Introduction to GNU/Linux and Shell Scripting

Rameez Thonnakkal,
ssrameez@gmail.com/ssrameez@fsmk.org

Agenda

- Introduction
  - GNU/Linux and Shell
  - GNU/Linux Commands and Utilities
  - Basic Shell Scripting Structure
- Shell Programming
  - Variable
  - Operators
  - Logic Structures
- Examples
- Hands-on Exercises

Why Shell Scripting ?

- Shell scripts can be used to prepare input files, job monitoring, and output processing.
- Useful to create own commands.
- Save lots of time on file processing.
- To automate some task of day to day life.
- System Administration part can be also automated.

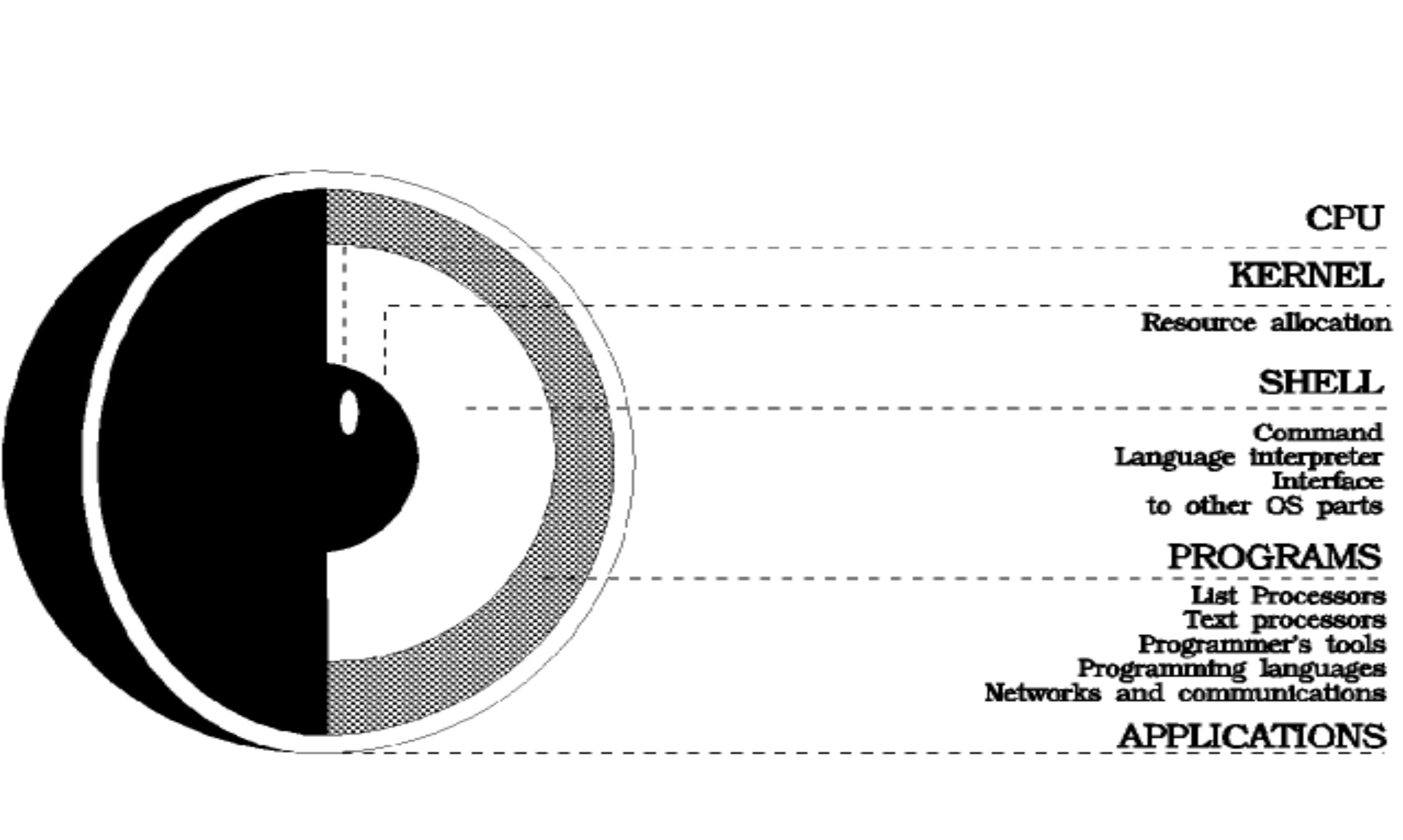
Objectives & Prerequisites

- **After this workshop, you should be:**
  - Familiar with GNU/Linux, Borne Shell, shell variables/ operators
  - Able to write simple shell scripts to illustrate programming logic
  - Able to write basic scripts

A little History of Unix and Linux

- Unix is a command line operating system developed around 1969 in the Bell Labs
- Unix is designed so that users can extend the functionality
  - To build new tools easily and efficiently
  - To customize the shell and user interface.
  - To string together a series of Unix commands to create new functionality.
  - To create custom commands that do exactly what we want.
- 1982 RMS started Free Software Movement.
- Around 1990 Linus Torvalds of Helsinki University started off a freely available academic version of Unix
- Linux is the Antidote to a Microsoft dominated future

UNIX/Linux Basic Architecture



CPU

KERNEL

Resource allocation

SHELL

Command
Language interpreter
Interface
to other OS parts

PROGRAMS

List Processors
Text processors
Programmer's tools
Programming languages
Networks and communications

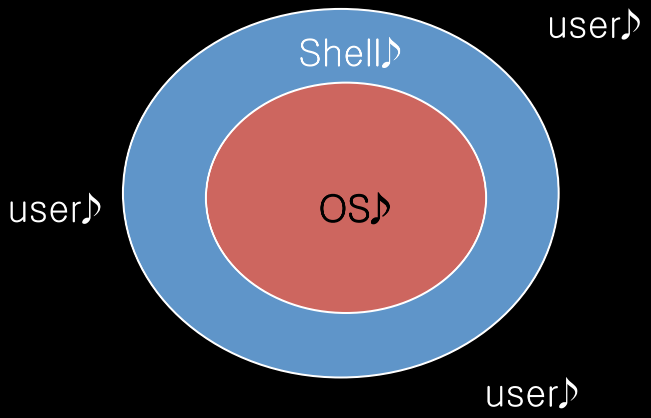
APPLICATIONS

