by Peter Collinson, **Hillside Systems**

BEK SHAKIROV

Redirection and Pipes

Most of the programs you use on any computer system operate by taking input from some source, performing some processing on that data, and then generating output. From the earliest computing days, designers realized that it was desirable to make the program operate independently from the actual peripheral that was to be used to deliver the input data or process the program's output. For example, the FORTRAN programming language, on which I cut my teeth, read input from a *channel* and wrote output to another channel. The program didn't care or know anything about the actual devices that were used to deliver input or process its output. The job of the operating system was to provide the actual connection between the I/O instructions from the program and the particular devices that the program was to use.

We give this notion a grand title: *device independence at runtime*, but I suspect the idea grew from pragmatic operating convenience rather than any specific theory of operating system design. In those days the system was run by operators. They wanted to take your card deck, place it into the nearest available card reader and divert its output to the printer that was free at that moment. The operators didn't want to mess with your card deck and wanted to make the decision about what peripheral device was to be used at the moment the job was entered into the machine.

However, the idea of device independence for programs made its way into many operating systems. When UNIX came along, each process was given three standard channels already established by the operating system when the process was started. Programmers were able to write programs that read from channel 0, and wrote their output to channel 1. Errors could be written onto channel 2, so that the user could choose to distinguish between the normal output from the program and any errors the program reported.

Channels on UNIX are "file descriptors." When a program opens a file on UNIX, it is handed a small positive integer used to refer to the file in any I/O requests. The kernel maps the file descriptor to a particular file, and maintains a positional pointer into the file. The pointer makes life simpler for the average process. When reading, the process doesn't have to remember where it was in the file, it is just given the next piece of data. When writing, the process can issue a series of write requests in the knowledge each write will append new data to the output file.

The kernel, then, establishes the first three file descriptors just before the process is run. We call these channels the *standard input* (channel 0), the *standard output* (channel 1) and *standard error* (channel 2). You will also come across them being called *stdin, stdout* and *stderr,* after the symbolic names used to refer to them in some programming libraries.

Where there are settings that change the way things work, UNIX has always tried to set up sensible defaults. If you are

running a program from the command line, then the three channels are set to point at your terminal. Typing

```
$ cat
```

will sit there waiting for you to type something, because in the absence of any arguments, the `cat` program will read from its standard input channel and write to its standard output channel. If you now type something like `hello world`, `cat` will write that to its standard output channel, which will print it on the terminal.

The `cat` program will continue copying data until it gets an end-of-file indication and we need some way of making the terminal tell the `cat` program that you've finished typing. The terminal interface has a special character (usually Control-D) which does this. Typing `^D` terminates your input to the `cat` program and it will exit, having completed its task. Your shell will regain control of the terminal. Incidentally, `^D` can often be used to log out from shells, because you have given an end of file indicator to the shell when it is trying to read a command and it will exit. Some shells inhibit this action, and request you to use the `logout` or `exit` command to leave the shell.

Redirection

The redirection operators work by changing the standard input and output channels, so when you type

```
$ cat < file
```

the shell opens the named file and establishes it as the standard input to the `cat` command. The operating system contains a system call that allows a process to re-assign file descriptors, so when the shell opens the file it may have file descriptor 5 but uses a system call to re-assign it to file descriptor 0. The command always reads from its standard input, which is now the file and writes to the terminal which remains as the standard output.

```
$ cat < file > opfile
```

will assign the standard input as before, and also alter the standard output channel. The shell will create a new file, `opfile`, and will set the standard output of the `cat` command to be the new file. Note that the file is created before the command is run so

```
$ cat < samefile > samefile
```

will result in a zero-length `samefile` being created, because the shell will delete all the information in `samefile`, and run the command which now has no data to copy. Don't do this.

Finally, we can append to a file by using

```
$ cat < file >> opfile
```

Dealing with Errors

The `cat` command is unlikely to generate any errors during its run, but how do we cope with a command that generates output and also generates errors? Well,

```
$ command > file
```

will print any errors on the terminal, which is often what we want. We do occasionally want to capture errors to a file and we need some way to redirect channel 2 output. In Bourne shell and its derivatives, like `ksh` or `bash`, we can put a channel number into the redirection syntax:

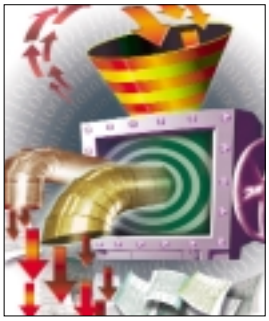
```
$ command > file 2> errorfile
```

In `csh` and its derivatives, you can only redirect both channels to one file by saying

```
% command >& logfile
```

The output from the command and any output on the error channel will be written to `logfile`. Actually, capturing both output streams to a single file is what we normally want and the Bourne shell syntax for achieving it can seem a little arcane.

```
$ command > logfile 2>&1
```



The operating system contains a system call that allows a process to re-assign file descriptors.

