1. Semester and Year	: 2012 FALL
2. Course Number	: CS-227
3. Course Title	: Linux & System Software
4. Work Number	: LA-07
5. Work Name	: Lab 07
6. Work Version	: Version1
7. Long Date	: Sunday, 14, October, 2012
8. Author(s) Name(s)	: Jake Waffle

Task 1.8: Manage the Shell Environment

In this task I'll be learning how to change environment variables for the terminal. Then I will be learning how to make those changes permanent.

Adjusting Your Shell Prompt

To change your shell prompt, is a mighty simple task. Just one variable is needed to be changed in order to do so. The variable is called, “PS1” and can be changed with a simple entry into the terminal.

$ PS1=”>> “

The above command is what I entered into my terminal to change the prompt. But although it changes the current terminal's prompt, a derived program will not inherit the prompt. To make the prompt inherited by a derived program, another simple command can be used.

$ export $PS1

Above is the simple command that allows derived programs to inherit the prompt.

Setting a Program-Specific Environment Variable

The variable that the book goes over is $EDITOR (Smith, page 40.) And it is said that when a program goes to automatically open a text-editor, it will use this variable to find the text-editor that will be used. Changing the $EDITOR environment variable can be done with the *export* command like so.

$ export EDITOR=”/usr/bin/vi”

Because the programs specifically uses variable $EDITOR to find the text-editor that the user prefers, just changing the variable for the current bash session will not be enough to change the text-editor being opened by programs that use $EDITOR.

Making Your Changes Permanent

Bash has a good amount of different configuration files. The only one listed by the book that I recognize from my dealings with Arch Linux is “~/.bashrc” (Smith, page 41.) If I remember correctly, within the ~/.bashrc configuration file you can set the $Path variable as well as some other variables I can't quite remember. That file is also said to be a common file by the book. These configuration files are pretty much just bash scripts that get run on startup. To change how your shell environment is permanently, modifying or adding to one of the bash configuration files is all that needs to be done.

Exploring Other Environment Variables

It is possible for a user to add in environment variables, but if they aren't in use by bash or a program then they won't really be necessary.

Common Environment Variables

-$USER -A variable that stores your current username.

-$SHELL -A variable that holds the path to the current command shell.

-$PWD -The present working directory.

-$HOSTNAME -The computer's TCP/IP hostname.

-$PATH -The path variable holds the paths that programs will look through for files that they may need.

-$HOME -This stores the path to the home directory.

$LD_LIBRARY_PATH -The book says that a few programs use this to store the directory of included files for the program (Smith, page 43.)

$PS1 -This variable stores a string that the shell uses as its prompt.

$DISPLAY -This holds something like an ID for the display used by the X window system.

$EDITOR -This variable holds the directory of the text-editor that is preferred by the current user.

Task 1.9: Write Basic Scripts

In this task, I'll be learning how to create scripts and modify their file permissions in order to make the scripts executable.

Beginning a Shell Script

Shell scripts are written in text-editors as normal text-files according to the book (Smith, page 45.) But the first line of the script indicates that the file is a script and denotes the path for the shell. Then after a script is written, one can use a command like

$ chmod a+x script

to change the file permissions to executable (indicated above by option +x) for all users (indicated above by option a.) So with the script being executable, we can now use

$ ./script

to execute the script. The “/.” prefix tells the shell to look inside the present working directory for the script.

Using Commands

When writing shell scripts the book says that it is allowed to use internal and external commands (Smith, page 46.) External commands are familiar in to me so far in that they have been used for manual tasks in the terminal. It is also possible for one to run a program by listing its path name in the script. External commands are actually mostly programs and grep, one of the external commands given by the book can be found in /usr/bin along with kill and some other commands that can be found with command

$ help

Using Variables

The book lists three types of variables that can be used by scripts: parameter, regular and environment variables (Smith, page 48.) The parameter variables are denoted by a '$' with a number following it. And parameter variables will be given to the script when the user runs it. Regular variables also when accessed need to have a '$' before their name. But when looking at listing 1.4 in the book it looks like when assigning a commands output to a variable, the '$' isn't needed (Smith, page 49.) Environment variables were explained to me in task 1.8. They are pretty much just variables that scripts and programs may need.

Using Conditional Expressions

Bash allows for pretty mainstream conditional statements and loops. The if statement allows one to only execute the following block (denoted by prefix “then”) if the expression given is true. The expression can be represented with the output of a command, a comparison of variables or a filename with a option prefix like (-f filename.)

```
if [comand]
    then
        commands
fi
```

For loops can also be used to loop through a set of items outputted by a command. They seem to work a lot like python's for loops.

```
For I in [command]
    do
        commands
done
```

While loops are also an option, but not much is said about them. I'm assuming that they mainly just use a comparison of variables for the condition.

```
While [condition]
do
        commands
done
```

Using Functions

Functions I think are pretty straight-forward, they define a block of commands that will be executed when the function is called. If scripts are anything like programs written in C, then the function has to precede the point that it's called or referenced. Functions can be defined with the keyword 'function' at the beginning followed by its name and a couple of parenthesis and some brackets like the following.

```
function functionName() {
        commands
}
```

References

-Roderick W. Smith. linux administrator StreetSmarts. Induanapolis, Indiana: Wiley

Publishing, Inc,

2007