Computing with Pipes (Just Pipe It!)

TCS Developer's SIG
March 7, 2006

William Mitchell (whm)
Mitchell Software Engineering (.com)

Overview

Basics of Piping

Some Handy Tools

Ripped from the Headlines

I/O Redirection

Command Substitution

Piping and Editors

Things to Remember

The basics of piping

One way to solve a problem is to write a program but some problems can be solved by simply connecting programs.

Pipes let us connect programs.

An essential notion underlying pipes is that of standard input and standard output. A pipe connects the standard output of one program to the standard input of another.

Here is a *pipeline* that displays users in order by login name:

```
$ who | sort
dmr      pts/73   Feb 15 15:49
drh      pts/50   Feb 13 08:18
drh      pts/73   Feb 14 11:01
ken      pts/50   Feb 13 08:18
ralph    pts/47   Feb 12 12:47
rob      pts/47   Feb 16 21:47
wnj      pts/44   Feb 16 16:42
```

The standard output of who is piped into the standard input of sort. In turn the standard output of sort is displayed on the console.

By default, standard input is the keyboard and standard output is the console.

Note: The UNIX-derived tools used in this presentation can be obtained at www.cygwin.com.

Basics, continued

At hand:

```
$ who | sort
dmr      pts/73   Feb 15 15:49
drh      pts/50   Feb 13 08:18
ralph    pts/47   Feb 12 12:47
...
```

The output of sort can be piped into another program.

Here are some programs that read standard input and write to standard output: (Sometimes called “filters”.)

cut — Extract vertical strips from standard input, either by delimited fields or columns. (Examples: cut -f1 -d" ", cut -c10-20)

uniq — Outputs unique lines of (sorted) standard input. With -c, outputs a count of each unique line.

head — Outputs the first N lines of standard input. (Example: head -20)

wc — Outputs lines, "words", characters on standard input.

The customer wants to know:

How many people are logged in?

Who has the most logins active?

Who has been logged in the longest?

The grep command

The grep command outputs lines that match a regular expression. A simple example:

```
$ grep print *.java
Hello.java:        System.out.println("Hello, world!");
args.java:           System.out.println("|" + args[i] + "|");
dir.java:    void print()
dir.java:           System.out.println(entries[i].inode_number + ": " +
dir.java:       d.print();
dir.java:       d2.print();
lc.java:       System.out.println(count);
```

grep can read standard input:

```
$ cat *.java | grep print
        System.out.println("Hello, world!");
           System.out.println("|" + args[i] + "|");
    void print()
           System.out.println(entries[i].inode_number + ": " +
        d.print();
        d2.print();
        System.out.println(count);
```

(Note that cat *.java outputs the contents of each file in turn.)

How do the two outputs above differ? Why?

How could we output only the names of the files that contain "print"?

grep, continued

The -l (L) flag causes grep to simply output the names of files that have an occurrence of the pattern (a regular expression) specified on the command line:

```
$ grep -l print *.java
args.java
lc.java
```

The -v causes inversion—non matching lines are output:

```
$ grep -v print args.java
public class args {
    public static void main(String args[]) {
        for (int i = 0; i < args.length; i++)
        }
    }
```

Other handy options (among many):

-w	searches for whole "words".
-c	outputs a count of matching lines.
-n	outputs line numbers, too.
-C	outputs surrounding lines (five-line "window" by default).
-e	is used like this: 'grep -e -x ...', to search for -x.
-f *file*	reads patterns from a file.

grep, continued

The file words contains a list of words, one per line:

```
$ head words
aardvark
aaron
aback
abacus
abaft
abalone
...
```

Problems:

How many words contain every vowel, not counting words that contain a doubled vowel, like “food”?

Produce a sampling of the file by printing every 100th word (or so).

A story ripped from the headlines

The file newcust.out contains debugging output from stress tests of an ERP system interface being used to create customers. Here are a few lines:

```
=== clip 1 ===
/v/ss 530 % sg repeat_s.newcust
DON'T FORGET TO SET inside_firewall and/or run ssh vnet (sleep
15!) !!!!
Replaced underscores, result: 'repeat s.newcust'
Running 's.newcust'
Created customer 261427
Elapsed time 6123ms for 's.newcust'
Warning: server pool exhausted
DON'T FORGET TO SET inside_firewall and/or run ssh vnet (sleep
15!) !!!!
Created customer 261430
Elapsed time 5150ms for 's.newcust'
Warning: server pool exhausted
```

Problems:

How many customers were made?

Were any duplicate customer IDs created?

Were any sequence numbers skipped?

Was there anything unexpected in the output?

What was the largest/smallest elapsed time?

Some curious behavior

The ls (LS) command is like dir—it displays information about files.

What's wrong with this picture?

```
$ ls
backup             pipes.notes              who.1
badsort            pipes.nts.last           who.bak.icn
lines              pipes.nts.wpd            who.exe
mostlogins         pipes.sli.last           who.icn
notes.notes        pipes.sli.pdf            who.sh
ooee               pipes.sli.wpd            words
oowords            recover1.pdf             x
pipes.bug1.wpd     s.newcust.022806.1509
pipes.crash2.wpd   slides.notes

$ ls | wc -l
25
```

Problem: Print the name of the most recently modified file in the current directory.

Questions to ponder...

What program characteristics do (or don't) make it easy to use a program in a pipeline?

Out of the box, which Windows XP programs can be used in a pipeline?