The `>&` is general-purpose syntax to manipulate file descriptors. In the example, we set channel 1 to be the `logfile`, and then say: make channel 2 point to the same destination as channel 1.

Pipes

Once we have created a file that contains the output of a command, we will undoubtedly want to process it. We may want to deal with the output in its entirety perhaps by looking at the result with an editor or a paginator like `more`. We may want to find out something about the output. For example,

```
$ ls > /tmp/lsct
$ wc -l < /tmp/lsct
```

will tell us how many files there are in the directory. The `-l` (*ell* not one) option to the `wc` command prints the number of lines that it finds and since each line of the file contains a single filename, counting the lines tells us how many files there are.

However, this is cumbersome. UNIX is a multiprocessing system, which means that we can run several commands in parallel. Things would become much simpler if we ran the `ls`

command at the same time as the `wc` command while making the standard output of the `ls` command feed directly into the standard input of the `wc` command. Pipes allow us to achieve this:

```
$ ls | wc -l
```

The vertical bar symbol is used to indicate that our command line consists of two commands, where the standard output of the first is to be connected to the standard input of the second. The commands now run in parallel with the `wc` command counting the data that's given to it by the `ls` command.

How Pipes Work

It's worth spending a little time looking at how this works. The pipe mechanism is provided by the operating system via a special system call that returns two file descriptors. The first of these ends up being the standard input channel of the `wc` command. After decoding its arguments, the `wc` command will read from its standard input channel, but since there is nothing there to read, the process will be put to sleep until some data appears.

The second of the file descriptors returned by the pipe system call ends up being the standard output of the `ls` command. The command will look at the current directory and merrily write its information to its standard output channel, which is now a pipe. The pipe mechanism will accept the data and store it ready for the `wc` program to read. Since data is now available in the pipe, the `wc` program will be woken up and will read the data until there is no more, then it will be put to sleep again waiting for more data.

We don't really want the two processes to operate in step, with the `ls` program writing data, then the `wc` program reading it, then the `ls` program writing some more data and so on. The standard *modus operandi* of `write` system call allows us to run both processes in parallel.

If you think about a process writing some data to a disk file, what normally happens is that the process says: "write this" and the data is moved into a kernel buffer before being written to the disk. Once the data has been moved into the kernel, it's safe for the process to be restarted. At some time later, the disk block will be written to the disk. When the process is restarted, it may generate some more data to be written, which may also be added to the kernel's buffer. Alternatively, the kernel may consider that it doesn't have space for the data and the process will not be run again until there is available buffer space to store the data. The main consequence of the buffering is that a process that writes data is not constrained to wait until that data is stored on the disk, so control can be returned to it and it can go off and do its other tasks. The process isn't working in lock step with the disk system.

The same mechanism is used for pipes. The pipe system is provided with a buffer, and, when a writing process sends data to the pipe, the process has control returned to it, so it can go off and do some other work before returning to add more data. If the buffer is full, the writing process is put to sleep until the reading process can empty the buffer sufficiently for the writing process to be able to add more data.

This buffering proved an immense boon for the `dump` program (known as `ufsdump` on Solaris). Its job is to scrabble about the disk dumping a file system and then writing that data to tape. In the original systems, you could watch the disk jump about, then the tape move, and then watch the disk jump about and so on. By making the top half of the program write to a pipe, and the bottom half read from the pipe and write to tape, suddenly the tape could be driven at full speed and it would operate asynchronously from the disk reading part of the program.

In summary then, the pipe consists of two file descriptors, connected by a buffer, which allows processes at each end of the pipe to run asynchronously. It's also worthwhile noting that the processes that use the pipe have to be related. The system call that creates the pipe has to be executed in one process and is then passed into child processes by the `fork` system call, which always preserves open file descriptors. There is then an elaborate dance to ensure that the appropriate file descriptor remains open in the correct process, so the process that's generating data will write to the output half of the pipe while the data sink process reads from the input half.

