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

Prise the back off Linux and find out what really makes it tick.

Signals

Get the attention of running processes by sending them signals.

In the world of Linux system programming, a signal is an event that's delivered to a process by the kernel. A signal says to the process "something has happened that you might want to respond to". A few signals are generated as a result of something that the program itself is doing (usually something bad), but most of them originate from sources external to the program itself.

Why do you need to know about signals? Well, they're important to a system administrator because they provide a way to interact with running processes (in particular, to kill them). And the most important single reason that a developer needs to be aware of signals is so that he knows how to write programs that ignore them. But there are more useful things you can do with signals, as we'll see.

There are several different types of signal. If you're running a *Bash* shell, the built-in command **kill -l** will show you a list of them. It's a slightly scary list but you don't need to know about most of them, and here we're going to focus on the 10 or so you're most likely to use.

SIGHUP

This signal has an interesting history. The "HUP" stands for "hang up" and it harks back to the days when telephones hung on a hook on the wall and you would terminate a call by hanging up the phone. The scenario went like this: Dennis was logged into his PDP11 computer via a dial-up line and a modem. Without logging off, he simply "hung up" the connection. Later, Ken dialled in to the same modem, thus finding himself connected to Dennis's abandoned shell. To prevent this undesirable state of affairs, the device driver for the modem would detect the loss of carrier when Dennis hung up and deliver a SIGHUP signal to terminate the shell session.

Well, dial-up logins are history now, and SIGHUP was looking forward to a peaceful retirement when it was offered a new job. Nowadays, SIGHUP is interpreted by some daemons to mean "your configuration file has been changed, please go and re-read it". One example is the system logging daemon (**syslog** or **rsyslog**) which re-reads the config file **/etc/syslog.conf** (or **/etc/rsyslog.conf**) on startup and on receipt of a SIGHUP. In some cases the daemon simply stops and restarts when it receives this signal.

SIGINT

This is the signal that is sent to a foreground process by the terminal driver when you enter ^C on the terminal. By default, programs will terminate when they receive this signal. Some programs, especially ones that operate interactively, choose to ignore this signal.

SIGTERM

This is conventionally used as a polite "please tidy up and terminate" request. For example, when you shut down a Linux system with the **shutdown** command, it begins by sending SIGTERM to all the running processes in the hope that they will do the decent thing and go away. If this doesn't work, **shutdown** waits for a few seconds then sends a SIGKILL. SIGTERM is the default signal type sent when you use programs like **kill** and **pkill**.

SIGKILL

This is the most brutal signal because a process cannot choose to catch or ignore it. A process receiving SIGKILL is instantly terminated. Best practice suggests that as a way of killing a process it should be a last resort, when more polite requests such as SIGTERM have failed. This is particularly true for services that maintain lock files or other temporary data files, because they won't have opportunity to clean them up and you may end up having to manually remove them before the service will restart.

SIGALRM

This signal is "self-inflicted" – it's generated as a result of an alarm clock timing out. Typically a C program might request an alarm call 10 seconds from now with:

```
alarm(10);
```

and use it to implement a timeout on a potentially blocking operation.

Setting your interrupt character

The terminal driver (the code inside the kernel that's reading characters from your keyboard) recognises a number of characters that are handled specially. Well-known examples include the "interrupt" character (usually ^C) which sends a SIGINT to any foreground processes running on that terminal, and the "end-of-file" character (usually ^D) which tells a program that's reading its standard input from the keyboard that there is no more data. You can see all of these settings with:

```
$ stty -a
```

Although most of what you'll see here harks back to the days of terminals that plugged into serial ports and is not relevant now, you can also change them. For example, to set your interrupt character to ^X:

```
$ stty intr '^X'
```

The control character is entered here as two characters, a caret (^) then the X.

