Laboratory Handout # 2: Introduction to vi and Shell Scripting

Objectives:
At the end of this meeting, students should be able to:

- Learn how to use the vi editor; and
- Write bash scripts.

Introduction

- It is useful to learn because it is universally available
- It uses standard alphanumeric keys for commands
- It uses very few system resources

Basic Text Input and Navigation in vi

- Command Mode
  - characters typed perform actions
  - mode after starting vi
  - pressing `i` switches vi to Input Mode
  - `h, j, k, l` : left, up, down, right
  - `^, $`: beginning of line and end of line
  - `w, b`: beginning of next word, beginning of previous word
  - `1` then `G`: top of the document
  - `G`: bottom of the document
  - `^F, ^B` : skip forward a page, go back a page
  - `55G`: go to line number 55
  - `x`: delete the current character
  - `dw`: delete the next word
  - `d4w`: delete the next four words
  - `dd`: delete the next line
  - `4dd`: delete the next for lines
  - `d$`: delete to the end of the line
  - `dG`: delete to the end of the document
  - `u`: undo last change
- Input Mode
  - characters typed are inserted or overwrite existing text
  - pressing `ESC` switches vi to Command Mode

Moving and Copying Text in vi

- vi uses buffers to store text that is deleted. There are 9 buffers and an undo buffer
- To cut and paste, delete the text using `dd` then move to the line where you want to paste the text then press `p`.
- To copy and paste, “yank” the text using `<n>yy` to copy `n` lines then move to the line where you want to paste then press `p`.

Searching for and Replacing Text in vi

- Using regular expressions, we can search for text in command mode.
- To search forward, type `/` and then a regular expression and press enter
- To search backward, type `?`
- To find the next text that matches you regular expression press `n`.
- To search and replace <pattern1> with <pattern2> type `:%s/pattern1/pattern2/g`

Other Commands

- `:set number` – displays line number
- `:set  nonumber` – disables the display of line number
- `:w` – saves a file
- `:wq` – save and quit
- `:q!` - force quit without saving
- `:e <filename>` - start editing another file
- `!<shellcommand>` - execute a shell command then return to vi
- `:!echo %` - prints the name of the current file
- `.` - repeats the last command

How to write shell script

Following steps are required to write shell script:

- Use the vi editor to write shell script.
- After writing shell script set execute permission for your script as follows

  S*yntax:*

  ```
  chmod permission your-script-name
  ```

  Examples:

  ```
  $ chmod +x your-script-name
  $ chmod 755 your-script-name
  ```

- Execute your script as

  S*yntax:*

  ```
  bash your-script-name
  sh your-script-name
  ./your-script-name
  ```

  Examples:

  ```
  $ bash bar
  $ sh bar
  $ ./bar
  ```

  Syntax:

  ```
  . command-name
  ```

  Example:

  ```
  $ . foo
  ```

How Shell Locates the file (My own bin directory to execute script)

Tip: For shell script file try to give file extension such as .sh, which can be easily identified by you as shell script.

A simple shell script
A shell script is little more than a list of commands that are run in sequence. Conventionally, a shellscript should start with a line such as the following:

```
#!/bin/bash
```

A simple example
Here's a very simple example of a shell script. It just runs a few simple commands

```
#!/bin/bash
echo "hello, $USER. I wish to list some files of yours"
echo "listing files in the current directory, $PWD"
ls  # list files
```

Variables
Any programming language needs variables. You define a variable as follows:

```
X="hello"
```

and refer to it as follows:

```
$X
```

More specifically, `$X` is used to denote the value of the variable X. Some things to take note of regarding semantics:

- bash gets unhappy if you leave a space on either side of the = sign. For example, the following gives an error message:

```
X = hello
```

- while I have quotes in my example, they are not always necessary. where you need quotes is when your variable names include spaces. For example,

```
X=hello world # error
X="hello world" # OK
```

Single Quotes versus double quotes

```
#!/bin/bash
# -n option stops echo from breaking the line
echo -n '$USER='
echo "$USER"
# this does the same thing as the first two lines
echo "\$USER=$USER"
```

Using Quotes to enclose your variables
Sometimes, it is a good idea to protect variable names in double quotes. This is usually the most important if your variables value either (a) contains spaces or (b) is the empty string. An example is as follows:

```
#!/bin/bash
X=""
# -n tests to see if the argument is non empty
if [ -n $X ]; then
     echo "the variable X is not the empty string"
fi
```

A better script would have been:

```
#!/bin/bash
X=""
# -n tests to see if the argument is non empty
if [ -n "$X" ]; then
     echo "the variable X is not the empty string"
fi
```

Variable Expansion in action

```
#!/bin/bash
LS="ls"
LS_FLAGS="-al"

$LS $LS_FLAGS $HOME
```

This looks a little enigmatic. What happens with the last line is that it actually executes the command

```
ls -al /home/user
```

That is, the shell simply replaces the variables with their values, and then executes the command.

