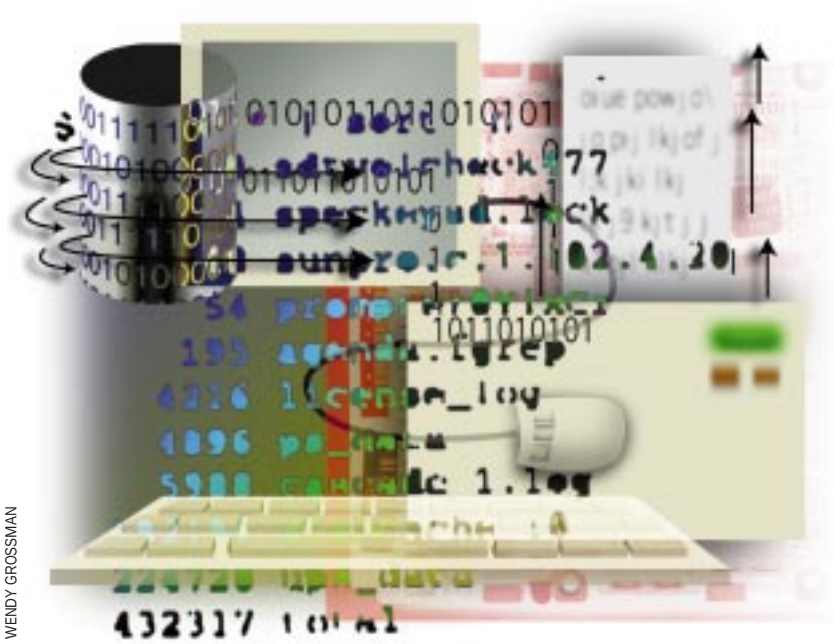
by Peter Collinson, **Hillside Systems**



WENDY GROSSMAN

Tool Talk

Last month, I talked about the notions of device independence and UNIX. We saw that most programs read and write streams of bytes. Programs are generally written so they read a stream from somewhere, or write a stream of bytes to somewhere, with no compiled-in knowledge of exactly where that "somewhere" is. The system provides mechanisms for setting up a source or a destination of the appropriate byte stream outside the running program. So a program can write to a disk file, a terminal screen or a printer where the destination is chosen by the user at run time.

Strictly, a byte stream is a sequence of characters processed by program. However, on UNIX, many programs impose structure onto the stream. These programs deal in text, where the byte stream is split into lines, several groups of characters each separated by a new line. Of course, this is convenient and maps directly onto human activity. Humans generate lines of text from their keyboard and expect to see lines of text scrolling up their screen.

Many utility programs on UNIX process text in some way or other and expect to receive or transmit new line-separated, variable length chunks of information. Actually, many programs that read data also have a secret limit to the length of the text line that they are able to process. The limit was set by a programmer back in the mists of time. When the program was written, the programmer wanted to store a complete line of text for processing by the program, and usually chose an internal buffer size that seemed infinite at the time. On 80-column terminals, no one would create lines longer than 256, 512, or 1,024 bytes, would they?

These hidden limits have proved a fruitful source of ways to hack into UNIX. All the hacker needs to do is provide a long text line, causing the program to overflow that previously infinite buffer. Then, perhaps, the program can be made to do unexpected things. It's true to say that most of these bugs have been fixed in recent system releases. The fix that was applied, generally, did not make the buffer infinite but ensured there is sufficient range checking so that the buffer overflow cannot happen.

To my mind, this is fixing the wrong bug. I want my UNIX tools to deal with lines of any length. There's a good reason for this. I'm beginning to find that some of these limits are more than inconvenient when dealing with HTML files created by some of the WYSIWYG HTML editors. These programs usually come from the heritage of word processing, where lines are very long because the new line character only appears at the end of a paragraph. When such files are transplanted onto a UNIX system, they appear to contain text but the lines are too long to be handled successfully by many of the tools, including many UNIX text editors.