SIGSEGV

This signal is generated by the kernel when a process tries to access a memory address that's outside its address space. Of course this should never happen in a correctly written program; typically it occurs in C code that makes a reference through a pointer that hasn't been initialised, as this two-liner demonstrates:

```
void main()
{
  int *p;
  *p = 0;
}
```

Assuming the code's in the file **segvdemo.c**, compile and run it like this:

```
$ gcc -o segvdemo segvdemo.c
$ ./segvdemo
Segmentation fault (core dumped)
$ echo $?
139
```

SIGILL

Another signal that arises directly from the execution of the process. It indicates an illegal instruction, and should never occur unless your compiler is buggy or the executable has become corrupt, or maybe because it calls a function through an unitialised pointer.

SIGBUS and SIGFPE

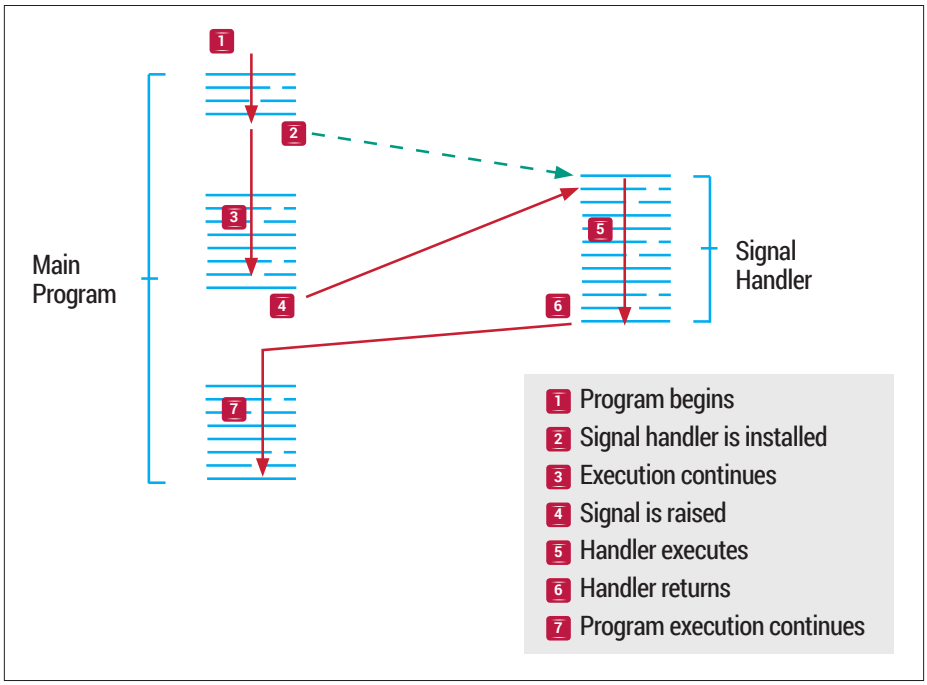
These indicate an incorrectly aligned memory access, and a floating point exception (or other arithmetic error) respectively. It's easy to deliberately generate a SIGFPE – just divide by zero:

```
void main()
{
 int a, b, c;
 a = 1; b = 0;
 c = a / b;
}
```

If you compile and run this program, you'll see something like this:

Signals are not exception handling

Some languages support exception handling, typically with keywords like "try", "catch" and "throw". For example, if you try to open a file for writing and don't have write permission, in some environments the runtime will throw an exception that you can choose to catch in order to handle the error. We mention this because this is NOT what signals do. You can (to a very limited extent) install exception handling of a sort by catching signals like SIGFPE, but failed system calls and library routine calls do not throw exceptions; they return -1 (or sometimes a null pointer) to indicate failure and you need to explicitly test the return value to detect this.



Arrival of a signal interrupts the execution of the main program and runs the handler.

```
$ ./fpedemo
Floating point exception (core dumped)
$ echo $?
136
```

Again, notice the exit status (136). Subtracting 128 gives 8, the signal number of SIGFPE.