Using Braces to Protect Your Variables

OK. Here's a potential problem situation. Suppose you want to echo the value of the variable X, followed immediately by the letters "abc". Question: how do you do this ? Let's have a try :

```
#!/bin/bash
X=ABC
echo "$Xabc"
```

This gives no output. What went wrong? The answer is that the shell thought that we were asking for the variable `Xabc`, which is uninitialised. The way to deal with this is to put braces around `X` to seperate it from the other characters. The following gives the desired result:

```
#!/bin/bash
X=ABC
echo "${X}abc"
```

The Test Command and Operators

The command used in conditionals nearly all the time is the test command. Test returns true or false (more accurately, exits with 0 or non zero status) depending respectively on whether the test is passed or failed. It works like this:

```
test operand1 operator operand2
```

for some tests, there need be only one operand (operand2) The test command is typically abbreviated in this form:

```
[ operand1 operator operand2 ]
```

To bring this discussion back down to earth, we give a few examples:

```
#!/bin/bash
X=3
```

```
Y=4
empty_string=""
if [ $X -lt $Y ]        # is $X less than $Y ?
then
     echo "\$X=${X}, which is smaller than \$Y=${Y}"
fi

if [ -n "$empty_string" ]; then
     echo "empty string is non_empty"
fi

# test to see if ~/.fvwmrc exists
if [ -e "${HOME}/.fvwmrc" ]; then
     echo "you have a .fvwmrc file"
     # is it a symlink ?
     if [ -L "${HOME}/.fvwmrc" ]; then
          echo "it's a symbolic link
     # is it a regular file ?
     elif [ -f "${HOME}/.fvwmrc" ]; then
          echo "it's a regular file"
fi
else
        echo "you have no .fvwmrc file"
fi
```

Some pitfalls to be wary of

The test command needs to be in the form `operand1<space>operator<space>operand2` or `operator<space>operand2` , in other words you really *need* these spaces, since the shell considers the first block containing no spaces to be either an operator (if it begins with a '-') or an operand (if it doesn't). So for example; this

```
if [ 1=2 ]; then
        echo "hello"
fi
```

gives exactly the "wrong" output (ie it echos "hello", since it sees an operand but no operator.) Another potential trap comes from not protecting variables in quotes. We have already given an example as to why you *must* wrap anything you wish to use for a -n test with quotes. However, there are a lot of good reasons for using quotes all the time, or almost all of the time. Failing to do this when you have variables expanded inside tests can result in *very* wierd bugs. Here's an example: For example,

```
#!/bin/bash
X="-n"
Y=""
if [ $X = $Y ] ; then
        echo "X=Y"
fi
```

This will give misleading output since the shell expands our expression to

```
[ -n = ]
```

and the string "=" has non zero length.

A brief summary of test operators

Here's a quick list of test operators. It's by no means comprehensive, but its likely to be all you'll need to remember (if you need anything else, you can always check the bash manpage ...)

operator	produces true if...	number of operands
`-n`	operand non zero length	1
`-z`	operand has zero length	1
`-d`	there exists a directory whose name is *operand*	1
`-f`	there exists a file whose name is *operand*	1
`-eq`	the operands are integers and they are equal	2
`-neq`	the opposite of -eq	2
`=`	the operands are equal (as strings)	2
`!=`	opposite of =	2
`-lt`	*operand1* is strictly less than *operand2* (both operands should be integers)	2
`-gt`	*operand1* is strictly greater than *operand2* (both operands should be integers)	2
`-ge`	*operand1* is greater than or equal to *operand2* (both operands should be integers)	2
`-le`	*operand1* is less than or equal to *operand2* (both operands should be integers)	2

Loops

For loops

```
#!/bin/bash
for X in red green blue
do
    echo $X
done
```

The for loop iterates the loop over the space seperated items. Note that if some of the items have embedded spaces, you need to protect them with quotes. Here's an example:

```
#!/bin/bash
colour1="red"
colour2="light blue"
```

```
colour3="dark green"
for X in "$colour1" $colour2" $colour3"
do
    echo $X
done
```

Can you guess what would happen if we left out the quotes in the for statement ? This indicates that variable names should be protected with quotes unless you are pretty sure that they do not contain any spaces.