If we ignore these infrequent line length problems, it's true to say that UNIX provides many tools that are designed to handle text. The tools fit casually together to process text byte streams and, with the shell, create an

extensible language to make new tools. To illustrate the language and its use, let's start with the wc program. Unlike some UNIX commands, wc is actually mnemonic and stands for "word count." It prints the number of lines, words and characters in a file. So, we'll get something like this:

```
$ wc file
     204     482     4674     file
```

This file has 204 lines, 482 words and 4,674 bytes. The command is intrinsically useful by itself, we can use it to find out information about the file. We can give the command several files. When we do this, we are also told the total number of lines, words and characters for all files:

```
$ wc file.*
     204     482     4674     file.1
     123    1678     7892     file.2
     327    2160    12567     total
```

The wc command will also read from its standard input and give us a similar result to the first example:

```
$ wc < file
     204     482     4674
```

It doesn't know the file name, so cannot print it. Any command that can read data from its standard input can also be placed into a pipeline. Here, two commands run in parallel, with the output of one being glued to the input of another:

```
$ cat file | wc
     204     482     4674
```

Sometimes, we'd like to control the output from the wc program, perhaps just seeing the number of lines in the file, or the number of words or the number of characters. In common with many UNIX programs, the wc program has options that modify its output, so to only print the number of lines, we'll add -l:

```
$ cat file | wc -l
     204
```

The next step is to realize that many UNIX system programs generate lines of text as a byte streams. We mostly display those lines on the screen, but we can pipe them into other programs, trivially answering new questions. For example, we might ask: How many people are logged in at present?

```
$ who | wc -l
      3
```

How many processes are running on this system? (Use ps ax on SunOS.)

```
$ ps -ef | wc -l
      92
```

Now, the astute reader will have noticed that perhaps I'm not telling the whole truth about this example. If you look at the output from the ps command, you'll see that it prints a title line, so the total number of lines outputted by the command is the number of processes running on the machine plus one. Subtracting one isn't that difficult, but if you're using the sequence in a script, you might want to remove the first line automatically. For example, by inserting a call to the sed command to delete the first line of the output from ps:

```
$ ps -ef | sed 1d | wc -l
```

The ps command has always printed a title line, showing that the ideas of tools and making commands fit together postdate the early development of UNIX.

Let's have another question: How many files are there in the current directory?

```
$ ls | wc -l
      16
```

There's something odd going on here, too. When you type ls, then you will see its output in columns:

```
$ cd /bin
$ ls
acctcom    filesync   mc68040    setuname
adb        find       mconnect   sh
addbib     finger     mesg       showrev
admintool  fmli       mkdir      sleep
alias      fmt        mkfifo     soelim
and so on
```

Output from ls wasn't always presented like this. The default output from the original command, as supplied in UNIX V6 and V7, only printed the names of the files, one per line. Of course, when the directory contains loads of files, then the file that you're trying to find will inevitably scroll off the screen. You would probably think this is bad. Well, in some ways, single-column output may be a good thing. Eons ago, whenever I saw names scrolling away in someone's home directory, I would use it as an excuse to tell them about how wonderful directories were, and how they could use the hierarchical file structure to organize their files. Single-column output provided them with a focus to put some order into their file collections. Rather than placing all their files in one directory, they were encouraged to collect them together in meaningful groups.

Anyway, at an early point in the development of the Berkeley System Distribution (BSD), the ls command was given columnar output by default. However, the ls command was already embedded in loads of shell scripts that could not be easily altered. The command was made backwards compatible by inserting a test to make it check where its output is going and alter its behavior depending on the result of that test. If its output device is a terminal, then it displays multicolumn output; if not, it prints a single column.

The folks at Bell Labs, the home of UNIX, didn't like this change to the `ls` command much, they regarded it as typical Berkeley hackery. They considered it unethical for programs that are intended to be general-purpose tools to ask specific questions about the devices with which they are communicating. A second, and perhaps more important, reason was they felt that programs should not change their behavior unless told to do so by the user.

