Session 2 - Shells and Programming Languages

Shells

The language of modern computers is the binary numbering system (1's and 0's). All the processing that computers do and the data they work with is in binary. Linux shells are applications which allow humans (who natively do not speak binary) with computers (who natively do speak binary). A shell is one of the tools we have which allow humans to tell the computer what they want it to do. A shell takes human readable commands and translates them into commands the kernel can understand and process.

When you open a terminal window on Linux, or when you remotely log into a Linux machine using SSH a shell gets started. From this shell you can type commands and interact with the computer.

There are several shells available on Linux. To see which shells you have on your Linux system type `cat /etc/shells` in a terminal windows and then press enter. We will concentrate on `bash`, the most popular shell on Linux.

Basic Commands

Command	Result
`pwd`	Print working directory. The directory you are currently in
`cd` *`optional path`*	Change directory to your home directory or the directory given after command
`ls` *`optional path`* Other options: `ls -l, ls -la, ls -lrt`	List the current directory contents or the contents of the directory given after command
`cat filename`	Print the contents of the given file
`date`	Prints the system time
`mv old_filename new_filename`	Renames a file
`cp original_file new_file`	Creates a copy of a file with a new name
`rm filename`	Deletes a file
`echo "some words"`	Prints a string to standard output
`mkdir some_dir, rmdir some_dir`	Creates or removes a directory

`touch`	Creates an empty file (or update last change time to current for existing files)
`df -h`	Show the amount of storage on a filesystem
`grep`	A command line search utility
`man command_name`	Prints the manual for the command given
`whoami`	Prints the username
`printenv`	Prints the environment variables in your shell

File Matching metacharacters

Symbol	Result
`*`	Matches any number of characters
`?`	Matches exactly one character
`[...]`	Matches any one of the characters in brackets which can be a comma separated list or a hyphen-separated range

In-class activity 1

In a terminal type

```
cd ~   this command puts you in your home directory
mkdir wordtest
cd wordtest
touch rat cat bat splat drat ration duration rats
ls ?at
ls *at
ls ??at
ls *ration
ls *at*
ls *[pt]*
echo "good morning" > greeting
echo "good evening" >> greeting
cat greeting
cd ..
mv wordtest testwords
rm -r testwords
sudo cat /etc/passwd
```

Find your account in the list and you will see your default shell at the end of the line.

The following are very good resources for learning Bash
http://www.tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html
http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html

Bash Environment Variables, Pipes, and Redirection

The Bash shell environment has variables which customize and enhance the operation of the shell. Environment variables are typically set in the files described next.

When you log into a system using a Desktop Environment and then open a terminal window you are creating a **non-login** shell. It is a non-login shell because you have already logged into the system. Non-login shells run the `~/.bashrc` file at startup. The tilde `~` is a shorthand way of expressing the path to your home directory.

When you open a terminal window you have a set of environment variables pre-set for you. These come from the `~/.bashrc` file. Modifying this file is one way you can customize your environment.
Example of a `~/.bashrc` file

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# User specific aliases and functions
PATH=$PATH:$HOME/bin

export PATH

# The PS1 variable is how you change your prompt
PS1='$PWD> '

# If I was running an Oracle database I would need something like
this
ORACLE_SID=testdb
export PS1 ORACLE_SID
```

When you remotely connect into a system you are creating a **login shell** and the `~/.bash_profile` file gets run at startup. Modifying this file is another way you can customize your environment. Example of a `~/.bash_profile` file

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs
```

In-class activity 2

Work through the `vipractice.txt` file. The instructor will step you through this introduction to the `vi` editor.

In-class activity 3

In a terminal window run the following commands:

`vi ~/.bash_profile`

add this line to the file

`export bash_profile_variable="this was set by the .bash_profile file"`

save the file and exit

```
sudo apt-get install ssh
Do you want to continue? [Y/n] y
ssh localhost
```

provide your password

`printenv | grep bash_profile_variable`

What you've done is log into your own machine on a loopback interface. And shown that when you log in remotely the `~/.bash_profile` file is read and interpreted.

Different ways to make the computer do what you want it to

1. You could write programs in a very low level language like assembly. For example to make the computer say "Hello World" you could do the following:
   Start a terminal window
   type: `vi helloworld.s`

```
.section .data
     hello:
          .ascii "Hello world!\n"     # output string

.section .text
     .globl _start
     _start:
     movl $4,%eax          # system call ID: sys_write
     movl $1,%ebx          # file descriptor for standard output
     movl $hello,%ecx      # string address
     movl $13,%edx         # length of the string
     int $0x80             # generate interrupt
     movl $1,%eax          # system call ID: sys_exit
     movl $0,%ebx          # exit code 0 success
     int $0x80             # generate interrupt
```

   assemble the program: `as -o helloworld.o helloworld.s`
   link the program: `ld -o helloworld helloworld.o`
   run the program: `./helloworld`

2. You could write programs in a higher level compiled language like C.
   Start a terminal window
   type: `vi helloworld.c`

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
}
```

   compile the program: `gcc -o helloworld helloworld.c`
   run the program: `./helloworld`

3. You could write programs in another higher level intermediate compiled language like Java.
   Start a terminal window
   type: `vi helloworld.java`

```
public class HelloWorld {
    public static void main(String[] args) {
        // Prints "Hello, World" to the terminal window.
        System.out.println("Hello, World");
    }
}
```

compile the program: `javac helloworld.java`
run the program: `java ./helloworld`

4. you could write programs in an interpreter like Bash
   Start a terminal window
   type: `printf "Hello World!\n"`

5. you could write programs for another interpreter like Python
   Start a terminal window
   type: `python`
   type: `print "Hello World!\n"`

Interpreted and Compiled Languages

In the examples above we saw:

Compiled into machine code languages - assembly and C

When code written in a language is compiled, its syntax is transformed into an executable form (machine code, 1's and 0's) before running. This is the original mode of compilation, and languages that are directly and completely transformed to machine-native code in this way may be called "truly compiled" languages.

Compiled into intermediate representations languages - Java

When code written in an intermediate representation language is compiled, its syntax is transformed into byte code (similar to assembly). The byte code must then be interpreted or further compiled to execute it. This is done by a Virtual Machine.

Interpreted languages - Bash and Python

When code written in a language is interpreted, its syntax is read and then executed directly, with no compilation stage. A program called an *interpreter* reads each program statement, following the program flow, then decides what to do, and does it.

https://en.wikipedia.org/wiki/High-level_programming_language

So why not just write everything in an interpreted language? ***Speed and Efficiency***
Compiled programs run much faster than interpreted programs and they use less memory.
What language do you think the Linux kernel is written in?

Session 2 Homework

Try variations on all the the commands located under Bash Basics, File Commands, and Directory Commands in the Bash Cheat Sheet. Save the outputs to a file to turn in to show what you tried. Make note of which programs are not installed and any other observations you have. Turn in your homework by emailing it to your instructor.