Globbing in for loops

The shell expands a string containing a * to all filenames that "match". A filename matches if and only if it is identical to the match string after replacing the stars * with arbitrary strings. For example, the character "*" by itself expands to a space seperated list of all files in the working directory (excluding those that start with a dot ".") So

```
echo *
```

lists all the files and directories in the current directory.

```
echo *.jpg
```

lists all the jpeg files.

```
echo ${HOME}/public_html/*.jpg
```

lists all jpeg files in your public_html directory.
As it happens, this turns out to be very useful for performing operations on the files in a directory, especially used in conjunction with a for loop. For example:

```
#!/bin/bash
for X in *.html
do
     grep -L '<UL>' "$X"
done
```

While Loops

```
#!/bin/bash
X=0
while [ $X -le 20 ]
do
     echo $X
     X=$((X+1))
done
```

Command Substitution

Command Substitution is a very handy feature of the bash shell. It enables you to take the output of a command and treat it as though it was written on the command line.

There are two means of command substitution: **brace expansion** and **backtick expansion**.

Brace expansion works as follows: $(*commands*) expands to the output of *commands* This permits nesting, so *commands* can include brace expansions
Backtick expansion expands `*commands*` to the output of *commands*

An example is given:

```
#!/bin/bash
files="$(ls)"
web_files=`ls public_html`
echo "$files"      # we need the quotes to preserve embedded
newlines in $files
echo "$web_files"  # we need the quotes to preserve newlines
X=`expr 3 \* 2 + 4` # expr evaluate arithmatic expressions.
man expr for details.
echo "$X"
```

Arrays

To declare an array in bash

```
array=( one two three )
files=( "/etc/passwd" "/etc/group" "/etc/hosts" )
limits=( 10, 20, 26, 39, 48 )
empty=()
```

To print an array

```
printf "%s\n" "${array[@]}"
printf "%s\n" "${files[@]}"
printf "%s\n" "${limits[@]}"
```

To iterate through array values

```
for i in "${arrayName[@]}"
do
:
# do whatever on $i
done
```

Other constructs

```
${arr[*]}           # All of the items in the array
${!arr[*]}          # All of the indexes in the array
arr[*]}             # Number of items in the array
${#arr[0]}          # Length of item zero
```

Gettting Input

Use the read command

```
     read [options] NAME1 NAME2 … NAMEN
```

```
Example
     read X
     echo $X
```

Reference

- http://www.doc.ic.ac.uk/~wjk/UnixIntro/Lecture6.html
- http://www.panix.com/~elflord/unix/bash-tute.html
- http://freeos.com/guides/lsst/

Exercise # 2: Shell Scripting

1. **sorting** – sorts an array of n integers in ascending order
   Input: `$./sort.sh <n> <n integers>`
   Example: `$./sort.sh 10 10 9 8 7 6 5 4 3 2 1`
   `1 2 3 4 5 6 7 8 9 10`

2. Write a **calculator** script (case statements):

```
****CALCULATOR****
[1] Add
[2] Subtract
[3] Multiply
[4] Divide
[5] Factorial
[6] Quit
```

3. Look for any initialization script in `/etc/init.d`. Study the script and describe in a few sentences what you think the script does.

4. Write a script called `checking` that displays information about a specified user. The script should:
   - display a prompt asking for the username
   - read the user input
   - `finger` the user
   - display any results from the `who` command about this user only
   - display any results from the `ps` command about this user only

   Test your script using your own username. Then, login on another console as floopy, switch back to your own console and make sure that your script displays the correct data about floopy.

5. Code a script called `dirchk` that displays data about the current directory. The script should:

   - display a count of the number of subdirectories of this directory.
   - display a count of the number of files in the directory.
   - list all of the files in the directory that are zero length (use an option of the find command to do this)
   - use `du` to display the amount of storage space used by this directory

   To test your script, you should create some subdirectories, some files that are zero length using `touch` and some files that are not zero length using vi, redirection or `cp`.