Bash Scripting

CS 580U - Fall 2017

The Shell

- The shell is an interactive command line interface for working with your computer
  - The windowed GUI that you are used to is just a wrapper for the shell
  - The shell is also called ‘Terminal’, and ‘Command Line’
- The Unix shell (sh) was the interface for the first version of the Unix OS.
  - Bash is an extension of the interface that adds features

Unix Shell

- Bourne shell (sh) was the first major shell
  - C and TC shell (csh and tcsh) had improved command interpreters, but were less popular than Bourne shell for programming
- Bourne Again shell (bash) was an improvement of Bourne shell and added features like history and tab completion
  - Other Bash-like shells: Korn shell (ksh), Z shell (zsh)
- Bash is the dominating Unix shell today
  - You can find the bash executable in /bin/bash on your linux machine
    - *When you run the terminal, if you are running bash, you are just executing /bin/bash*

Shell Features

- Different shells have different features, and you might want to switch between them to suit the work you are doing
  - BASH has the following features:
    - *tab auto-completion*
    - *directory shortcuts such as ~*
- BASH also has a scripting feature where you can write a script to run multiple shell commands
- The shell scripting language is a macro language
  - A macro language uses text replacement

Why use bash?

- Why not use Python or Ruby or some other full programming language?
  - Not dependant on version (Python3 vs Python2) and more integrated into the OS
  - Designed for system level operations
- In practice, use BASH when you need to run a repeated set of commands. If you start requiring a lot of logic, switch to a programming language

Sets of Commands

- What if I wanted to encrypted a file on my system, remove the original file, and create a flag file. What shell commands could I use to do this?
  - Flag File: a file that's only purpose is to let me know something happened
- I could do the following in the command line:
  - gpg -c example.c
    rm example.c
    touch example.log

Scripting Commands

- My previous encryption script required 3 commands:
  - gpg -c example.c
    rm example.c
    touch example.log
- These commands will be the same every time, so why not automate them?
  - I will need a variable for the file name, but otherwise, everything else is scriptable

Writing A Bash Script

- Traditionally, shell scripts end in the .sh extension
  - encryptlog.sh
- All of the shell scripts we'll see in this course begin with a shebang (#!). This is followed by the full path of the shell we'd like to use as an interpreter: /bin/bash
  - #! /bin/bash

    # This is the beginning of a shell script
    - *Any line that begins with a # (except the shebang) is a comment.*

Writing and Running the Script

- Write your commands just like you would in the shell
  - gpg -c example.c
    rm example.c
    touch example.log
- You can now run your script with:
  - sh encryptlog.sh
- Or you can mark it as an executable and run it directly
  - chmod +x encryptlog.sh
    ./encryptlog.sh

Variables

- Our filename will change, so we should make it a variable
  - MYFILE=”example”
    - *variables are in all caps by convention*
- Shell Scripts are whitespace sensitive
  - Notice there are no spaces around the assignment operator when creating the variable. This is essential.
- Now you can use your variable with the $
  - gpg -c $MYFILE.c
    rm $MYFILE.c
    touch $MYFILE.log

Command Line Arguments

- You can script any valid sequence of bash commands
  - however, it is more useful if you can take arguments form the shell
- When you invoke the script, you can pass in arguments
  - ./encryptlog.sh example
- You can access CLA in your script as $1, $2, $3, ... and so on
  - gpg -c $1.c
    rm $1.c
    touch $1.log

Special parameters

- $0
  - returns the name of the shell script running as well as its location in the file system
- $#
  - is the number of parameters passed
- $@
  - gives an array of words containing all the parameters passed to the script

Read

- The read command allows you to prompt for input and store it in a variable.
  - #!/bin/bash
    echo -n “Enter name of file to encrypt: ”
    read file
    echo “Type 'y' to remove the original file”
    rm -i $file
    - *Line 2 prompts for a string that is read in line 3. *
    - *Line 4 uses the interactive remove (rm -i) to ask the user for confirmation.*

Environmental Variables

- There are two types of variables
  - Local variables
  - Environmental variables
- Environmental variables are set by the system
  - They can usually be listed by using the “env” command
  - They can be explicitly set by using the “export” command
- Environmental variables hold special values.
  - echo $SHELL

    /bin/bash

Single vs Double Quoted Strings

- When assigning character data containing spaces or special characters, the data must be enclosed in either single or double quotes.
  - Using double quotes to show a string of characters will allow any variables in the quotes to be resolved
    - *var=“test string”*
      newvar=“Value of var is $var”
      echo $newvar
      Value of var is test string
  - Single quotes prints the string exactly as it is

Arithmetic Evaluation

- The let statement can be used to do mathematical functions:
  - let X=10+2*7
    echo $X
    - *24*
- An arithmetic expression can be evaluated by $[expression]
  - echo “$[123+20]”
    - *143*

Classwork Pt 1: Encryption Script

expressions

- An expression can be: String, Numeric, File, and Logical
  - String Comparisons:
    - *= #compare if two strings are equal*
    - *!= #compare if two strings are not equal*
    - *-n #evaluate if string length is greater than zero*
    - *-z #evaluate if string length is equal to zero*
  - Examples:
    - *[s1=s2] #true if s1 same as s2, else false*
    - *[s1 != s2] #true if s1 not same as s2, else false*
    - *[-n s1] #true if s1 has a length greater then 0, else false*
    - *[-z s2] #true if s2 has a length of 0, otherwise false*

Number Comparisons

- Numeric comparisons use flags
  - Number Comparisons
    - *-eq #compare if two numbers are equal*
    - *-ge #compare if one number is greater than or equal to a number*
    - *-le #compare if one number is less than or equal to a number compare*
    - *-ne #if two numbers are not equal*
    - *-gt #compare if one number is greater than another number*
    - *-lt #compare if one number is less than another number*
  - Examples:
    - *[n1 -eq n2]*
    - *[n1 -ge n2]*
    - *[n1 -le n2]*

file expressions

- Files have their own set of operators to make working with files easy:
  - Operators
    - *-d #check if path given is a directory*
    - *-f #check if path given is a file*
    - *-e #check if file name exists*
    - *-s #check if a file has a length greater than 0*
  - Examples
    - *[-d fname]*
    - *[-s fname]*

if statements

- Conditionals let us decide whether to perform an action or not by evaluating an expression.
  - if [expression] #Put spaces after [and before], and around the operators and operands.
    then
        statements
    elif [expression] #the elif (else if) and else sections are optional
    then
        statements
    else
        statements
    fi

for statements

- The for structure is used when you are looping through a range of variables.
  - for var in list
    do
        statements
    done
    - *statements are executed with var set to each value in the list*

- Example
  - #!/bin/bash
    let sum=0
    for num in 1 2 3 4 5
    do
        let “sum = $sum + $num”
    done
    echo $sum

while statements

- The while loop terminates as soon as the condition becomes false. If condition never becomes false, loop will never exit.
  - while expression
    do
        statements
    done

Debugging

- Bash provides two options which will give useful information for debugging
  - -x
    - *displays each line of the script with variable substitution and before execution*
  - -v
    - *displays each line of the script as typed before execution*
- Example
  - #!/bin/bash –v or #!/bin/bash –x or #!/bin/bash –xv

Classwork Pt 2: Encryption Script

Limitations

- Slow
  - Since everything comes from external commands, every command will go through a fork and exec. Hence expensive and slow.
- Difficult to do low level operations:
  - It’s quite difficult to, say, tokenize strings
- Dependency hell
  - It is not easy to just assume that an external command will do what we expect it to do. For instance, stuff like PATH variable plays a role. Also the exact implementation of the command may vary on different systems.