A new version of the `ls` command (called `lc`) was created that always printed multicolumn output. The `lc` command found its way onto many versions of the "official" AT&T UNIX releases, but didn't really catch on. Berkeley `ls` prevailed. However, the result is a still an idiosyncratic hack. It can be confusing that:

```
$ ls
```

and

```
$ ls | cat
```

will give different results.

Filter Programs

Once we have the idea of passing the output from one program into the input of another, then it's a short step to begin to think about filters. These are programs designed to suck in data, usually one line at a time, do some processing and write an answer. We've had one of these already. The `sed` stream editor is a general-purpose tool often used to clean up data passing down a pipeline so that the sequence of commands you're using will give a correct answer. Of course, there is often no clear distinction between a stand-alone program and a filter. You can often employ a stand-alone program as a filter, and vice versa.

It's very common to want to look for some text in a file, a collection of files or a byte stream. The main filter program that is employed for searching is the `grep` command. By default, the command scans its input sources for the string that you give it and prints the lines it finds. For example, I might wish to find all the occurrences of `grep` in the articles I have written:

```
$ grep grep */*.ms
lines of output
```

When I pipe the output from this command into `wc`, I see that I have 135 lines containing `grep` in the text files that contain these articles. The format of the `grep` command is simple, the command name is followed by a single string (which is actually a regular expression allowing variable character matches), followed by a list of files to be searched. If there are no files, then the command assumes it should read its standard input.

For example, is my friend `aws` logged on?

```
$ who | grep aws
```

The command prints nothing, so `aws` is not logged on. We can use `grep` to find out what's happening on the system. For example, is `sendmail` running? (Use `ps -ef` on Solaris.)

```
$ ps ax | grep sendmail
 2369 ?     S 0:07 /usr/lib/sendmail -bd –q1h
19191 pts/4 S 0:00 grep sendmail
```

This shows that `sendmail` is running and also `grep`.

The `grep` command has three switches that are useful. First, the `-i` option performs a case-independent search for the string to be matched. This is useful when looking through general text and you have no clear idea about the case of the target. Second, the `-v` option prints the lines that don't contain the string to be matched. You can often use this to reduce output from a search to manageable proportions. If we wanted to list all the files in a directory that start with `a`, but don't contain `.doc`, we might say

```
$ ls a* | grep -v .doc
```

This pipeline results in a list of file names. We can use this list to move the selected files somewhere else:

```
$ mv `ls a*|grep -v .doc` fred
```

Here, we use the backquote operator to take the output from a command or command pipeline and insert it into the command line. On recent shells, like `ksh` or `bash`, you can use the POSIX syntax for this operation:

```
$ mv $(ls a*|grep -v .doc) fred
```

If the result of the `ls...grep` sequence is

```
a67.ms
auto.txt
```

then the mv command becomes

```
$ mv a67.ms auto.txt fred
```

moving the named files into the fred directory.

Third, you can give grep a switch making it print a list of file names that contain the lookup string rather than the lines in the files it matches. At least once a week, I'll say something like

```
$ vi $(grep -l somestring */*.ms)
```

starting the vi editor on a list of files that contain the word somestring. The ability to accurately pick up a list of files and edit them saves so much time.

Incidentally, there are traditionally three versions of grep. The standard, general-purpose grep takes regular expressions and searches for matching lines. The simpler fgrep is given strings with no regular expression metacharacters and uses a different match algorithm because it's looking for a complete match against a fixed string. The command can be given a list of search strings. The more complex egrep takes *extended regular expressions* and permits more complex matching, including alternates (look for *fred* or *jim*).

Personally, I use grep unless I need one of the features that is supported by one of the other commands. Actually, some of the publicly available versions of grep map the three variants to a single binary that takes different actions depending on how it is called.