What is a “Shell”?

- The “Shell” is simply *another program* on top of the kernel which provides a basic human-OS interface.
  - It is a command interpreter
    - Built on top of the kernel
    - Enables users to run services provided by the UNIX OS
  - In its simplest form, a series of commands in a file is a shell program that saves having to retype commands to perform common tasks.

- How to know what shell you use

```
echo $SHELL
```



UNIX Shells

- sh Bourne Shell (Original Shell) (*Steven Bourne of AT&T*)
- bash Bourne Again Shell (*GNU Improved Bourne Shell*)
- csh C-Shell (C-like Syntax)(*Bill Joy of Univ. of California*)
- ksh Korn-Shell (Bourne+some C-shell)(*David Korn of AT&T*)
- tcsh Turbo C-Shell (More User Friendly C-Shell).
- To check shell:
  - $ `echo $SHELL` (shell is a pre-defined variable)
- To switch shell:
  - $ exec shellname (e.g., `$ exec bash` or simply type `$ bash`)
  - You can switch from one shell to another by just typing the name of the shell. `exit` return you back to previous shell.

Which Shell to Use?

- sh (Bourne shell) was considered better for programming
- csh (C-Shell) was considered better for interactive work.
- tcsh and korn were improvements on c-shell and bourne shell respectively.
- bash is largely compatible with sh and also has many of the nice features of the other shells
- On many systems such as our LINUX clusters sh is symbolically linked to bash, /bin/sh -> /bin/bash
- We recommend that you use sh/bash for writing new shell scripts but learn csh/tcsh to understand existing scripts.
- Many, if not all, scientific applications require csh/tcsh environment (GUI, Graphics Utility Interface)
- **All Linux versions use the Bash shell (Bourne Again Shell) as the default shell**
  - **Bash/Bourn/ksh/sh prompt: $**

What is Shell Script?

