Week02 - Pipes, I/O Redirection, and a Few More Tools

January 25, 2021

This week we'll look a couple of quintessential parts of Unix scripting, which are pipes and I/O (input/output) redirection. Then we'll spend a few minutes looking a few command line programs we were'nt able to cover last week.

1 Setup

Turn on a Debian 10 server and connect with ssh.

After the class you can try to go through these notes with your Macbook, with the gitbash terminal you installed on your PC, or with a different Linux/BSD distribution. You'll find most (if not all) of the things we do will work just as well on those platforms.

2 Learn as we go

I said I'd teach you a few new commands tonight. We'll just learn them as we go. You'll learn to use:

- history
- grep
- head
- tail
- cut
- sort
- uniq

3 Pipes

Pipes are one of the famous/important/fantastic command line tools you use on Linux. They allow you to create beautiful little programs by stringing together the basic programs you know like "cat", "echo", "cut", etc..

The pipe symbol is “| ".

You put this between the different commands you want to run to cause the data to flow through them like water through a pipe.

https://www.youtube.com/watch?v=bKzonnwoR2ILater you could watch this video of Brian Kernighan talking about pipes. He's one of the creators of many of the things you are learning this semester.

Here are a few examples in no particular order.

3.1 Searching your history.

history - this commands tells you all the things you've typed into your terminal. Go ahead and try it! Type history and you'll see all the things you previously typed.

grep - this cool program is for searching.

You might rename a PNG image to a new name. Now you can't remember what you renamed it to. So use a pipe!

```
root@machine$ history
# all of the things you typed
root@machine$ history | grep "mv"
# all of the mv commands you typed
root@machine$ history | grep "mv" | grep "png"
# all of the mv commands you typed that also contain the
    letters "png"
```

I often forget the ip address of the last machine I ssh-ed into. Use the same trick with history, grep and a pipe!

```
root@machine$ history | grep ssh
# all of the old ssh connections I made
```

3.2 Count the letters in the second word

Maybe you want to know how many letters are in the second word of a sentence.

```
root@machine$ sentence="the quick brown fox jumps over the
    lazy dog"
root@machine$ echo $sentence
the quick brown fox jumps over the lazy dog
root@machine$ echo $sentence | cut -d" " -f2 | wc -c
6
```

Actually we know the word "quick" has 5 letters - don't forget that cut appends a "
n" character.

In the above example we also learned a little bit more about "cut". "cut" takes a '-d' flag, which means 'delimiter'. In this case our delimiter was " ". It also takes an '-f' flag, which means 'field'. We cut the sentence on spaces, and grabbed the second one.

3.3 How many words on the third line?

Assume you want to know how many words are in the third line of a text file.

```
root@machine$ cat sentences.txt
whose woods these are I think I know
his house is in the valley though
he will not see me stopping here
to watch his woods fill up with snow
root@machine$ head -n3 | tail -n1 | wc -w
7
```

The head command gives us the first three lines. From there we use the tail command to trim off just the last line. Then we use wordcount to count the words.

3.4 What is the largest number in Column 2?

Assume we have a CSV file (Comma Separated Values) and for some reason we want to know the largest value in column 2.

```
root@machine$ cat data.csv
1,2,3,4
4,5,6,7
7,8,9,0
4,4,4,4
root@machine$ cut -d "," -f2 data.csv | sort -gr | head -n1
8
```

In this example we use cut to extract the second column. Then we pass the second column on to the sort program and we sort it in general numeric order (g) and descending (r), and then we pass that sorted data on to 'head' and take the first line. This gives us the largest number.

3.5 How many unique numbers in numbers.txt?

The uniq command takes a bunch of lines of sorted data and outputs just the unique elements. The input must must must be sorted

```
root@machine$ cat numbers.txt
1
2
3
2
1
root@machine$ uniq numbers.txt
1
2
3
2
1
```

See? It doesn't work. Make sure to sort first.

```
root@machine$ cat numbers.txt
1
2
```

```
3
2
1
root@machine$ sort -g numbers.txt | uniq
1
2
3
```

3.6 The Message

All of the above examples show that we can quickly write useful little programs on the command line by just piping data through the basic programs we already know.

