UNIX Lab 2 – File Permissions, Input/Output, Searching and Regular Expressions (and Shell Metacharacters)

Introduction to UNIX Access Permissions

These exercises will familiarize you with the basic UNIX commands for working with file access permissions. Please use the directories and files you created in the first lab, or you can create them again here for these exercises.

- Make sure that you are in your unixlab1 exercise directory, and then do a long listing of your files:

```
cd ~/SAN/unixlab1
ls -alg
```

- Notice the access permissions assigned to each file. Can you tell:

  - which files are directories?
  - if a file is executable?
  - who has write access to each file?
  - who has read access to each file?
  - who owns the file?
  - which group the owner belongs to?
  - which permissions the group has?
  - which permissions others have?

- Use the **chmod** command to change the permissions for your test files. ONLY DO THESE OPERATIONS ON YOUR TEST FILES – OTHERWISE YOU COULD STOP YOUR ACCOUNT FROM WORKING!

Before you do so, scan the **chmod** man page:

```
man chmod
```

List the file before and after each change and determine how the command changes the file permissions:

```
ls -l test2
chmod o-r test2
ls -l test2
ls -l test1
chmod g+w test1
ls -l test1
ls -l newtest3*
chmod go-r newtest3*
ls -l newtest3*
```

- Try **chmod** with numerical access specifications:

```
ls -l test2
chmod 754 test2
ls -l test2
```

Searching using grep

I have put a data file in my directory for you to copy over and use for this lab:

```
cp ~pszgtr/uk-500.txt .
```

Note the use of tilde (~) to access another user's home directory. Also note the dot at the end of the command – this tells the shell to put the copied file in the current directory (i.e. yours) as dot means "current directory" to the shell.

View the file uk-500.txt in your shell:

```
more uk-500.txt
```

You will see it contains a list of names and addresses.

Using grep

Now search this file for all lines containing people who live in Nottingham. There are two variations to try, one with quotes and one without:

```
more uk-500.txt | grep Nottingham
more uk-500.txt | grep “Nottingham”
```

Here you are using the **more** command to show the contents of the file and then piping the output of this to the **grep** command to search for the string Nottingham.

You will see that the output is the same. In the first example, Nottingham is not in quotes, but this is ok because it is not a regular expression and there are no characters that might be interpreted differently by the shell.

You can also use **grep** on its own (without the **more** command) to search for this string:

```
grep Nottingham uk-500.txt
```

In this case you are giving **grep** two arguments – the string to search for and the name of the file in which to look. So **grep** is taking its input from the file uk-500.txt. In the previous command where we are piping from **more**, **grep** is taking its input as whatever **more** ouputs.

Using quotes with grep

Now try the following two commands:

```
grep “West Ward” uk-500.txt
grep West Ward uk-500.txt
```

Can you see what has happened?

In the first command we have used quotes around the name “West Ward” and the search has returned addresses containing that string.

In the second command we haven’t used any quotes. The shell has therefore tried to interpret the space “ ” character and not understood it. It has therefore given an error and then executed the **grep** command with just the string “West”. Scroll back up and you can see the error that was produced by the shell, above all the output from the grep command.

File redirection (redirecting input and output)

Now what happens when you do the following: Why?

```
grep West Ward uk-500.txt > myAddresses
```

You will see that you get an error shown on the screen. This is because the error has been sent to STDERR (defaults to the screen) whereas we have redirected STDOUT (the output of the command grep) to the file myAddresses. You can see this if you do the following:

```
more myAddresses
```

Now redirect both channels, STDOUT and STDERR to the file myOutput:

```
grep West Ward uk-500.txt > myAddresses 2>&1
```

Here we have asked the shell to redirect STDOUT (the output of the **grep** command) to the file myAddresses and to also redirect the STDERR channel to the same place as STDOUT (i.e. redirect channel 2 to the address of where channel 1 is going).

If you look at the file myAddresses you will see that the error message is at the top of the file and then the STDOUT output is below this. Note the file has been ***overwritten*** with this new content.

You can redirect STDOUT to one file and STDERR to another as in the following example:

```
grep West Ward uk-500.txt > myAddresses 2> myErrors
```

Piping several commands together

Now search the file for all lines containing the city Manchester:

```
more uk-500.txt | grep “Manchester”
```

Now adapt this command to count the number of lines containing this string:

```
more uk-500.txt | grep “Manchester” | wc -l
```

You can see that piping enables the output of each command to be tailored by the next command. If we wish we can use file redirection to save the result to a file:

```
more uk-500.txt | grep “Manchester” | sort > manchester.txt
```

This command searches uk-500.txt for all lines containing the text Manchester and sorts them into alphabetical order before storing the result in the file manchester.txt

You can search for either one string or another. For this example we are using **egrep** which stands for “extended grep”. *(Note **egrep** has the same effect as using **grep -E** so we could use either here.)*

With **egrep** you can use the pipe symbol (|) to mean “or”, e.g.:

```
more uk-500.txt | egrep “Nottingham|Manchester”
```

Note, if you try this command without the quotes, you get an error!

```
more uk-500.txt | egrep Nottingham|Manchester
```

What has happened here?

