A veteran Unix and Linux enthusiast, Chris Brown has written and delivered open source training from New Delhi to San Francisco, though not on the same day.

CORE TECHNOLOGY

Dive under the skin of your Linux system to find out what really makes it tick.

Processes

All the world's a Unix filesystem/and all the processes merely players...

You're undoubtedly aware that Linux, like most modern operating systems, can run many applications at the same time, through a feature called multi-processing, or multi-tasking. I can easily count the number of processes that my machine is running with a command such as:

```
ps -e | wc -l
```

The answer turns out to be 175, which seems quite a lot for a humble laptop that's basically just editing a text file.

So how do all these processes come into being? Well, some of them are started at boot time. These are the system services (usually referred to as "daemons"), and many of them may continue to run indefinitely, until the system is shut down. Some services (such as the Apache web server) maintain a pool of processes to enable them to promptly service multiple concurrent clients. Quite a lot of processes come into existence when the desktop is started. Of the 175 processes I counted earlier, rather more than 60 are there to give life to my desktop or to run the applications that are active there. Finally, some processes are started by my own actions when I launch an application from a menu or dahsboard, or I type a command at a shell prompt.

A process is sometimes defined as "an instance of a program in execution". The analogy of an actor reading from the script of a play may be useful – the script represents the program: the passive list of instructions of what's to be done. The actor represents the process, the active entity responsible for carrying out those instructions. But perhaps the best way to get a handle on what a process is is to delve more deeply into the resources and information it carries. Take a look at the table, below.

Everything you know is wrong

Even if you're not a programmer, you'll be used to some of these concepts. Take the notion of a current directory for instance (as reported by the **pwd** command). You think of yourself as being "in" a particular directory. But really, it's the process that's running your shell that holds the concept of current directory. You're probably also aware that you're running as a specific user, as reported by the **id** command. But again, it's really the process that's running your shell that maintains the notion of user identity. This is the identity that's used when access control checks are made – am I allowed to write to such-and-such a file, for example.

Although there are many circumstances under which a new process is started, there is only one way to actually do it, and that's for a process to create a copy of itself. This operation is called forking. The new process is called the child and the original process is called the parent. The child shares the code segment with the parent... that is, it is executing the same program. It receives a copy of the data segment, the stack, and the heap. (The reality is actually a bit more complicated. These memory regions are not copied piecemeal, but the "copy-on-write" operation of the virtual memory system makes things behave as if they were.) The child also inherits many other things – user ID, current directory and so on – from the parent. In fact, the child begins life essentially as an exact clone of the parent except for one small thing – it has a different process ID. Returning to our actor/script analogy, forking is roughly analogous to our original (parent) actor calling out to the

Information carried by a process

Item	Description
Process ID (PID)	Unique integer identifying the process
Code segment	Memory region that holds the program's executable instructions
Data segment	Memory region that holds statically defined data
Stack	Expandable memory region used to store 'local' function variables
Heap	Expandable memory region used to store dynamically allocated objects
Priority	Determines how favoured the process will be by the scheduler. Derived from the 'nice value
Signal disposition	The way that the process responds to the receipts of signals – are they caught, ignored, etc.
Environment	A list of environment variables of the form NAME=VALUE
File descriptors	'Handles' on streams that are available for reading or writing (includes stdin, stdout and stderr)
UID	Read user identity
EUID	Effective user identity
CWD	Current working directory

Try It Out

Compile C code

To compile the C examples in this tutorial you'll need to install the C compiler (GCC). On a Debian-style system use this command:

```
# apt-get install gcc
```

and on a Red Hat-style system:

```
# yum install gcc
```

Now create a file called **forkdemo.c** with content as follows:

```
#include <stdio.h>
void main()
{
  int i;
  if (fork()) {
    for (i=0; i<10000; i++)
      printf("**  PARENT %d\n", i);
  } else {
    for (i=0; i<10000; i++)
      printf("  **CHILD  %d\n", i);
  }
}
```

