing all the problems. When you're ready, you can switch back to the GUI by pressing `Ctrl+Alt+F7`.

If you know the name of the program that's crashed, a quick way of getting rid of it is to use the `pgrep` command. This searches the list of processes for the program name you specify and then outputs the PID number. So if, say, Nautilus had frozen, you could type `pgrep nautilus`. Then you would use the `kill` command with the PID number that's returned.

Controlling Jobs

Whenever you start a program at the shell, it's assigned a job number. *Jobs* are quite separate from processes and are designed primarily for users to understand what programs are running on the system.

You can see which jobs are running at any one time by typing the following at the shell prompt:

```
jobs
```

When you run a program, it usually takes over the shell in some way and stops you from doing anything until it's finished what it's doing. However, it doesn't have to be this way. Adding an ampersand symbol (&) after the command will cause it to run in the background. This is not much use for commands that require user input, such as `vim` or `top`, but it can be very handy for commands that churn away until they're completed.

For example, suppose that you want to decompress a large zip file. For this, you can use the `unzip` command. As with Windows, decompressing large zip files can take a lot of time, during which time the shell would effectively be unusable. However, you can type the following to retain use of the shell:

```
unzip myfile.zip &
```

When you do this, you'll see something similar to the following, although the four-digit number will be different:

```
[1] 7483
```

This tells you that `unzip` is running in the background and has been given job number 1. It also has been given process number 7483 (although bear in mind that when some programs start, they instantly kick off other processes and terminate the one they're currently running, so this won't necessarily be accurate).

■ Tip If you've ever tried to run a GUI program from the shell, you might have realized that the shell is inaccessible while it's running. Once you quit the GUI program, the control of the shell will be returned to you. By specifying that the program should run in the background with the & (ampersand symbol), you can run the GUI program and still be able to type away and run other commands.

You can send several jobs to the background, and each one will be given a different job number. In this case, when you wish to switch to a running job, you can type its number. For example, the following command will switch you to the background job assigned the number 3:

```
%3
```

You can exit a job that is currently running by pressing Ctrl+Z. It will still be there in the background, but it won't be running (officially, it's said to be *sleeping*). To restart it, you can switch back to it, as just described. Alternatively, you can restart it but still keep it in the background. For example, to restart job 2 in the background, leaving the shell prompt free for you to enter other commands, type the following:

```
%2 &
```

You can bring the command in the background into the foreground by typing the following:

```
fg
```

When a background job has finished, something like the following will appear at the shell:

```
[1]+  Done          unzip myfile.zip
```

Using jobs within the shell can be a good way of managing your workload. For example, you can move programs into the background temporarily while you get on with something else. If you're editing a file in `vim`, you can press Ctrl+Z to stop the program. It will remain in the background, and you'll be returned to the shell, where you can type other commands. You can then resume `vim` later on by typing `fg` or typing `%` followed by its job number.

Tip Also useful is Ctrl+C, which will kill a job that's currently running. For example, if you previously started the `unzip` command in the foreground, hitting Ctrl+C will immediately terminate it. Ctrl+C is useful if you accidentally start commands that take an unexpectedly long time to complete.

NOHUP

What if you want to start a command running in a terminal window, but then want to close that terminal window? As soon as you close the window, any processes started within it are also closed. Try this now—type `gcalc` at the prompt to start the Calculator application and then quit the terminal window.

To get around this, you can use the `nohup` command. This stands for “no hangup,” and in simple terms, it tells the command you specify to stick around, even after the process that started it has ended (technically, the command is told to ignore the `SIGHUP` signal). However, commands run via `nohup` can still be killed.

To use `nohup`, simply add it before the command, for example:

```
nohup unzip myfile.zip
```

If the command requires `sudo` or `gksu` powers, add either of these after the `nohup` command.

Any command output (including error messages) is sent to the file `nohup.out`, which you can then view in a text editor. Note that if you run a command via `nohup` using `sudo` or `gksu`, the `nohup.out` file will have root privileges. If that's the case, you will also have to delete the `nohup.out` file via `sudo` before you can use `nohup` again as an ordinary user, because otherwise, `nohup` will be unable to overwrite the root-owned `nohup.out`.

Summary

This chapter has covered taking complete control of your system. We've looked at what processes are, how they're separate from programs, and how they can be controlled or viewed using programs such as `top` and `ps`. In addition, we explored job management under `BASH`. You saw that you can stop, start, and pause programs at your convenience.

In the next chapter, we'll take a look at several tricks and techniques that you can use with the `BASH` shell to finely hone your command-line skills.