This is a major part of Unix philosophy.

We have a bunch of simple, seemingly boring tools - but then we glue them together into a more meaningful program. And it is easy!

How long would it take you to write, compile and run a Java program to read a .csv file, find the largest value in a column, and then print the output? And how much typing would you have to do?

IO Redirection

In this section we learn about **stdin**, **stdout** and **stderr**

Intro

What a great time to be in a class. In the next few minutes you're going to hear about what all the Linux people know.

So far I've shown you stdout, but I never told you what it was.

```
melvyn@laptop$ # the > is for stdout
melvyn@laptop$ echo "hello" > hello.txt
melvyn@laptop$ cat hello.txt
hello
```

The greater than sign indicated stdout (standard out). It takes the output of a command and routes it somewhere. In this case, the standard out of 'echo' was routed to a file called 'hello.txt'.

Another interesting thing to note is that if you repeatedly use this operator, it does not append to the output file, it overwrites it.

```
melvyn@laptop$ # the > is for stdout
melvyn@laptop$ echo "hello" > hello.txt
melvyn@laptop$ echo "hello" > hello.txt
melvyn@laptop$ echo "hello" > hello.txt
melvyn@laptop$ echo "hello" > hello.txt
melvyn@laptop$ wc -l hello.txt
```

What do you expect? 4 lines or 1 line? Try it for yourself! You will find that there is only one line. If you want to append output and not overwrite, you have to use this operator:

```
melvyn@laptop$ # the > is for stdout
melvyn@laptop$ echo "hello" >> output.txt
melvyn@laptop$ echo "hello" >> output.txt
melvyn@laptop$ echo "hello" >> output.txt
melvyn@laptop$ echo "hello" >> output.txt
melvyn@laptop$ wc -l hello.txt
```

Now you will see there are four lines!

Details

All linux processes have at least 3 "file descriptors" or communication channels with the outside world. Does anyone not know what they are? Then Ill tell you. You have standard in, standard out, and standard error. We've already begun using them in this class briefly with the $>$ operator.

Open vim and create a file. Type a couple of words in it, we're going to use that file for the next exercise.

```
melvyn@laptop$ wc fileThatExists > log
```

You see nothing. Running cat log will show you the output of the wc tool

```
melvyn@laptop$ wc filethatdoesntexist > log
```

You see an error in the shell, cat log shows nothing. There was no std out of wc, but there was err output

```
melvyn@laptop$ wc filethatdoesntexist 1>log 2>errLog
```

Now there is no error on the screen, but cat errLog shows the error from wc.

```
melvyn@laptop$ wc filethatdoesntexist >log 2>errLog
```

You can put the 1 or not, it assumes the 1 is for stdout, 2 is for stderr.

A common "idiom" you'll see is

```
melvyn@laptop$ wc file 1>place 2>&1
```

that redirects both stdout and stderr to the same location.

You can also write

```
melvyn@laptop$ wc filethatdoesntexist &>place
```

to redirect both to 'place'.

As an aside look what vim does to the files. Not essential stuff, but this is just to deepen your knowledge of vim and how much attention to detail is required in programming. What if you are working on a project and there is a datafile that shouldn't have newlines. Then you peek in the file with vim and absent mindedly close it with :wq. You could cause an error that could take days to debug!!

```
wc errLog
0 blah blah
```

There are no newlines

Not open errLog with vim and quit immediately with :q

```
melvyn@laptop$ wc errLog
0 blah blah
```

Now open errLog with vim and close with wq. Then

```
melvyn@laptop$ wc errLog
1 blah blah
```

Now there is a newline! Writing with vim - vim always puts a newline at the end of the file.

