Lecture23
Shell Scripting II

Regular Expressions

We learnt about RegEx in Perl programming lectures. Regex is a pattern that can be described by a language. For example, we can find all directories within a given directory by typing

- ls –l | grep ^d

The output from ls –l is piped into another process where grep looks for lines that "starts" with d. If you would like to know all the non-directory entries, we can type:

- ls –l | grep ^[^d]

grep can take arguments that are regular expressions (regex) and match them to find patterns in files.

sed

sed, the *stream editor* is another tool. Its history is somewhat interesting. Back in "the day", unix's editor was a simple line-editor affectionately known as *ed*. It was designed for a teletype and, as a consequence, displayed only one line at a time. But, to make things easier, it could do fairly sophisticated edits on that line, including searches and replaces, &c.

Eventually, with the advent of CRT terminals, it was extended into the *vi*sual editor that we all know and love. *sed*, the *s*tream *ed*itor, is another one of the original *ed*'s offspring. When it comes right down to it, *sed* is a programmable filter. Text is piped into its stdin, gets filtered, and gets pumped back out via stdout in its mutated form. The programming is done via regular expressions. Since *sed* is a filter, its normal behavior is to output everything that it is given as input, making any changes that it needs to make along the way.
Perhaps the most common use of sed is to perform some type of search and replace.

Here's an example. It will change every instance of the |-pipe into a ,-comma.
This might be used, for example, to change the delimiter within some database "flat file".

cat somefile.txt | sed 's/|/,/g' > outfile

s is the substitute command. The form of s is **s/pattern/replacement/flag(s)**
The delimiter can be anything, in this case we have chosen that to be a /. Regardless, the character immediately after the s-command is the delimiter. We might change it, for example, if we want to use the delimiter as part of the pattern. For example, if you are looking for directory names you may want to do this. Perhaps we can use #

For example, the following is equivalent but uses a #-pound as the delimiter:

cat somefile.txt | sed 's#\|#,#g' > outfile

Delimiter can also be escaped with \ if you'd like.

The pattern can be a regular expression, and the replacement is usually the literal replacement. The flag is usually "l" for local or "g" for global. A local replacement replaces only the first match within each line. A global replacement replaces each and every match within all lines. A specific number can also be used to replace up to, and including, that many matches -- but not more. At this point, let's remember that sed is designed to work on text files. It has a routine understanding of a line.

Sed can also be directed to pay attention to only certain lines. These lines can be restricted by number, or to those that match a particular pattern. The following only affects lines 1-10. Notice the restriction before the substitute.

cat somefile.txt | sed '1,10 s/\/,/g' > outfile

The pattern below would affect lines 10 through the end:
cat somefile.txt | sed '1,$ s/\/,/g' > outfile

The example below will operate only on those lines beginning with a number. Notice the pattern is contained within //'s:

cat somefile.txt | sed '/^[0-9]+/ s/\/,/g' > outfile

Another use of sed is to generate a more powerful grep. Remember, most greps work only with truly regular expressions. And, remember that most greps can't use {ranges}. Consider the example below where sed is used as a more powerful grep. It prints any line that begins and ends with the same 1-3 digit number:

cat somefile.txt | sed -n '/^\([0-9]\{1,3\}\).*\1$/ p' > outfile

Okay. So, let's decode the example above. Recall that sed is normally a pass-through filter. Everything that comes in goes out, perhaps with some changes. The "-n" filter tells sed that it should not print. This, under normal circumstances, makes it quiet. But, in this case, we are using the "p" command to tell sed to print. So, now we see the whole magic. First, we tell sed to be quiet by default. Then, we tell sed to print the selected lines. We make the line selection as we did above, by specifying a pattern.