SIGABRT

A self-generated signal raised when a program calls the **abort()** library function. By default it will cause immediate termination of the program.

Sending signals

OK, we've discussed some of the signal types. We've seen that some, such as SIGSEGV and SIGFPE, are raised automatically by the kernel as the result of some misdemeanour committed by the program. These are sometimes referred to as "synchronous" signals. But others need to be explicitly generated from outside the program (sometimes called "asynchronous" signals). How do we do that?

One way is to use the command **kill**. It's not a good name really; **raise** or **sendsignal** might be better. For example, we can send a SIGHUP signal to process 12345 like this:

```
$ kill -SIGHUP 12345
```

Or you can use the short signal name, or the signal number, like this:

```
$ kill -HUP 12345
$ kill -1 12345
```

This is a good time to point out that you can only send signals to processes that you own, unless you're running as root in which case you can deliver signals to any process.

Sending SIGHUP manually like this is commonly used to signal a service after changing its config file. Manually generated signals are also often used to terminate a "hung" process (or just one that seems to have been running for far too long), typically like this:

```
$ kill -TERM 12345
or more brutally:
$ kill -KILL 12345
```

If you don't specify a signal type, the default is SIGTERM.

As you'll see from these example you need to know the process ID to send a signal. If you're trying to kill a program called **foobar** you might get this by running:

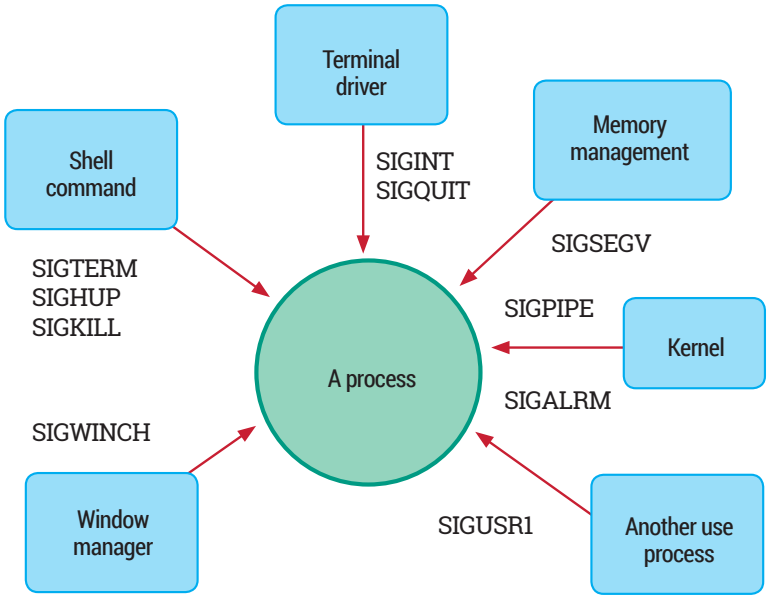
```
$ ps -ef | grep foobar
chris    4923  2586  0 18:07 pts/0    00:00:00 ./
foobar
chris    4968  4924  0 18:07 pts/6    00:00:00 grep
foobar
```

from which we see that the PID is 4923. (Ignore the "false positive" generated from the **grep** command.)

Sending signals from a program

So much for sending signals from the command line. You can also send signals from within a program. Here's a little C program I wrote called "terminate"; the idea is that you give it a PID as an argument and it begins by sending a polite SIGTERM signal to ask the process to terminate. If this doesn't work it just pulls out a gun and

Signal sources



The kernel delivers all signals, but different signal types typically originate from different places.