- A **shell script** is a script written for the shell

- Two key ingredients
  - GNU/Linux commands
  - Shell programming syntax

A Shell Script Example

```
#!/bin/sh

`ls -l *.log| awk '{print $8}' |sed 's/.log//g' > file_list`

cat file_list|while read each_file
do
    babel -ig03 $each_file".log" -oxyz $each_file".xyz“

    echo '# nosymmetry integral=Grid=UltraFine scf=tight rhf/6-311++g** pop=(nbo,chelpg)'>head
    echo ' ' >>head
    echo "$each_file'  opt pop nbo chelp aim charges ' >> head
    echo ' ' >>head
    echo '0 1 ' >>head

    `sed '1,2d' $each_file.xyz >junk`
    input=./$each_file".com"
    cat head  > $input
    cat junk >> $input
    echo ' '  >> $input

done
/bin/rm ./junk ./head ./file_list
```

Capture everything..

- cd
- script fsmkcamp2014

GNU/Linux Commands

- File Management and Viewing
- Filesystem Mangement
- Help,Job/Process Management
- Network Management
- System Management
- User Management
- Printing and Programming
- Document Preparation
- Miscellaneous

- To understand the working of the command and possible options use (`man` command)
- Using the GNU Info System (`info`, info command)
- Listing a Description of a Program (`whatis` command)
- Many tools have a long-style option, `` `--help' ``, that outputs usage information about the tool, including the options and arguments the tool takes. Ex: *`whoami --help`*

File and Directory Management

- **cd** Change the current directory. With no arguments "cd" changes to the users home directory. (cd <directory path>)
- **chmod** Change the file permissions.

  Ex: `chmod 751 myfile` : change the file permissions to rwx for owner, rx for group and x for others (x=1,r=4,w=2)

  Ex: `chmod go=+r myfile` : Add read permission for the group and others (character meanings u-user, g-group, o-other, + add permission,-remove,r-read,w-write,x-exe)

  Ex: `chmod +s myfile` - Setuid bit on the file which allows the program to run with user or group privileges of the file.
- **chown** Change owner.

  Ex: chown <owner1> <filename> : Change ownership of a file to owner1.
- **chgrp** Change group.

  Ex: chgrp <group1> <filename> : Change group of a file to group1.
- **cp** Copy a file from one location to another.

  Ex: cp file1 file2 : Copy file1 to file2;   Ex: cp –R dir1 dir2 : Copy dir1 to dir2

File and Directory Management

- **ls** List contents of a directory.

  Ex: ls, ls –l , ls –al, ls –ld, ls –R

- **mkdir** Make a directory.

  Ex: mkdir <directory name> : Makes a directory

  Ex *mkdir –p /www/chache/var/log* will create all the directories starting from www.

- **mv** Move or rename a file or directory.

  Ex: mv <source> <destination>

- **find** Find files (find <start directory> -name <file name> -print)

  Ex: *find /home –name readme -print*

  Search for readme starting at home and output full path, “/home" = Search starting at the home directory and proceed through all its subdirectories; "-name readme" = Search for a file named readme "-print" = Output the full path to that file

- **locate** File locating program that uses the slocate database.

  Ex: locate –u to create the database,
  locate <file/directory> to find file/directory

File and Directory Management

- **pwd** Print or list the present working directory with full path.
- **rm** Delete files (Remove files). (rm –rf <directory/file>)
- **rmdir** Remove a directory. The directory must be empty. (rmdir <directory>)
- **touch** Change file timestamps to the current time. Make the file if it doesn't exist. (touch <filename>)
- **whereis** Locate the binary and man page files for a command. (whereis <program/ command>)
- **which** Show full path of commands where given commands reside. (which <command>)

File viewing and editing

- **vi** Editor with a command mode and text mode. Starts in command mode.
- **gedit** GUI Text Editor
- **tail** Look at the last 10 lines of a file.