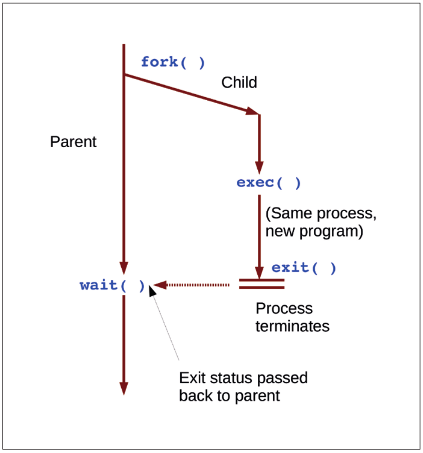
To compile the program, run the command:

```
$ gcc -o forkdemo forkdemo.c
```

and assuming you don't get any compilation errors, run it like this...

```
$ ./forkdemo
```

The thing to notice here is that the output from the parent and the child is interleaved in a non-deterministic way, as the scheduler switches back and forth between the processes. It's also possible (rather likely in fact) that some of the CHILD messages will appear AFTER the shell's next prompt.


fork( )
Child
Parent
exec( )
(Same process, new program)
exit( )
wait( )
Process terminates
Exit status passed back to parent

The classic fork/exec/exit/wait model. This is what happens every time you run a command.

wings for a new actor, who trots out onto the stage and stands by the side of the first actor, sharing a copy of the script.

Scary C code

To see how this really works I'm going to inflict a few lines of C code on you. The action here centres around the **fork()** system call. It's a simple enough call -- it takes no arguments and returns just an integer result – but of all the system calls in Linux it's perhaps the one that is hardest to get to grips with, because although only one process makes the call, two processes return from it – the original parent and the new child. They figure out which is which by examining the return value from the call. In the parent, **fork()** returns the PID of the newly-created child. In the child, it returns zero. So you invariably see the **fork()** call wrapped inside an **if** test. Keeping in mind that in C any non-zero integer value counts as "true", and zero counts as "false", the code might look something like the boxout above.

Sometimes, parent and child continue to execute the same program. This happens, for example, when Apache spawns its pool of "spare" server processes. More often, though, the child will start to execute a different program. This is what happens, for example, when I type a command into the shell. (I'm talking about running a command that lives in an external executable file, not a built-in shell command such as **echo** or **cd**.) A process runs a new program by executing an **exec()** system call. In doing this, the code, date, stack and heap of the old program are discarded; however, the process retains its process ID, its current directory, its open file descriptors, and (usually) its environment. To return to our actor/script analogy, an **exec()** is akin to the actor discarding his current script, picking up a copy of Macbeth, and starting at the beginning. He remains the same actor, but he's performing a different play. There are, in fact, six versions of the **exec** call, but I will not burden you with the minutiae of their differences. The call is unusual in that it doesn't return except in the case that it fails (usually because it can't find the executable).

Two other system calls are important for management of processes – **exit()** and **wait()**. The **exit()** call is simple enough – it terminates the process. You can pass an integer argument **to exit()**, which is passed back to the parent process and is known as the "exit status". The convention is that an exit status of zero indicates a normal, successful termination, and non-zero values indicate failure of some sort.

For programs started from a shell prompt, you can find the exit status of the last command through the variable **$?**. For example,

```
$ grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
$ echo $?
0
$ grep xyzzy /etc/passwd
$ echo $?
1
$ grep root /etc/xyzzy
grep: /etc/xyzzy: No such file or directory
$ echo $?
2
```

Here we see that **grep** returns an exit status of 1 if it doesn't find the pattern and 2 if it can't find the input file.

The **wait()** call is used by a parent process to wait until its child process (or one of its children) terminates. This call can also be used to retrieve the exit status of the child.