While we are talking about the power of regular expressions within sed, let me mention one of its features: the &-ampersand. When used on the right-hand side of a substitute, the &-ampersand represents whatever was actually matched. Consider the following example:

cat somefile.txt | sed 's/[0-9][0-9]*[+\-*\/][0-9][0-9]*/(&)/g' > outfile
sed has many more features that we're not discussing. "man sed" for more details.

cut

cut is a quick and dirty utility that comes in handy across all sorts of scripting. It selects one portion of a string. The portion can be determined by some range of bytes, some range of characters, or using some delimiter-field_list pair.
The example below prints the first three characters (-c) of each line within the file:
cat file | cut -c1-3
The next example uses a :-colon as a field delimiter and prints the third and fifth fields within each line. In this respect lines are treats as records:
cat file | cut -d: -f3,5
In general, the ranges can be expressed as a single number, a comma-separated list of numbers, or a range using a hyphen (-).

tr

The *tr* command *tr*anslates certain characters within a file into certain other characters. It actually works with bytes within a binary file as well as characters within a text file.
tr accepts as arguments two quoted strings of equal length. It translates characters within the first quoted string into the corresponding character within the next quoted string. The example below converts a few lowercase letters into uppercase:

cat file.txt | tr "abcd" "ABCD" > outfile.txt

tr can also accept ranges, as below:

cat file.txt | tr "a-z" "A-Z" > outfile.txt

Special characters can be represented by escaping their octal value. For example, '\r' can be represented as '\015' and '\n' as '\012'. "man ascii" if you'd like to see the character-tonumber mapping.

The "-d" option can be used to delete, outright, each and every instance of a particular character. The example below, for example, removes '\r' carriage-returns from a file:
cat file.txt | tr -d "\015" > outfile.txt

Pipes as an Inter-Process Communication (IPC) Primitive

We've seen the use of pipes lately. In a very real way, they are the glue that tie shell scripts together. But, interesting enough, pipes are tools for "Inter-Process Communication (IPC)". In other words, they connect two different processes together.

And, at the command line, this makes sense. Consider "ls | cat | more". If we launch this from the UNIX shell, we've got four processes in play: the shell, itself, ls, cat, and more. In order to launch each new command, the shell "forks" a new process and then "execs" the desired program within it. So, one pipe bridges "ls" and "cat" and s second pipe bridges "cat" and "more". And, based on our experience using the shell, this makes good sense.

But, how does this work within a shell script? Well, exactly the same way. The shell interpteter is running the shell script. But, each time the shell runs a command, it forks a new process and execs that command. So, each command is a different process and can be bridged using a pipe.

Pipes Can't Edit In Place

Often times we want to edit a file by applying filters using pipes. The following is an example -- illustrating a very common **problem**. It is intended to translate carriage-returns into line-feeds using *tr* and then wraps long lines using *fold*. And, depending on your use cases, it'll often seem to work -- but it is badly broken. Do you see the problem?

```
cat somefile.txt | tr -d "\015" "\012" | fold > somefile.txt
```

The input file at the beginning of the pipeline and the output file at the end are the same file. If this file is small, and is read in its entirety before output begins to emerge from the *fold* at the end, this example will work fine. But, life becomes interesting when the *fold* emits output before the input file is fully read by *cat*. When *fold* goes to write the file, it truncates it and overwrites it from the beginning. So, as *cat* proceeds to read the file, it will either find it empty and end early or end up swallowing newly written content rather than the original data.

To understand the nature of the problem, we need to understand how a pipe works. It is a finite buffer -- a typical size might be 8k. If the input file is less than 8k, it is likely to be quickly read in and completely placed within the buffer. In this case, it probably gets there before the last command in the pipe begins producing output, causing the input file to be truncated.

But, think about what happens if the input file is larger than the buffer. If *cat* starts feeding the pipe and gets ahead of the processing in the pipeline, the buffer can become full. When this happens, the *cat* can't write into the full buffer, so the operating system temporarily pauses it, until it can. This temporary pausing is called *blocking*. In the meantime, the output might start flowing, truncating the original input file, making it impossible for the *cat* to get the rest of it.

In order to solve this problem, we always send the output to a temporary file. Then, once done, we move the temporary file back over the original. Consider the example below:

```
cat ${1} | tr -d "\015" "\012" | fold > ${1}.tmp
mv ${1}.tmp ${1}
```

The example above is a step in the right direction, but it isn't quite as we'd like. If two user's run the same program at the same time, it can run into problems. Both instances of the script will try to make use of the same temporary file --even though they are likely to be at different points in the process and operating on different input files. The first one to start will own the file and get to make use of it -- the other one is likely to lose. It won't be able to write to the file. And might not be able to read it. So, at best it does nothing as it can neither read nor write the input file. At worst, it takes the output of the other instance as its result.