  Ex: tail –f <filename> ;   Ex: tail -100 <filename>
- **head** Look at the first 10 lines of a file. (head <filename>)
- Cat
- less
- more

File and Directory Management

File compression, backing up and restoring

- **compress** Compress data.
- **uncompress** Expand data.
- **gzip** - zip a file to a gz file.
- **gunzip** - unzip a gz file.
- **tar** Archives files and directories. Can store files and directories on tapes.

  Ex: tar -zcvf <destination> <files/directories> - Archive copy groups of files. tar –zxvf <compressed file> to uncompress
- **zip** – Compresses a file to a .zip file.
- **unzip** – Uncompresses a file with .zip extension.
- **cat** View a file

  Ex: cat filename
- **cmp** Compare two files.
- **cut** Remove sections from each line of files.

File and Directory Management

- **diff** Show the differences between files.

  Ex: diff file1 file2 : Find differences between file1 & file2.

- **echo** Display a line of text.
- **grep** List all files with the specified expression.
  (*grep pattern <filename/directorypath>*)

  Ex: `ls –l |grep sidbi` : List all lines with a sidbi in them.

  Ex: grep " R " : Search for R with a space on each side

- **sleep** Delay for a specified amount of time.;#time sleep 2
- **sort** Sort a file alphabetically.
- **uniq** Remove duplicate lines from a sorted file.
- **wc** Count lines, words, characters in a file. (wc –c/w/l <filename>).
- **sed** stream **ed**itor, extremely powerful!
- **awk** an extremely versatile programming language for working on files

Useful Commands in Scripting

- `grep`
  - Pattern searching
  - Example: `grep ‘boo’ filename`
- `sed`
  - Text editing
  - Example: `sed 's/XYZ/xyz/g' filename`
- `awk`
  - Pattern scanning and processing
  - Example: `awk ‘{print $4, $7}’ filename`

Shell Scripting

- Start vi scriptfilename.sh with the line

  ```
  #!/bin/sh
  ```

- All other lines starting with # are comments.
  - make code readable by including comments
- Tell Unix that the script file is executable

  ```
  $ chmod u+x scriptfilename.sh
  $ chmod +x scriptfilename.sh
  ```

- Execute the shell-script

  ```
  $ ./scriptfilename.sh
  ```

My First Shell Script

```
$ vi myfirstscript.sh
    #! /bin/sh

    # The first example of a shell script
    directory=`pwd`
    echo Hello World!
    echo The date today is `date`
    echo The current directory is $directory
$ chmod +x myfirstscript.sh
$ ./myfirstscript.sh
    Hello World!
    The date today is Fri Jul 18 23:45:59 IST 2014
    The current directory is /home/rs/DevOps
```

Shell Scripts

- Text files that contain sequences of UNIX commands , created by a text editor
- No compiler required to run a shell script, because the UNIX shell acts as an interpreter when reading script files
- After you create a shell script, you simply tell the OS that the file is a program that can be executed, by using the chmod command to change the files' mode to be executable
- Shell programs run less quickly than compiled programs, because the shell must interpret each UNIX command inside the executable script file before it is executed

Commenting

- Lines starting with # are comments except the very first line where `#!` indicates the location of the shell that will be run to execute the script.
- On any line characters following an unquoted # are considered to be comments and ignored.
- Comments are used to;
  - Identify who wrote it and when
  - Identify input variables
  - Make code easy to read
  - Explain complex code sections
  - Version control tracking
  - Record modifications

Quote Characters

There are three different quote characters with different behaviour. These are:

“ : double quote, weak quote. If a string is enclosed in “ ” the references to variables (i.e *$variable*) are replaced by their values. Also back-quote and escape \ characters are treated specially.

‘ : single quote, strong quote. Everything inside single quotes are taken literally, nothing is treated as special.

` : back quote. A string enclosed as such is treated as a command and the shell attempts to execute it. If the execution is successful the primary output from the command replaces the string.

Example: `echo “Today is:” `date``

Echo

Echo command is well appreciated when trying to debug scripts.

Syntax : `echo {options}  string`

Options: `–e` : expand \ (back-slash) special characters

`–n` : do not output a new-line at the end.

String can be a “weakly quoted” or a ‘strongly quoted’ string. In the weakly quoted strings the references to variables are replaced by the value of those variables before the output.