Using fork(), exec(), exit() and wait() we can write a minimal shell:

Try It Out

Write a shell

Create a file called **tinyshell.c** with this content:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
/* A minimal shell */
main()
{
  char line[100];
  /* Main command loop */
  while (printf("> "), gets(line) != NULL) {
    if (fork() == 0) { /* Child */
      execlp(line, line, (char *)0);
      /* Don't come here unless execlp fails */
      printf("%s: not found\n", line);
      exit(1);
    }
    else  wait(0); /* Parent */
  }
}
```

Now compile and run it something like this:

```
$ gcc -o tinyshell tinyshell.c
$ ./tinyshell
> date
Wed Mar 19 15:25:35 GMT 2014
> hostname
ubuntu1204
> who
chris   tty7      2014-03-19 13:52
chris   pts/0     2014-03-19 14:05 (:0)
> ls -l
ls -l: not found
```

Try It Out

Braaaaaains!
It's rather easy to make zombies. Create a file called **makezombies.c** with content as follows:

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
  int i;
  for (i=0; i<5; i++)
    if (fork() == 0)
      exit(0); /* Child */
  sleep(1000);
}
```

Then compile it and run it in the background:

```
$ gcc -o makezombies makezombies.c
$ ./makezombies &
```

Now if you run ps you'll see a parent and five zombie children:

```
$ ps
  PID TTY        TIME CMD
 5982 pts/0    00:00:00 makezombies
 5983 pts/0    00:00:00 makezombies <defunct>
 5984 pts/0    00:00:00 makezombies <defunct>
 5985 pts/0    00:00:00 makezombies <defunct>
 5986 pts/0    00:00:00 makezombies <defunct>
 5987 pts/0    00:00:00 makezombies <defunct>
 5988 pts/0    00:00:00 ps
32052 pts/0    00:00:00 bash
```

The point of this example is that the parent creates five children that immediately exit. Rather than waiting for them, the parent enters a 1000-second sleep. You cannot kill the zombies in the usual way. For example, the command

```
$ kill 5987
```

has no effect.

The trick to getting rid of zombies is to track down and kill the parent, which is easy to find in this case:

```
$ kill 5982
```

When the parent dies, the zombie children are inherited by init (PID=1) which will collect the exit status from each of them, allowing them to be finally laid to rest.

You'll notice that the last example fails – the shell doesn't understand about command options and tries to find a file called **ls -l**. Well, honestly, what do you expect from 12 lines of code? But despite its simplicity, this little program illustrates the essence of what happens when I run a command in the shell.

With this model in mind, it's easy to see how the shell implements background jobs (when the command line ends with **&**). The shell simply doesn't do the **wait()** before it prompts for the next command.

Occasionally, things go wrong with this model. If a process exits, and its parent is not waiting for it, the child enters a "defunct" state (also known as a "zombie"). The child cannot be completely laid to rest because it needs a waiting parent to pass its exit status back to. Here's an analogy. Little Jimmy, who's only six, is usually met by his mother after school. She waits for him by the school gate and Jimmy is always excited to see her. One day, mum gets held up in traffic and isn't there when Johnny comes out of school. He has no-one to report his exit status to, and enters a zombie state. (Of course, analogies should not be pushed too far!)

Let's move away from C code and take a look at processes from the command line. The classic command for viewing processes is **ps**. Other commands you might find helpful are **pstree**, which shows the process ancestry using simple ASCII art, and **pgrep**, which reports the process ID(s) of the processes that are running a specified program, or have a specified owner or a specified parent, and so on.

A figment of the kernel's imagination

Commands such as **ps** garner most of their information from the **proc** filesystem, which is usually mounted on the **/proc** directory. Through this filesystem the kernel exposes a large amount of process information in the form of what it calls 'files'. They are not files as we normally think of them; they are an illusion provided by the kernel to give us a file-like view of some of its internal data structures. You can tell there's something funny going on because the command

```
$ ls -l /proc
```

shows most of the 'files' as having zero length, but if you read from them there is content. Try this:

```
$ cat /proc/partitions
```

for example. A directory listing of **/proc** will reveal a number of subdirectories named directly after the process IDs. In these subdirectories you'll see a standard set of 'files' that expose per-process information. Let's change to the directory corresponding to the process running my **rsyslog** daemon:

```
$ pgrep rsyslogd
527
$ cd /proc/527
```

Much of this information you'll find here is very low-level. Try this:

```
$ cat status
```