Now lets move on to standard in. We've already seen alot of it. Pipes use standard in!! You can always pipe the stdout of one process to the stdin of another!

```
username@computer$ cat file | wc
```

The stdin of wc picks up the stdout of file! Processes can take inputs and give outputs.

There are various ways to feed a process's standardin. Some tools allow you to just give it as a positional argument, like wc.

```
student@computer$ wc file
```

'file' goes to the stdin of wc.

You can also be more explicit. We saw that stdout is "1" for some mysterious reason, and stderr is "2" for an equally mysterious reason. Well, not a mystery, this is just boring technical stuff. Some programmer way back when made the decision to associate these numbers with the indicated channels, and that's the way it is! Why do we type "cp" to copy a file??? Just another boring technical reason, that's how the computer program was written , right?? Well, stdin is 0. Also, output is associated with > and », input is associated with <

So we can also type 'wc < file' or 'wc 0< file'

I don't know if I've ever written this syntax, maybe once or twice. I'm an application and operating system developer mainly. It certainly isn't in my day to day work, but it's good to know in case you come across it in a script, an exam, or have to teach it!

Of course, you can mix these things

```
melvyn@laptop$ wc 0< file > log`
```

Also, don't forget about "append" vs write when it comes to the stderr/stdout stuff

```
melvyn@laptop$ wc filethatdoesntexist 2>> err
melvyn@laptop$ wc filethatdoesntexist 2>> err
melvyn@laptop$ wc filethatdoesntexist 2>> err
melvyn@laptop$ wc filethatdoesntexist 2>> err
melvyn@laptop$ cat err
```

You'll see alot of error messages logged in "err"

An aside, maybe you're interested? How to pipe standard error? `https://stackoverflow.com/questions/1507816/with-bash-how-can-i-pipe-standard-error-into-another-process`

A common thing you'll see and maybe want to do is make your processes shoosh! If a process is running and giving alot of output you don't want you can throw away the output channels.

```
melvyn@laptop$ wc f 2>/dev/null
```

A common place to send unwanted output is to /dev/null, it just writes your output to the ether and you don't see it and it isn't logged anywhere.

The above command should give an error, but you won't see it, its just gone. There are many times you'll want to do this. I won't dream up some big situation right now to illustrate this to you, just know you'll see this all over the place in bash scripts and there will come a time, if there hasn't already, where you'll just want to throw away either stdout or stderr and never hear about it.

Using Stdout, Stderr in Your Own Programs

Java

```java
public class StdoutStderrExample{
  public static void main(String[] args){
    System.out.println("I went to stdout!");
    System.err.println("I went to stderr!");
  }
}
```

You can use vim to write that program. Then you can compile it, run it, and use the io redirect operators to send the output to different files.

```
$ apt install default-jdk # this will install your java tools.
$ javac StdoutStderrExample.java
$ java StdoutStderrExample 1>java_out.log 2> java_err.log
$ cat java_out.log
I went to stdout
$ cat java_err.log
I went to stderr
```

C++

```cpp
#include <iostream>

int main(){
  std::cout << "I went to stdout!" << std::endl; //endl is
      endline, not end one.
  std::cerr << "I went to stderr!" << std::endl;
}
```

You can use vim to write that program. Then you can compile it, run it, and use the io redirect operators to send the output to different files.

```
$ apt install build-essential # this will install your c plus
    plus tools.
$ g++ StdoutStderrExample.cpp
$ ./a.out 1>cpp_out.log 2>cpp_err.log
$ cat cpp_out.log
I went to stdout
$ cat cpp_err.log
I went to stderr
```

Python3

```
import sys

sys.stdout.write("I went to stdout!\n")
sys.stderr.write("I went to stderr!\n")
```

You can use vim to write that program. Then you can run it and use the io redirect operators to send the output to different files.

```
$ apt install python3-dev # python3
$ python3 StdoutStderrExample.py 1>py_out.log 2>py_err.log
$ cat py_out.log
I went to stdout
$ cat py_err.log
I went to stderr
```