User Input During Shell Script Execution

- As shown on the hello script input from the standard input location is done via the read command.
- Example

```
echo "Please enter three filenames:”
read  filea fileb filec
echo “These files are used:$filea  $fileb  $filec”
```

- Each read statement reads an entire line. In the above example if there are less than 3 items in the response the trailing variables will be set to blank ‘ ‘.
- Three items are separated by one space.

Hello script exercise continued…

- The following script asks the user to enter his name and displays a personalised hello.

```
#!/bin/sh
echo “Who am I talking to?”
read user_name
echo “Hello $user_name”
```

- Try replacing “ with ‘ in the last line to see what happens.

Debugging your shell scripts

- Generous use of the echo command will help.

- Run script with the –x parameter.
  E.g. sh –x ./myscript
  or set –o xtrace before running the script.

- These options can be added to the first line of the script where the shell is defined.
  e.g. #!/bin/sh -xv

Shell Programming

- **Programming features of the GNU/Linux shell:**
  - ***Shell variables*: Your scripts often need to keep values in memory for later use. Shell variables are symbolic names that can access values stored in memory**
  - ***Operators*: Shell scripts support many operators, including those for performing mathematical operations**
  - ***Logic structures*: Shell scripts support sequential logic (for performing a series of commands), decision logic (for branching from one point in a script to another), looping logic (for repeating a command several times), and case logic (for choosing an action from several possible alternatives)**

Variables

- **Variables are symbolic names that represent values stored in memory**
- **Three different types of variables**
  - Global Variables**: Environment and configuration variables, capitalized, such as HOME, PATH, SHELL, USERNAME, and PWD.**

    When you login, there will be a large number of global System variables that are already defined. These can be freely referenced and used in your shell scripts.
  - Local Variables

    Within a shell script, you can create as many new variables as needed. Any variable created in this manner remains in existence only within that shell.
  - Special Variables

    Reserved for OS, shell programming, etc. such as positional parameters $0, $1 …

A few global (environment) variables

SHELL	Current shell
DISPLAY	Used by X-Windows system to identify the display
HOME	Fully qualified name of your login directory
PATH	Search path for commands
MANPATH	Search path for <man> pages
PS1 & PS2	Primary and Secondary prompt strings
USER	Your login name
TERM	terminal type
PWD	Current working directory

Referencing Variables

Variable contents are accessed using ‘$’:

e.g.

```
$ echo $HOME

$ echo $SHELL
```

To see a list of your environment variables:

```
$ printenv
```

or:

```
$ printenv | more
```

Defining Local Variables

- As in any other programming language, variables can be defined and used in shell scripts.
- Unlike other programming languages, variables in Shell Scripts are not typed.
- Examples :

  a=1234   # a is NOT an integer, a string instead

  b=$a+1 # will not perform arithmetic but be the string '1234+1'

  b=`expr $a + 1 `   will perform arithmetic so b is 1235 now.

  Note : +,-,/,*,**, % operators are available.

  b=abcde # b is string

  b='abcde' # same as above but much safer.

  b=abc   def # will not work unless 'quoted'

  b='abc def' # i.e. this will work.

  IMPORTANT NOTE: DO NOT LEAVE SPACES AROUND THE =

Referencing variables --curly bracket

- Having defined a variable, its contents can be referenced by the $ symbol. E.g. `${variable}` or simply `$variable`. When ambiguity exists $variable will not work. Use `${ }` the rigorous form to be on the safe side.
- Example:

```
a='abc'
b=${a}def # this would not have worked without the{ } as
             #it would try to access a variable named adef
```

Variable List/Arrary

- To create lists (array) – round bracket

  ```
  $ set Y = (UNL 123 CS251)
  ```

- To set a list element – square bracket

  ```
  $ set Y[2] = HUSKER
  ```

- To view a list element:

  ```
  $ echo $Y[2]
  ```

- Example:

  ```
  #!/bin/sh
  a=(1 2 3)
  echo ${a[*]}
  echo ${a[0]}
  ```

  Results:

  ```
  1 2 3
  1
  ```

Positional Parameters

- When a shell script is invoked with a set of command line parameters each of these parameters are copied into special variables that can be accessed.
- $0 This variable that contains the name of the script
- $1, $2, ….. $n 1st, 2nd 3rd command line parameter
- $# Number of command line parameters
- $$ process ID of the shell
- $? Return code ‘exit code’ of the last command
- `Shift` command: This shell command shifts the positional parameters by one towards the beginning and drops $1 from the list. After a shift $2 becomes $1 , and so on … It is a useful command for processing the input parameters one at a time.