Which is better: `dir/p` or `dir | more`?

Is piping incompatible with GUIs?

Does an operating system need to support multitasking in order for a shell to provide piping?

What other sorts of computation does piping remind you of?

Redirection operators

Shells commonly support **<** and **>** as *redirection operators*. They allow standard input and output to be redirected from/to files.

Examples:

```
$ wc < words
 47958  47958 494442

$ grep oo < words > oowords

$ wc oowords
 859  859 8453 oowords

$ grep oo < words | grep ee > ooee
```

Note that most, but not all, file processing utilities read standard input if no file arguments are specified on the command line.

Questions:

Are **<** and **>** really needed or are they just syntactic sugar?

Speculate about the result of this command: wc < words words

For simple programs, what is a great benefit of redirection being provided by a shell?

Truth is stranger than fiction

Once upon a time, users of DEC's VMS operating system did output redirection like this,

```
$ assign/user sys$output out
$ run program
```

Contrast with UNIX:

```
$ program > out
```

Which do you think came first, VMS or UNIX?

Command Substitution

Note: All examples shown previously work on the XP command line. The following slides explore a facility found only(?) in POSIX shells, such as bash. *(But those shells are available on Windows via Cygwin.)*

The *command substitution* facility provides a way to turn the output of a command into command-line arguments. Example:

```
$ cat srcfiles
lc.java
mkall.icn
getpid.c
$ echo $(cat srcfiles)
lc.java mkall.icn getpid.c
```

(Note: The echo command simply outputs its arguments, all on one line.)

On a command line, the form $(*command-line*) indicates to run the enclosed *command-line* and substitute the whitespace-separated words it produces for the $(...) construct. The resulting command line is then executed.

Any number of command substitutions may appear on a command line, the enclosed commands may be arbitrarily complex, and substitutions may be nested.

Command substitution, continued

Three more examples:

```
$ ls -l $(cat srcfiles)
-rw-r--r--   1 whm      74 Sep  1 14:23 getpid.c
-rw-r--r--   1 whm     360 Aug 14 18:54 lc.java
-rw-r--r--   1 whm     115 Aug 17 00:57 mkall.icn

$ wc $(cat srcfiles datafiles)
      15      36     360 lc.java
       6      16     115 mkall.icn
       6      13      74 getpid.c
       7      24     452 lc.1
      14      36     259 lc.2
      48     125    1260 total

$ wc $(cat srcfiles datafiles | sort -k 2 -t.)
       7      24     452 lc.1
      14      36     259 lc.2
       6      13      74 getpid.c
       6      16     115 mkall.icn
      15      36     360 lc.java
      48     125    1260 total
```

Problem: Use more to look through the files in the current directory with the suffix .icn and that contain the word “reverse”.

Command substitution, continued

Note that the echo command and command substitution are inverses:

> *echo turns arguments into output; command substitution turns output into arguments.*

Consider this:

```
$ echo a b c
a b c
$ echo $(echo a b c)
a b c
```

An older, but very commonly used form of command substitution is `...` (back-quotes):

```
finger `whoami`
wc `cat srcfiles datafiles | sort +1 -t.`
```

The older form is a little easier to type, but doesn't nest:

```
$ echo $(echo $(echo x))
x
$ echo `echo `echo x``
echo x
```

A sip from a firehose

Here's a UNIX shell script:

```
$ cat script1
for i in $*
do
    mv -i $i $(echo $i | tr A-Z a-z)
done
```

Usage:

```
script1 *.dat
```

Speculate: What does the script do?

Problem: Write a one-pipline script named mostlogins that displays the name of the user with the most active login sessions, and the number of sessions. Example:

```
$ mostlogins
gifford is logged in 13 times
```

Hint:

```
$ printf "x=%d y=%s\n" 5 apples
x=5 y=apples
```

Pipes and editors

Many UNIX-grown editors like Emacs and vi provide facilities to filter buffer contents through a pipe.

In Emacs, M-| (shell-command-on-region) prompts for a command line and runs it, supplying the contents of the selected region as standard input. If an argument is specified for shell-command-on-region, the output of the command line replaces the region.

Problem:

> The file numbers contains the integers from 1 to 1000 in a random order. Pick a 50-number sequence somewhere in the middle and see what its sum is.

Practical application:

> Imagine a filter named genfmt that reads expressions, one per line, and generates a Java System.out.format statement that produces labeled output for the expressions. That can be used to generate code for debugging. (If only there were a preprocessor in Java...)

Things to Remember

This talk introduced a handful of programs that work well in pipelines. There are many more. Two more that are especially handy are find, for finding files with various attributes, and sed, a stream editor.

Remember that the man command can be used to display documentation on program options. On properly configured systems, man -k *word* looks for commands whose descriptions contain the specified word.

If you're on a UNIX/Linux/POSIX system you've already got the tools used in the presentation.

If you're on Windows, the Cygwin tools work pretty well. Get them at cygwin.com but before you go after them, be sure to read Section 2, *Setting Up Cygwin*, of the Cygwin User's Guide:

http://www.cygwin.com/cygwin-ug-net/cygwin-ug-net.html

The presenter used to use The MKS Toolkit, another port of UNIX tools for Windows, but is out of touch with how the current MKS product compares to Cygwin.

You don't need to get any tools at all to make use of the notion of piping:

Simple programs that read from standard input and write to standard output can be combined to perform significant computations.

Write those programs and Just Pipe it!