shoots the process in the head with SIGKILL. Note that the line numbers are for reference, they are not part of the file:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <signal.h>
4. #include <errno.h>
6. int main(int argc, char* argv[])
7. {
8.   int targetpid, i;
9.
10.  /* Get target process ID from command line */
11.  targetpid = atoi(argv[1]);
12.
13.  /* Check that the process exists */
14.  if (kill(targetpid, 0) < 0)
15.  {
16.    switch (errno)
17.    {
18.      case ESRCH:
19.        printf("Process %d does not exist\n",
targetpid);
20.        exit(1);
21.      case EPERM:
22.        printf("Do not have permission to terminate
that process\n");
23.        exit(2);
24.    }
25.  }
26.
27.  /* Ask the process to terminate (politely) */
28.  kill(targetpid, SIGTERM);
29.
30.  /* Wait for up to 5 seconds for the process to
die */
31.  for (i = 0; i < 5; i++)
32.  {
33.    if (kill(targetpid, 0) < 0)
34.      exit(0);
35.      sleep(1);
36.  }
37.
38.  /* Asking nicely didn't work, bring out the big
guns */
39.  printf("SIGTERM ineffective, sending
SIGKILL\n");
40.  kill(targetpid, SIGKILL);
41.  exit(3);
42. }
```

If you don't read "C", here's a guided tour:

At line 11 we grab the process ID from the command line. There should really be some error checking here to verify that the user did actually supply a PID as argument. At line 14 we try to send signal number 0 to the process. The **kill()** system call is analogous to the **kill** command, though notice that the arguments are in the opposite order. (Hey, this is Linux! You want consistency?) Now there is no signal number 0, so the call will not actually deliver a signal to the process, but it will fail (returning -1) if either the process doesn't exist, or we don't have permission to signal it (ie we don't own it and we're not root). These two conditions are trapped at lines 18 and 21, where we print an appropriate error message and exit.

If we make it as far as line 28, we know that the process exists and we have permission to signal it, so we send a polite SIGTERM to the process, hoping it will oblige and go away. Then the loop starting at line 31 repeatedly probes (sending the dummy signal 0 again) to see if the process has terminated. If it has, then fine, our job is done, and we exit at line 34. We continue for five seconds, probing at one-second intervals. Finally, if we reach line 39, we forcefully terminate the process with SIGKILL. This approach (SIGTERM followed if necessary by SIGKILL) is essentially what happens to all running processes during a system shutdown.

Catching signals

So now we know how to send signals. Let's look at the other side of the story – how does a process respond when it receives a signal? Well, each signal has a default disposition ("disposition" is just a posh word meaning "what will happen when a signal arrives"). The three dispositions shown in the table are:

1. **Term** The process is terminated (this is the most common behaviour).
2. **Core** The process is terminated, a memory image (core file) may be written.
3. **Ignore** The signal is ignored.

However – and here it gets interesting – a program can install handlers for the various signal types – pieces of code that will run if the signal arrives.

Rather than do this in C again, we'll do it in a shell script. The purpose of this script is to count the number of prime numbers less than one million.

Now of course, doing a computation-rich thing like this in a shell script is pretty stupid, and I'm not using the most efficient algorithm either, which doesn't help. But that's not the point. The point of this example is that it represents a long-running program that gradually works its way through a data set. Here's the script:

```
1. #!/bin/bash
2.
3. function isprime()
4. {
```

Signals and exit codes

When a process terminates "normally" (by executing an **exit()**), it chooses an exit code to pass back to the parent – 0 to indicate success and a small integer to indicate some sort of failure. But if the process is terminated by a signal it doesn't get a choice. In this case, the exit status will be 128 plus the number of the signal that killed it. So for example a process killed by a SIGKILL (signal 9) will have exit status 137 (128+9).