You'll see lots of memory usage information and stuff about signal dispositions that is downright inscrutable. Usually, the output from programs such as **ps** and **lsof** will be easier to scrute (is that a word?). Let's delve a little deeper though. The subdirectory "fd" contains symbolic links that are named after the process's open file descriptors:

```
$ sudo ls -l fd
total 0
lrwx------ 1 root root 64 Mar 21 14:40 0 ->
socket:[7307]
l-wx------ 1 root root 64 Mar 21 14:40 1 -> /var/log/
syslog
l-wx------ 1 root root 64 Mar 21 14:40 2 -> /var/log/
mail.log
lrwx------ 1 root root 64 Mar 21 14:40 3 ->
socket:[7309]
lr-x------ 1 root root 64 Mar 21 14:40 4 -> /proc/
kmsg
l-wx------ 1 root root 64 Mar 21 14:40 5 -> /var/log/
auth.log
l-wx------ 1 root root 64 Mar 21 14:40 6 -> /var/log/
kern.log
```

We see here that file descriptor 1 (for example) is connected to **/var/log/syslog**.

The environment

One important piece of baggage that a process carries around with it is the environment, which is basically just a list of strings, each of the form **NAME=VALUE**. A child process inherits its environment from its parent after a **fork()** and the environment usually remains intact across an **exec()**, too, depending on which version of the **exec()** call is used. From your command prompt

Threads

Linux allows multiple concurrent paths of execution within the address space of a single process. Some people refer to threads as "lightweight processes" because they carry a lot less context around with them than regular processes, and are much cheaper to create. Threads exist within a process; they rely on the process to carry around most of the execution context. By default a process has only one execution thread. In a more thread-oriented view of the world, you could consider the process as being the passive holder of context, and the thread as being the active entity that gets stuff done.

Here's an analogy. Sue (the process) takes her three children (the threads) on a picnic. She struggles up the hill to a nice spot overlooking the river carrying a blanket, a food hamper, and some folding chairs. She sets them all out on the ground. The kids, meanwhile, just have a good time running around. They are not laden with hampers or blankets – they carry with them almost no context of their own.

you can examine the environment of your shell's process with the **env** command:

```
$ env
SSH_AGENT_PID=31487
TERM=xterm
SHELL=/bin/bash
USER=chris
LANG=en_GB.UTF-8
HOME=/home/chris
```

The list I've shown here is vastly stripped down. As they are inherited, environment variables can be used to pass configuration information to applications. Examples include **EDITOR**, (which tells commands like **crontab** and the command line mail program which editor to use) **DISPLAY** (which tells graphical programs where their X server is) and **CLASSPATH**, which tells Java apps where to look for their class files. Environment variables are commonly set in shell startup files such as **/etc/profile**.

As a final thought

There's quite a bit of interest at the moment in Linux Containers, which is finally reaching some level of maturity. In his excellent feature about them in LV002, Jonathan Roberts wrote: "The idea of containers is to make it easy to run multiple applications on a single host, all the while ensuring each remains separate."

But you can do that with processes too! In fact we can see threads and processes as one end of a spectrum of containment technologies, with chroot jails, containers and full-blown virtualisation at the other end. As we move across the spectrum, less is shared and more is separate. For example, processes all share the same TCP/IP port space – I can't have two processes binding to port 22. Each container, on the other hand, provides a separate port space.