Sorting Data

Sorting data into order is another way of seeing what is interesting without being force-fed screenfuls of meaningless information that you are required to scan. By default, the sort command sorts its input by lines into ASCII order, or actually these days, the order that is defined by the locale on your machine. I'll guess this means ASCII for most readers.

The sort command has a very useful option (-n) that makes it look for a numeric value at the start of each input line and then sorts the lines into numeric order depending on the value. We can use this to list files with wc and sort them into size order:

```
$ wc -c * | sort -n
       0 sdtvolcheck477
       4 speckeysd.lock
      48 sunpro.c.1.102.4.20
      54 prompter0YixCr
     195 agenda.fgrep
    4216 license_log
    4896 ps_data
    5988 cascade_1.log
  192196 dtdbcache_:0
  224720 ups_data
  432317 total
```

(This is my current /tmp directory.) The -c option to the wc command tells it to print the number of characters in a file. Here, the wc command is generating a number and a name on each line, and there are plenty of programs in UNIX that print their output like this.

One way of compressing lots of data into more meaningful output is to examine the frequency at which a text string appears in the data. Let's look at something concrete to illustrate this. The last command chugs through the login history on the machine (stored in /usr/adm/wtmp) telling you when someone last logged in and how long they stayed logged in. On a busy machine, the information can be voluminous, but let's say we were trying to find the actual user population on the machine, who has logged in and how often.

Well, the first column of the output from the last command is the login name of the user, so if we can pick that up and eliminate all the other information we don't need, something like

```
$ last | awk '{print $1}'
```

will give us the necessary data. The awk command is another general tool, we are telling it to print column 1 of every line that it receives. By default, columns for the awk command are defined as white space-separated chunks of text on a line. This pipeline will print a line for each user that has logged into the machine.

We can impose some order by sorting the data. We can then pass the sorted data into the uniq program. This program examines its input and collapses multiple copies of the same line into just one copy. Again, the uniq program is intrinsically useful. I've often used it to reduce sets of repeated blank lines in a file to just one blank line per source set. In our current context, we can use it to reduce output to only show those users who have logged in:

```
$ last | awk '{print $1}' | sort | uniq
```

The output is a sorted list of people who have logged into the machine. Beware that you'll get an apparently extraneous wtmp login, because the last command finishes its output by saying:

```
wtmp begins Sat May  1 21:28
```

wtmp is the file that is being examined by the last command. On a busy machine, this list may be useful, but it may be more helpful to know how many times a particular user has logged in. If we use the -c (count) option to the uniq command, it will print the line preceded by the number of repeats that it has seen of that line. Notice that this format is eminently suitable for passing into sort -n. So my full pipeline that tells me who has logged in and how frequently is

```
$ last | awk '{print $1}' |
         sort | uniq -c | sort -nr
```

which will give me output like the following:

```
 146 mmi1
 103 cj9
  99 jmb
  93 dpm
  88 cghd1
  73 eeb
  72 mcp
  69 ldc
```

and so on...

The `sort|uniq -c|sort -n` sequence is a common way of processing statistical data. The key idea is to create a format where the data that interests us is shown on separate text lines. Once we've done that, we can sort it, mostly to ensure that identical lines follow each other. We pass that data into `uniq` to count the repeats and then pass the data to `sort -n` to place the results in order. I use this technique for processing many log files where the most and least frequent events are often the most interesting.

Further Reading

There's lots of related material in *UNIX Power Tools* by Jerry Peek, Tim O'Reilly, Mike Loukides and others (now in its second edition, published by O'Reilly & Associates Inc., ISBN 1-56592-260-3). Also, get hold of *The UNIX Programming Environment* by Brian Kernighan and Rob Pike (published by Prentice-Hall Inc., ISBN 0-13-937681-X). This book is perhaps a little old now, it dates to 1984, but is one of those seminal UNIX books that all UNIX users should read.

***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 a SPARCstation 2. Email:* `pc@cpg.com`.