Filters

Once the notion of pipes was introduced to UNIX, many processes were changed to act as *filters*, taking data on their standard input, massaging it in some way, and writing an answer to their standard output. Many programs can be given a filename argument but default to reading from their standard input in the absence of a specific file in their argument list. This adaption allows them to be placed into a pipeline of commands, so

```
$ sort file
$ sort < file
$ cat file | sort
$ cat < file | sort
```

are all equivalent.

Some commands also allow a minus sign to be used to indicate that input is to be taken from their standard input. For example,

```
echo 'Line 2' | cat line1 - line3
```

will print

```
Line1 from file line1
Line2
Line3 from file line3
```

if the files `line1` and `line3` exist and each contain a single line of data as shown.

Pipes allow us to treat commands like tools, grouping them together to answer questions about our environment. I started

the last section with a pipeline that answers the question: how many files are there in the current directory? You can obviously use the `wc` command to count the output of any other command that generates data with one line for each item. For example, how many processes are now running on the system?

```
$ ps -ef | wc -l
```

and then remember to subtract one, because `ps` prints a header line.

We can start to do quite complicated things by massaging output from other standard commands. For example, let's look at finding the largest file in a directory. We can start with the `ls` command and note that it prints the size of the file when it's given the `-l` option. So what we'd like to do is sort the output of the `ls` command using the file size as the key. Now, we can undoubtedly do this with the `sort` command itself, but I personally find that the field specification mechanism for the `sort` command is quite hard to use. Also, and this is more important, it's difficult to see whether you have it right.

Ideally, I'd like to cut out the file size from the `ls` listing and just sort on that. We might also retain the filename. So my first attempt to see if I have the right data is

```
$ ls -l | awk '{print $5, $9}'
```

I am using `awk` to print columns five and nine of the output from the `ls` command. Some people would advocate using the `cut` command to pull out the information, but I personally prefer `awk` because by default it thinks columns are chunks of non-space characters separated by any number of spaces. So I can use my finger on the screen to count the columns starting from one, identifying the information I want to extract. The `cut` command forces me to count the character positions of the data I want to use. This is a real possibility, but a tedious one.

The command pipeline above prints a number, which is the size of the file, a space and then the filename. This format is useful, because `sort` understands it and will sort into numerical order on the initial value when given the `-n` switch. I can feed the sorted output into the `tail` command to generate the last line, giving me the size of the largest file in bytes and its name:

```
$ ls -l | awk '{print $5, $9}' |
      sort -n |
      tail -1
```

If you don't have the `tail` command then you can use `sed`:

```
....... | sed -n '$p'
```

The `-n` switch inhibits normal output from the `sed` command, and we just tell it to print the last line with `$p` which must be quoted to get the dollar character through the shell.

An alternative is to reverse the order of the sort and print the first line:

```
... | sort -nr |
      sed -n 1p
```

Of course, there are many ways to handle this problem. The main thing about the solution is you know it is correct, because each step involves a simple step whose action is verifiable. After a bit you become confident to just type the pipeline into the shell with no real testing or checking.

A common technique when writing pipelines is to add data to the line and remove it later when it has served its purpose. If I continue with my example, but want a full version of the `ls` command that sorts in size order, then I can start with

```
$ ls -l | awk '{print $5,$0}'
```

In `awk`, the `$0` variable contains all the current input line, so I am outputting the size of the file from column five followed by a space and the text of the original line. I can now sort on that initial numeric value, and having sorted I can remove the initial number with `sed`:

```
$ ls -l | awk '{print $5,$0}' |
       sort -n |
       sed -e 's/^[0-9][0-9]* //'
```

The `sed` command is given a regular expression that starts the match at the beginning of the line (the caret), followed by a character that must be from the range [0-9], followed by numeric value that's repeated zero or more times (shown by the star), followed by a space. If the match is found, it's deleted from the output line in the `s` command. Note that you cannot use just `[0-9]*` to match the number because the star operator matches *zero or more* repetitions of the previous character, so it will match nothing. To match a number that will always be there you must use `[0-9][0-9]*`.

If you try the command, you'll find that it will correctly show the "total" line that appears at the top of the `ls -l` listing. However, this is just luck. Changing the sort to `-nr` will cause the "total" line to appear at the bottom of the output.

I suspect that UNIX beginners are quite daunted by both `sed` and `awk`, and also by regular expressions. The book *sed & awk* by Dale Dougherty and Arnold Robbins (published by O'Reilly & Associates, ISBN 1-56592-225-5) is into its second edition and can help.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever ... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* `pc@cpg.com`.