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.

Pipes, shell scripting, and a little bit of C

The most fun you can have with a pipe without actually putting tobacco in it.

Nestled in the bottom-left corner of my laptop's keyboard is a key with two straight lines on it – a slopey one and a vertical one. The vertical line goes by various names – a stick, a poley or just "vertical bar". It finds use as the logical 'OR' operator in various languages, and is sometimes used in great numbers in ASCII art. But in the world of the Linux command line it is most often called a pipe.

Now we all know how to construct a pipeline on the command line, and (roughly) how it works. A simple command such as

```
ps -e | wc
```

runs the two programs **ps** and **wc** concurrently, with the output from **ps** appearing as the input of **wc**.

Pipes involve an upstream process (the producer of data) and a downstream process (the consumer), and the pipe imposes a loose synchronisation between the two. If the pipe becomes full, the upstream process will block if it tries to write; if the pipe is empty, the downstream process will block if it tries to read.

It is this simple mechanism for combining two programs, together with the large collection of programs that deal with input and output streams consisting simply of lines of text (in preference to binary formats) that facilitates the "tool building" approach that makes the command line so powerful.

Like last month, I'm going to inflict a few lines of 'C' code on you so that you can see how pipes work behind the scenes. This is (approximately) what the shell does if I enter the command **ls | sort -r** :

```
1. #include <unistd.h>
2.
3. void main()
4. {
5.   int p[2];
6.
7.   pipe(p);       /* Create the pipe */
8.   if (fork() == 0) {
9.     /* Downstream child: connect stdin to pipe */
10.    dup2(p[0], 0);
11.    close(p[1]);
12.    execlp("sort", "sort", "-r", (char *)0);
13.  }
14.  if (fork() == 0) {
15.    /* Upstream child: connect stdout to pipe */
16.    dup2(p[1], 1);
17.    close(p[0]);
18.    execlp("ls", "ls", (char *)0);
19.  }
20.  /* Parent: wait for both children */
21.  close(p[0]); close(p[1]);
22.  wait(); wait();
23. }
```

There isn't really that much code here, but it's perhaps not obvious what's going on, so let me explain:

Behind the scenes in the pipe factory

The system call at line 7 creates a pipe. We supply a little array of integers to this call (**p**) and in it we get back two file descriptors, one on the upstream end (**[p1]**) and one on the downstream end (**p[0]**). This corresponds to stage 1 of the diagram on page 65. At line 8 we create a child process which is destined to become the downstream program. (I discussed **fork()** in detail last month.) At line 10 we connect the standard input of this process (descriptor 0) to the downstream end of the pipe (**p[0]**). Then at line 12 this child program executes the **sort** program to do a reverse sort. We don't supply a filename argument, so (like any well-behaved filter program) **sort** will read from its standard input. The sequence at lines 14–19 is similar – we create the upstream child, connect its standard output to the upstream end of the pipe, and execute **ls**. Now we're at stage 3 of the diagram. All that is left for the parent to do (line 22) is to wait for its two children to finish.

Closing the unused descriptors on the upstream end of the pipe in the downstream child (line 11) and in the parent (line 21) is important. If we don't do this, the downstream child will never see an EOF ("End Of File") when it reads from the pipe, and will just hang there on the assumption that one of the processes with an open descriptor might write some more data. The first time I wrote code like this, which is a fair few years ago now, I remember being caught out by this. Notice that the are no

"Pipes involve an upstream process and a downstream process, and the pipe imposes a loose synchronisation between the two."

Try It Out

Find the named pipes on your system

```
sudo find / -type p
/var/spool/postfix/public/qmgr
/var/spool/postfix/public/pickup
```

You may get different results depending on what Linux distro you're running and what packages you have installed, but I would be willing to bet that you don't find very many.

Try It Out

Your very own named pipe

For this demonstration, set up two terminal windows side by side on your desktop. In one of the windows, begin by creating a named pipe. The command is **mkfifo** (and not, as you might reasonably guess, **mkpipe**):

```
$ mkfifo /tmp/demopipe
```

Verify that it exists:

```
$ ls -l /tmp/demopipe
```

```
prw-rw-r-- 1 chris chris 0 Apr  1 17:21 /tmp/demopipe
```