To solve this problem, we make use of the *$$* special variable. You'll recall that each instance of the shell script will have its own process id. As a result, the *$$* acts as a great unique identifier -- it makes sure that each instance operates upon its own temporary file.

```
cat ${1} | tr -d "\015" "\012" | fold > ${1}.tmp.$$
mv ${1}.tmp.$$ ${1}
```

As a final, minor revison, we'll put the tmp file into the "/usr/tmp" directory. The "/usr/tmp" and "/tmp" directories are writeable by anyone. And, as an added bonus, these directories are cleared upon reboot. This way, if we forget to clean up our temp files, there is some hope that they'll eventually get thrown away. Also, by writing into this directory, we make the purpose of these files clear -- they are temporary files. I've also enclosed the file names within ""-quotes -- it makes them more robust in the event of unescaped spaces.

```
cat "${1}" | tr -d "\015" "\012" | fold > "/usr/tmp/${1}.$$"
mv "/usr/tmp/${1}.$$" "${1}"
```

Pipes, Loops, and Subshells

Consider the example below. Notice the pipe between the cat and the while loop:

```
#!/bin/sh

FILE=${1}

cat ${FILE} |
while read value
do
  echo ${value}
done
```

This is a bit curious. A pipe can only connect two different processes. The while loop is interpreted by the shell. How does this work? Where is the second process?

The answer is that the shell creates a *subshell* for the while loop. A subshell is a separate shell spawned by the primary shell. The loop is run within the subshell rather than within the primary shell. So, the pipe connects the primary shell's "cat" with the subshell's "read". So now we have genuine IPC within a shell script.

Danger, Danger! Hidden Consequences

Check out the following example. Does it work as apparently intended? What do you think?

```
#!/bin/sh

FILE=${1}
max=0

cat ${FILE} |
while read value
do
  if [ ${value} -gt ${max} ]; then
    max=${value}
  fi
done

echo ${max}
```

What is the problem with the above script? The logic seems fine, but when we run it, it acts weird. At the end, "max" is still 0. But, if we inspect it, such as by an echo, within the loop, it is incrementing correctly.

The problem here has to do with subshells. We already know that a new subshell needs to be created for the while loop. So, what does that mean about the "max" variable? How does it get into the subshell? Well, the new subshell gets a clone of most of the old subshell's variables. So, we've actually got two "max" variables -- one in the original shell and one in the loop's subshell. The loop increment's its own copy of "max" -- leaving the "max" within the original shell.So instead of piping the results into the loop, we are going to capture them into a variable, and then iterate through the resulting list. Notice that, in the example below, there is no pipe and, as a result, no resulting subshell to cause problems. No variables are shadowed. This "capture and loop" idiom is very common in shell scripting.

```
#!/bin/sh

FILE=${1}
max=0
values=`cat ${FILE}`

for value in ${values}
do
  if [ ${value} -gt ${max} ]; then
    max=${value}
  fi
done

echo ${max}
```

Arrays in Bash

Although not often used, one dimensional arrays are available in bash. Array elements can be initialized as for example,

```
array[2]=23
array[3]=45
array[1]=4
```

To dereference an array variable, we can use, for example

```
echo  ${array[1]}
```

Array elements need not be consecutive and some members of the array can be left uninitialized. Here is an example of printing an array in bash. Note the C style loop. Also note the spaces between tokens.

```
for ((  i=1  ;  i<=3  ;  i++  ))
do
  echo ${array[$i]}
done
```

Exercises

[1] Write a shell script that takes a path of a directory and store all sub directories (recursively) in an array. Then print the array to show all the directories found.

[2] Write a shell script that takes a file where each line is given in the format
F08,guna,Gunawardena,Ananda,SCS,CSD,3,L,4,15123 ,G ,,
And creates a file that contains only the user ID's on each line.

[3] Write a shell script that takes a folder as a command line argument and set system:anyuser access permission to none for all folders and subfolders

[4] Write a shell script that takes a single file C program main.c and an expected output file output.txt and produce an error message if the program does not produce the expected output and congratulates if the program does produce the expected output