Example:

Invoke : `./myscript  one two buckle my shoe`

During the execution of `myscript` variables $1 $2 $3 $4 and $5 will contain the values *one, two, buckle, my, shoe* respectively.

Variables

- vi myinputs.sh

```
#! /bin/sh
echo Total number of inputs: $#
echo First input: $1
echo Second input: $2
```

- chmod u+x myinputs.sh

- myinputs.sh RAMEEZ FSMK 2014

```
   Total number of inputs: 3
   First input: RAMEEZ
   Second input: FSMK
```

Shell Programming

- **programming features of the UNIX shell:**
  - ***Shell variables***
  - ***Operators***
  - ***Logic structures***

Shell Operators

- **The Bash/Bourne/ksh shell operators are divided into three groups: defining and evaluating operators, arithmetic operators, and redirecting and piping operators**

Defining and Evaluating

- A shell variable take on the generalized form variable=value (except in the C shell).

```
$ set x=37; echo $x
37
$ unset x; echo $x
x: Undefined variable.
```

- You can set a pathname or a command to a variable or substitute to set the variable.

```
$ set mydir=`pwd`; echo $mydir
```

Pipes & Redirecting

- **Piping: An important early development in Unix , a way to pass the output of one tool to the input of another.**

  ```
  $ who | wc -l
  ```

  By combining these two tools, giving the wc command the output of who, you can build a new command to list the number of users currently on the system

- **Redirecting via angle brackets**: Redirecting input and output follows a similar principle to that of piping except that redirects work with files, not commands.

  ```
      tr '[a-z]' '[A-Z]' < $in_file > $out_file
  ```

  The command must come first, the *in_file* is directed in by the less_than sign (<) and the *out_file* is pointed at by the greater_than sign (>).

Arithmetic Operators

- **expr supports the following operators:**
  - **arithmetic operators: +,-,*,/,%**
  - **comparison operators: <, <=, ==, !=, >=, >**
  - **boolean/logical operators: &, |**
  - **parentheses: (,)**
  - **precedence is the same as C, Java**

Arithmetic Operators

- vi math.sh

```
#!/bin/sh
count=5
count=`expr $count + 1 `
echo $count
```

- chmod u+x math.sh
- math.sh

```
6
```

Arithmetic Operators

- vi real.sh

```
#!/bin/sh
a=5.48
b=10.32
c=`echo “scale=2; $a + $b” |bc`
echo $c
```

- chmod u+x real.sh
- ./real.sh

```
15.80
```

Arithmetic operations in shell scripts

var++ ,var-- , ++var , --var	post/pre increment/decrement
+ , -	add subtract
* , / , %	multiply/divide, remainder
**	power of
! , ~	logical/bitwise negation
& , |	bitwise AND, OR
&& ||	logical AND, OR

Shell Programming

- **programming features of the UNIX shell:**
  - ***Shell variables***
  - ***Operators***
  - ***Logic structures***

Shell Logic Structures

The four basic logic structures needed for program development are:

- **Sequential logic:** to execute commands in the order in which they appear in the program
- **Decision logic:** to execute commands only if a certain condition is satisfied
- **Looping logic:** to repeat a series of commands for a given number of times
- **Case logic:** to replace “if then/else if/else” statements when making numerous comparisons

Conditional Statements
(if constructs)

The most general form of the if construct is;

```
if command executes successfully
then
        execute command
elif this command executes successfully
then
        execute this command
        and execute this command
else
        execute default command
fi
```

However- elif and/or else clause can be omitted.