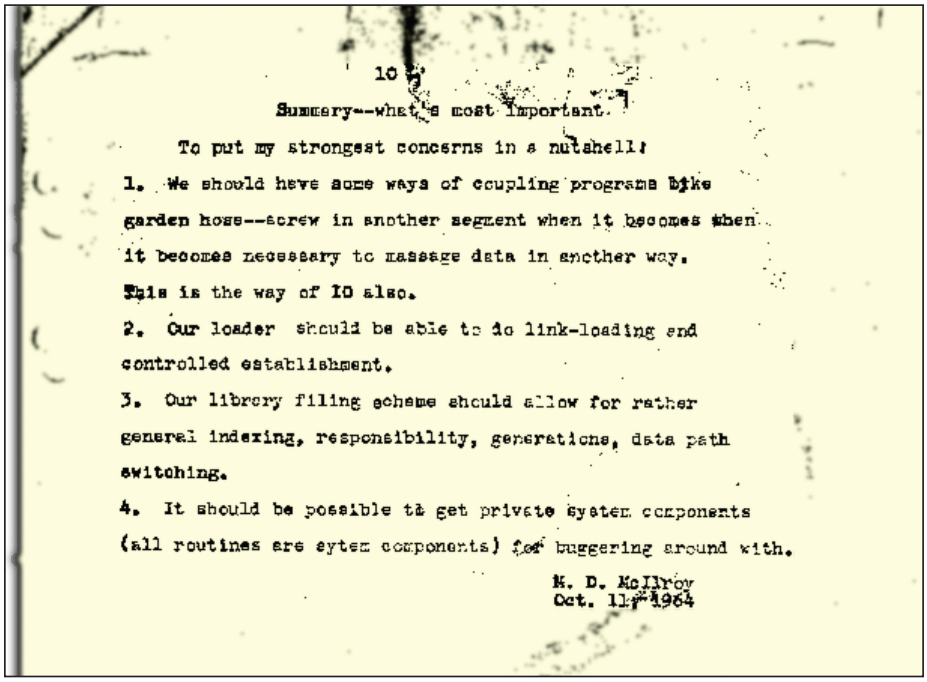
Now, in the left-hand window type

```
$ cat > /tmp/demopipe
```

and in the right-hand window type

```
$ cat < /tmp/demopipe
```

Now type a few lines of text into the left-hand window. What you type is written to the pipe by the "left" cat. In the right window you'll see the output of the "right" cat echoing what you typed onto the screen. To quit the experiment, enter Ctrl+D (your "end of file" character) into the left window. The left cat will terminate, closing the upstream end of the pipe. The right cat, as it tries to read from the pipe, notices that it has been closed, and exits.



10

Summary--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes then it becomes necessary to massage data in another way. This is the way of IO also.

2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

4. It should be possible to get private system components (all routines are sytem components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964

Pipes in the making: a note from one of the founding fathers of UNIX, Doug McIlroy, dated October 1964, suggests "ways of coupling programs like garden hose".

read() or **write()** calls corresponding to stage 4 of the diagram in the code presented here; the reading and writing goes on in the child processes, which simply access their standard input or standard output streams. They are not aware that these streams are connected to a pipe.

This code is so quintessentially UNIX that I think all system programmers should be required by law to re-write it annually to remind themselves of the simplicity and elegance of the original UNIX API. If you want to delve into this code in more detail, look at the man pages for **fork()**, **exec()** and **dup2()**.

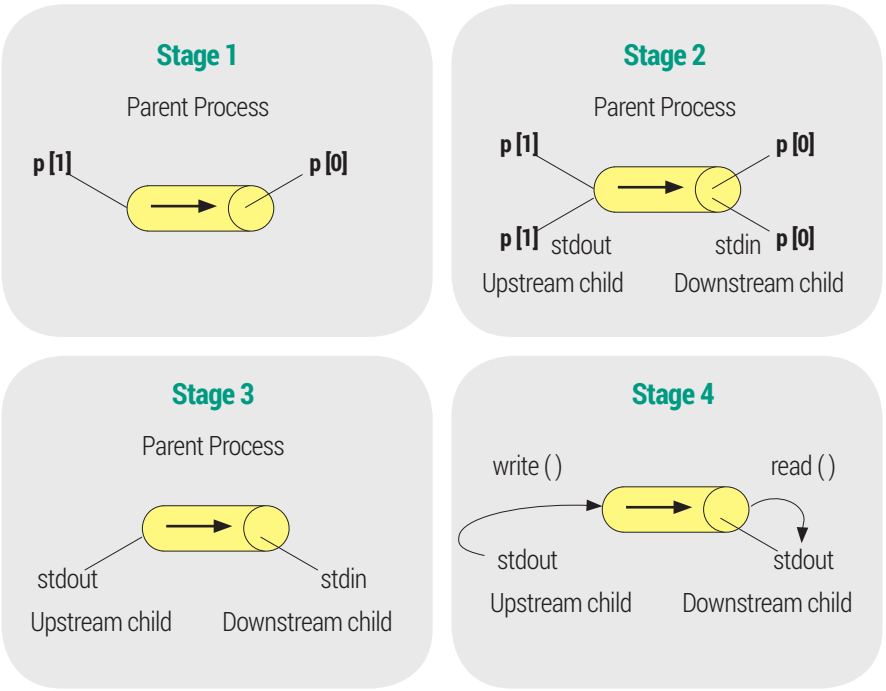
When created on the command line, pipelines can only be connected in a linear sequence. There is no way, for example, to feed the output of two programs into the input streams of a comparison program, or to run a program that generates two parallel output streams and sends each down a different pipe. These are really limitations of the command line syntax; it is perfectly possible to do both these things with pipes if you're willing to create and manipulate them in code.

Pipes and loops

You can, of course, create and use pipes within shell scripts. After all, a shell script is basically just a bunch of shell commands put into a file. What is less obvious is that you can pipe into and out of loops within the script. Here's an example of piping out of a loop:

```
1. #!/bin/bash
2. # Piping the output of a loop
3.
4. i=0
5. while (( $i < 10 ))
6. do
7.   echo $RANDOM
8.   (( i++ ))
```

Creating a pipe



A parent process creates a pipe and hands the file descriptors down to its children. Descriptors on the "unused" ends of the pipe are closed.