```
5.  n=$1
6.  factor=2
7.  while (( factor * factor <= n ))
8.  do
9.    if (( n % factor == 0 ))
10.   then
11.     return 1 # number is not prime
12.   fi
13.   (( factor++ ))
14. done
15. return 0 # no factors, number is prime
16. }
17.
18. trap 'echo Testing value $val, found $count primes so far' HUP
19. trap 'echo Buzz off I am busy counting primes!' TERM
20. trap '' INT
21.
22. count=2
23. val=5
24. while (( val < 1000000 ))
25. do
26.   if isprime $val
27.   then
28.     (( count++ ))
29.   fi
30.   (( val += 2 ))
31. done
32. echo count is $count
```

Let's walk you through this. Lines 3 to 16 define a function called **isprime**. It takes the number we want to test as an argument, and returns 0 (success) if the number is prime and 1 (failure) if it isn't. The code is not difficult, but its details do not concern us here. The script really starts at line 22. We enter a loop (lines 24 to 31), testing all odd numbers between 5 and 1,000,000 for prime-ness and counting them. (I do at least have the sense not to test even numbers.)

On exiting the loop we print out the answer (line 32).

If you want to try this out (and we hope you will) put the code into a file called **countprimes** and make it executable:

```
$ chmod u+x countprimes
```

Now run the script:

```
$ ./countprimes
```

It will take quite a while to run (17 minutes on my laptop). Meanwhile, go back and look at lines 18–20. These are the lines that install signal handlers for SIGHUP, SIGTERM, and SIGINT signals respectively. In these example we have written the signal-handling actions "in line", though we could also have written them as functions, which would be easier to deal with if we wanted the handler to do several things. The SIGHUP handler prints a progress report. The SIGTERM handler prints a rude message, but the program continues executing. The empty SIGINT handler at line 20 simply makes the script ignore SIGINT signals. Since ignoring signals is a common requirement, we'll allow ourselves one more line of C:

```
signal(SIGINT, SIG_IGN);
```

which says to ignore SIGINT signals and is equivalent to the **trap** statement at line 20 in the script.

So go back to the terminal where **countprimes** is running and enter '^C'. As we've seen, this will send a SIGINT signal to the process. If we didn't have line 20 in the script this would terminate the program, but now it is simply ignored and the program continues.

Now open a second terminal window. Enter the command:

```
$ pkill -HUP countprimes
Testing value 861877, found 68481 primes so far
```

As you'll see, the SIGHUP handler tells us how far we've got in our prime-counting task. Now try:

```
$ pkill countprimes
```

which sends the default SIGTERM signal and will elicit the response:

```
Buzz off I am busy counting primes!
```

but again, the program will continue. Finally (unless you are actually interested in knowing how many primes under 1,000,000 there really are and would like to allow the program to run to completion) we can forcefully terminate the program with:

```
$ pkill -KILL countprimes
```

We haven't installed a handler for SIGKILL, and we couldn't if we wanted to because you can't catch or ignore SIGKILL, so in the first terminal window you'll see the message

```
Killed
```

If you then examine the exit status in that terminal:

```
$ echo $?
137
```

you see that it's 137. Subtracting 128 as before gives 9, the signal number of SIGKILL.

That's all for this month. If you'd like to learn more, the man page for **signal** (**man 7 signal**) has a great deal more detail, but rapidly gets rather techie. There's also a good discussion in the GNU C Library manual at **www.gnu.org/software/libc/manual/html_node/Signal-Handling.html**. Happy signalling!

"This example represents a long-running program that gradually works its way through a data set."

Common signals

Signal name	Number	Default action	Description
SIGHUP	1	Term	Some daemons interpret this to mean "re-read your configuration"
SIGINT	2	Term	This signal is sent by ^C on the terminal
SIGTRAP	5	Core	Trace/breakpoint trap
SIGBUS	7	Core	Invalid memory access (bad alignment)
SIGFPE	8	Core	Arithmetic error such as divide by zero
SIGKILL	9	Term	Lethal signal, cannot be caught or ignored
SIGSEGV	11	Core	Invalid memory access (bad address)
SIGPIPE	13	Term	Write on a pipe with no one to read it
SIGALRM	14	Term	Expiry of alarm clock timer
SIGTERM	15	Term	Polite "please terminate" signal
SIGCHLD	17	Ignore	Child process has terminated

Each signal has a name, a number, a default "disposition" and a purpose.