Examples

SIMPLE EXAMPLE:

```
    if date | grep “Fri”
    then
            echo “It’s Friday!”
    fi
```

FULL EXAMPLE:

```
    if   [  “$1”   ==   “Monday”   ]
    then
            echo “The typed argument is Monday.”
    elif [ “$1” == “Tuesday” ]
     then
            echo “Typed argument is Tuesday”
     else
            echo “Typed argument is neither Monday nor Tuesday”
    fi
```

```
# Note: =  or == will both work in the test but == is better for readability.
```

Tests

String and numeric comparisons used with test or [[   ]] which is an alias for test and also [   ] which is another acceptable syntax

string1 = string2	True if strings are identical
String1 == string2	…ditto….
string1 !=string2	True if strings are not identical
string	Return 0 exit status (=true) if string is not null
-n string	Return 0 exit status (=true) if string is not null
-z string	Return 0 exit status (=true) if string is null

int1 -eq int2	Test identity
int1 -ne int2	Test inequality
int1 -lt int2	Less than
int1 -gt int2	Greater than
int1 -le int2	Less than or equal
int1 -ge int2	Greater than or equal

Combining tests with logical operators || (or) and && (and)

Syntax: if cond1 && cond2 || cond3 …

An alternative form is to use a compound statement using the –a and –o keywords, i.e.

```
if cond1 –a cond22 –o cond3 …
```

Where cond1,2,3 .. Are either commands returning a value or test conditions of the form [ ] or test …

Examples:

```
if  date | grep “Fri”  &&  `date +’%H’` -gt 17
then
    echo “It’s Friday, it’s home time!!!”
fi

if [ “$a” –lt 0 –o “$a” –gt 100 ]     # note the spaces around ] and [
then
    echo “ limits exceeded”
fi
```

File enquiry operations

-d file	Test if file is a directory
-f file	Test if file is not a directory
-s file	Test if the file has non zero length
-r file	Test if the file is readable
-w file	Test if the file is writable
-x file	Test if the file is executable
-o file	Test if the file is owned by the user
-e file	Test if the file exists
-z file	Test if the file has zero length

All these conditions return true if satisfied and false otherwise.

Decision Logic

- **A simple example**

```
#!/bin/sh
if [ “$#” -ne 2 ] then
       echo $0 needs two parameters!
       echo You are inputting $# parameters.
 else
       par1=$1
       par2=$2
 fi
 echo $par1
 echo $par2
```

Decision Logic

Another example:

```
#! /bin/sh
#  number is positive, zero or negative
echo "enter a number:\c"
read number
if [ “$number” -lt 0 ]
then
      echo "negative"
elif [ “$number” -eq 0 ]
then
      echo zero
else
       echo positive
fi
```

Loops

Loop is a block of code that is repeated a number of times.

The repeating is performed either a pre-determined number of times determined by a list of items in the loop count (for loops) or until a particular condition is satisfied (while and until loops)

To provide flexibility to the loop constructs there are also two statements namely break and continue are provided.

for loops

Syntax:

```
        for  arg in list
        do
            command(s)
              ...
        done
```

Where the value of the variable ***arg*** is set to the values provided in the list one at a time and the block of statements executed. This is repeated until the list is exhausted.

Example:

```
        for i in 3 2 5 7
        do
              echo " $i times 5 is  $((  $i  * 5 ))  "
        done
```

The while Loop

- **A different pattern for looping is created using the while statement**
- **The while statement best illustrates how to set up a loop to test repeatedly for a matching condition**
- **The while loop tests an expression in a manner similar to the if statement**
- **As long as the statement inside the brackets is true, the statements inside the do and done statements repeat**

while loops

Syntax:

```
  while this_command_execute_successfully
    do
            this command
            and this command
    done
```

EXAMPLE:

```
    while test "$i" -gt 0      # can also be  while  [ $i > 0 ]
    do
            i=`expr $i - 1`
    done
```

Looping Logic

- Example:

```
#!/bin/sh
for person in Bob Susan Joe Gerry
do
   echo Hello $person
done
```

Output:

```
Hello Bob
Hello Susan
Hello Joe
Hello Gerry
```

- Adding integers from 1 to 10

```
#!/bin/sh
i=1
sum=0
while [ “$i” -le 10 ]
  do
  echo Adding $i into the sum.
  sum=`expr $sum + $i `
  i=`expr $i + 1 `
  done
echo The sum is $sum.
```

until loops

The syntax and usage is almost identical to the while-loops.