Answer: Because we left the quotes out, the shell thought that it should execute the command “egrep Nottingham” and then it thought that the “|” was a pipe to another command “Manchester” which of course confused it ☺ So we got an error from the shell. Pipe means different things to the shell and to **egrep**.

Here is another version of the command that will confuse the shell:

```
more uk-500.txt | egrep [Nottingham|Manchester]
```

```
grep: Unmatched [ or [^
bash: Manchester]: command not found...
```

Notice that we get an error from the shell and an error from **egrep**:

Because we left the quotes out, the shell thought that it should execute the command “egrep [Nottingham” and then it thought that the “|” was a pipe to another command “Manchester]”. So the shell wasn’t happy. But we also got an error from **egrep** because it didn’t understand why it got an opening bracket “[” but no closing bracket “]” (because the shell hadn’t let it have the “|Manchester]” part. So **egrep** wasn’t happy either.

You can search for strings or patterns at the beginning or end of the line. First try searching for the string “Marg”…

```
more uk-500.txt | grep Marg
```

…and then search for lines that contain the string “Marg” at the start of the line only:

```
more uk-500.txt | grep “^Marg”
```

Escaping using \

```
grep "St." uk-500.txt
```

This will return lines containing **St** followed by any single character (because grep interprets the dot to mean any single character). So it will return all lines containing the text **St**

But supposing we want to search for **St.** (with a dot) meaning street, rather than **St** (without a dot) being part of a name (e.g. Stafford). The above command would return both.

To be more specific try the following command:

```
grep "St\." uk-500.txt
```

Here, we have escaped the dot which forces grep to interpret the text literally and not interpret the dot as part of a regular expression.

Using $ to access the value stored in a variable

Create a variable, count, which has the value 23, as follows:

```
export count=23
```

This is just the same as creating a variable inside a program, except it is done in the shell and the shell has access to its value.

(NOTE, this is one example of a command that won't work in the same way in different types of shell. For example, if you were running csh, this command would return an error. To do the same thing in csh, you would have to type **setenv count 23** ☺)

Now try the following:

```
grep $count uk-500.txt
```

What happens?

Instead, try the following command:

```
grep '$count' uk-500.txt
```

What happens now? The reason for this is that we have used single quotes which have prevented the shell from interpreting $count as anything other than the literal text, so all it does is pass it to **grep**. So **grep** searched for "$count" (rather than the substituted value of 23) and found nothing.

Using back prime (`) with single or double quotes

We had the following command in one of the lectures which you should try now:

```
echo "You have `ls | wc -l` files in `pwd`"
```

You will see we have used double quotes around the argument to the **echo** command. Within these double quotes are two unix commands surrounded by back primes. Because double quotes have been used, the shell is allowed to interpret the contents of what is within the back primes and the commands are executed and the results printed to the screen along with the other text that **echo** is outputting. This is what we mean when we say command substitution can occur.

Now try the command with single quotes instead of double quotes:

```
echo 'You have `ls | wc -l` files in `pwd`'
```

Here you can see that command substitution hasn't occurred. This is because the single quotes have prevented the shell from executing the commands. Everything put within single quotes is taken literally (not interpreted).

The next command will have the same effect as this, but we are using double quotes again, and instead escaping each back prime with a backslash:

```
echo "You have \`ls | wc -l\` files in \`pwd\`"
```

Here, the double quotes won't prevent the shell from interpreting what is in the back primes BUT the back primes are not considered to be back primes by the shell because we have escaped each one with a backslash \. The back primes are instead taken literally and printed to the screen by the **echo** command.

The following is an example of use of double quotes where one of the commands is executed and one is not:

```
echo "You have \`ls | wc -l\` files in `pwd`"
```

Here you can see how using double quotes allows the shell to interpret some things, but you still have the ability to escape characters that you want to be treated literally by using backslash.

It is possible to treat the backslash character itself literally, but you have to escape it with another backslash ☺:

```
echo “I am trying to print \\ to the screen”
```

What happens if you don't escape the backslash?

```
echo “I am trying to print \ to the screen”
```

…the shell looks at the \ and assumes it is going to get further input and so doesn't give it to **echo** to print. Instead you get a prompt “**>**” and it waits for more input. Type **control C** to quit out of this.

However, if we use single quotes with this last command instead, we don't get an error:

```
echo ‘I am trying to print \ to the screen’
```

This is because the single quotes have prevented the shell interpreting anything inside the quotes, other than the closing quote and so the “\” is printed to the screen.