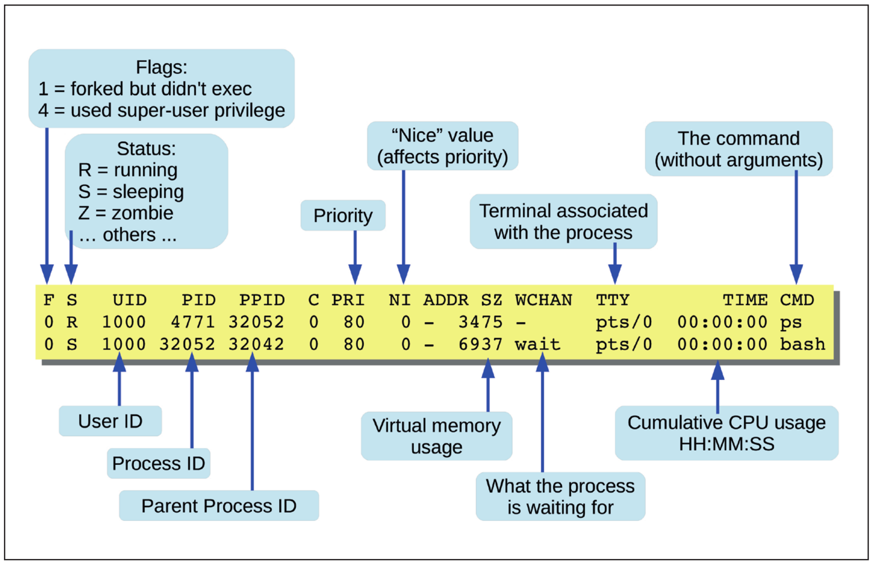
Command of the month: ps

ps is the classic command for listing the processes running on the machine. Used without options, it gives you a minimum of information about the processes associated with your terminal:

```
$ ps
 PID TTY          TIME CMD
 352 pts/0        00:00:00 ps
32052 pts/0       00:00:00 bash
```

Typically, this output will show only the **ps** command itself, and its parent shell (**bash**). How come the PID for the **ps** command (352) is less than the PID of its parent (32052)? That's because after PIDs have reached a specified limit (by default it's 32768) the numbers wrap around and the PIDs get recycled. I didn't deliberately engineer that result, it just happened to come out that way.

ps has a large and very confusing set of options, partly because it's trying to remain backwards compatible with both of its progenitors (the BSD and System V versions) and partly because, well, it's just got a lot of options. Some of them control which processes are listed; some control how much detail is shown. I suspect that most people just remember two or three combinations of options that they find useful, and stick with those.


```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY        TIME CMD
0 R  1000  4771 32052  0  80   0 -  3475 -      pts/0  00:00:00 ps
0 S  1000 32052 32042  0  80   0 -  6937 wait   pts/0  00:00:00 bash
```
Flags:
1 = forked but didn't exec
4 = used super-user privilege
Status:
R = running
S = sleeping
Z = zombie
... others ...
Priority
"Nice" value (affects priority)
Terminal associated with the process
The command (without arguments)
User ID
Process ID
Parent Process ID
Virtual memory usage
What the process is waiting for
Cumulative CPU usage HH:MM:SS

The **ps** command sure generates a lot of output. But what does it all mean?

ps in depth

The **-f** option gives a full listing:

```
$ ps -f
UID      PID  PPID  C STIME TTY        TIME CMD
chris   4770 32052  0 12:33 pts/0  00:00:00 ps -f
chris  32052 32042  0 Mar19 pts/0  00:00:00 bash
```

...and the **-l** option gives even more (rather like the **-l** option of **ls**):

```
$ ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY
TIME CMD
0 R  1000  4771 32052  0  80   0 -  3475 -      pts/0
00:00:00 ps
0 S  1000 32052 32042  0  80   0 -  6937 wait   pts/0
00:00:00 bash
```

The divided heritage of **ps** causes confusion. For example the two commands

```
$ ps -elf
$ ps aux
```

produce approximately the same output. The first uses System V syntax, the second uses BSD syntax (note the absence of a hyphen). They do not, however, show exactly the same output fields. I suspect that most administrators memorise two or three useful combinations of options and stick with those. That's what I do!

If you want detailed control over the output fields use the **-o** option. This example shows the process ID, group ID, and command line:

```
$ ps -o pid,gid,cmd

 PID  GID CMD
 6260 1000 ps -o pid,gid,cmd
32052 1000 bash
```

There are LOTS of output columns you can select here; see the STANDARD FORMAT SPECIFIERS section of the man page for the details.

The **-e** option lists every process, but you can also get finer control over which processes are listed. For example,

```
$ ps --ppid 1
```

shows those processes whose parent is PID 1 (note the use of the GNU-style command option with the double hyphen!), or to show just the processes running as **chris**:

```
$ ps -u chris
```