Except that the block is executed until the test condition is satisfied, which is the opposite of the effect of test condition in while loops.

Note: You can think of *until* as equivalent to *not_while*

Syntax:

```
until test
do
   commands ….
done
```

Switch/Case Logic

- **The switch logic structure simplifies the selection of a match when you have a list of choices**

- **It allows your program to perform one of many actions, depending upon the value of a variable**

Case statements

The case structure compares a string ‘usually contained in a variable’ to one or more patterns and executes a block of code associated with the matching pattern. Matching-tests start with the first pattern and the subsequent patterns are tested only if no match is not found so far.

```
#!/bin/sh
echo "Enter a number between 1 and 5. "
read NUM
case $NUM in
    1)    echo "one" ;;
    2)    echo "two" ;;
    3)    echo "three" ;;
    4)    echo "four" ;;
    5)    echo "five" ;;
    *) echo "INVALID NUMBER!" ;;
esac
```

Functions

- Functions are a way of grouping together commands so that they can later be executed via a single reference to their name. If the same set of instructions have to be repeated in more than one part of the code, this will save a lot of coding and also reduce possibility of typing errors.

SYNTAX:

```
functionname()
 {
      block of commands
  }
```

```
#!/bin/sh

  sum() {
     x=`expr $1 + $2`
     echo $x
     }


sum 5 3

echo "The sum of 4 and 7 is `sum 4 7`"
```

Take-Home Message

- Shell script is a high-level language that must be converted into a low-level (machine) language by GNU/ Linux Shell before the computer can execute it
- GNU/Linux shell scripts, created with the vi or other text editor, contain two key ingredients: a selection of Linux commands glued together by Shell programming syntax
- Linux shells are derived from the UNIX Bourne, Korn, and C/TCSH shells
- The shell supports numerous operators, including many for performing arithmetic operations
- The logic structures supported by the shell are sequential, decision, looping, and case

To Script or Not to Script

- Pros
  - File processing
  - Glue together compelling, customized testing utilities
  - Create powerful, tailor-made manufacturing tools
  - Cross-platform support
  - Custom testing and debugging

- Cons
  - Performance slowdown
  - Accurate scientific computing

Reference Books

Class Shell Scripting
http://oreilly.com/catalog/9780596005955/
LINUX Shell Scripting With Bash

http://ebooks.ebookmall.com/title/linux-shell-scripting-with-bash-burtch-ebooks.htm
Shell Script in C Shell
http://www.grymoire.com/Unix/CshTop10.txt
Linux Shell Scripting Tutorial
http://www.freeos.com/guides/lsst/
Bash Shell Programming in Linux
http://www.arachnoid.com/linux/shell_programming.html

- Advanced Bash-Scripting Guide
  http://tldp.org/LDP/abs/html/
- Unix Shell Programming

  http://ebooks.ebookmall.com/title/unix-shell-programmin ebooks.htm

Questions & Comments

Please direct comments/questions pertaining to this presentation to

fsmk-discuss mailing list. Else ssrameez@gmail.com

Hands-on Exercises

1. The simplest Hello World shell script – Echo command
2. Summation of two integers – If block
3. Summation of two real numbers – bc (basic calculator) command
4. Script to find out the biggest number in 3 numbers – If –elif block
5. Operation (summation, subtraction, multiplication and division) of two numbers – Switch
6. Script to reverse a given number – While block
7. A more complicated greeting shell script
8. Sort the given five numbers in ascending order (using array) – Do loop and array
9. Calculating average of given numbers on command line arguments – Do loop
10. Calculating factorial of a given number – While block
11. Optional: Write own shell scripts for your own purposes if time permits
12. Play https://github.com/viggyprabhu/Lord-of-the-shell game. Contribute to the same with additional levels.