# ITWS -1

## Monsoon 2017

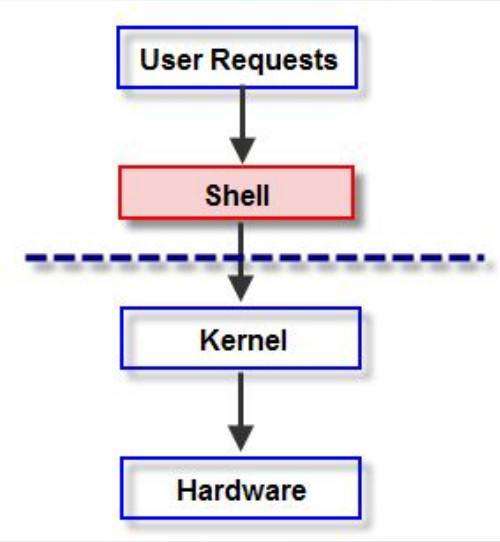
Kirti Garg and Lalit Mohan S

kirti@iiit.ac.in, lalit.mohan@iiit.ac.in

# Shell Scripting

# What is a Shell?

- A program (a.k.a. command-line interpreter) that allows the user to interact with the UNIX/Linux system.
  - Reads user's input.
  - Parses it (expansion/globbing...).
  - Works with the kernel to execute the command.



# Examples of Shells

- Bourne shell (sh)
- Bourne again shell (Bash)
- C shell (csh, tcsh)
- Korn shell (ksh)

# Gnu Bash


- Brainchild of Brian Fox.
- Replacement for Bourne shell (sh), hence Bourne-again shell.
- The default shell on GNU/Linux and Mac OS X.
- Features from ksh and csh. Some are:
- Command history, tab completion, internationalization…
- For more, man/info bash.

# What is a Shell Script?

- A regular text file that contains executable shell commands.
- Anything you can run normally on the command line can be put into a script.
- Similarly, anything you can put into a script can also be run normally on the command line and it will do exactly the same thing.

# Why write a Shell Script?

- Shell script can take input from user, file and output them on screen.
- Useful to create our own commands.
- Save lots of time.
- To automate some task of day today life.

# How to write a shell script

- Use any editor like vim or emacs to write a shell script.

- Give it execute permission.
  $ chmod +x <script_name>
  $ chmod 755 <script_name>

- Execute your script.
  $ bash <script_name>
  $ sh <script_name>
  $ ./<script_name>

# First Script

```
$ vim first.sh
```

```
#!/bin/bash
# My first script
echo "Hello World!"
```

(Shebang line, points to the interpreter)

```
$ chmod +x first.sh
$ ./first.sh
Hello World
```

# Variables

- No need to declare, no type (int, char...), case sensitive.
- Created when assigned a value.
  variable=value (no space in between. Why? )
- To read their values, precede them by a dollar sign ( $ ).

# Local vs Environment Variables

- Local variables exist in the current shell only.
- Environment variable are created and maintained by Linux itself.
- Environment/Global variables are passed (copies) to child processes but not local variables (i.e. their scopes differ).

  $SHELL, $PATH, $HOME, $USER, $PS1…

- The set, env/printenv and export commands

Caution: Do not modify System variables this can sometimes create problems.

# Console I/O

- The console I/O is generally done by the echo and read commands.

```
#!/bin/bash
# Ask the user for their name
echo Hello, who am I talking to?
read varname
echo It\'s nice to meet you $varname
```

# Quotes

Remove the special meaning of certain characters/words.

- Single Quote (‘ ’) - a.k.a. strong quoting
  Preserves the literal meaning of each character within it, except Itself.

- Double Quote (“ ”) - a.k.a. weak quoting
  Preserves the literal meanings of all characters within it except $ , ` , \ and itself. See man bash for more.

- Back Quote ( `` ) (a.k.a. Backtick)
  Executes the command it encloses (same as $ (a command) )

# Shell Arguments

- Passing arguments to our scripts is via positional parameters (a.k.a. command-line arguments).
- Are predefined buffers in the shell script.
- $ 1 through $ 9 ( read about the shift and xargs commands)
- During execution, the shell puts the first argument as $ 1, the second as $ 2 and so on.

Other Special parameters/variables:

- Name of the script ( $ 0)
- All parameters ( $ * and $ @)
- Number of arguments ( $ #)
- Exit status ( $ ?)

# Exit Status

- Commands return a value to the system when they terminate.
- The value (0-255) denotes success/failure of command's execution.
- The $ ? special variable stores the status of preceding command.
- Check out a command's man page for its exit status.

```
ls -l /bin
echo $?
```

(0 is success)

```
ls -l IExistNot
echo $?
```

(any other value is failure)

# Expression

- Sequence of operators and operands that reduces to a single value.

| | | |
|---|---|---|
| x=2 | $x + $y | $x < $y |
| y=4 | ($x * $y) / $x - $y | $x != $y |

- The expr command evaluates expressions. (requires space + escaping)
- Some operators:
  - Arithmetic operators
  - File operators
  - Comparison operators
  - Test operator
  - Logical operators

# Arithmetic Operators

- Addition, Subtraction (+, -)
- Multiplication, Division (*, /)
- Exponentiation (**)
- Modulus (%)
- Increment, Decrement (++, --).

- Short-hand assignments possible.
  +=  -=  *=  /=  %=
- Doing arithmetic → the ((...)) construct and the let shell built-in.

# The test operator

test expression or

[ expression ] [] is shorthand for test

- Performs a variety of checks.
- Returns exit status of 0 if expression is true; 1 otherwise.

- Read about double square bracket ( i.e. [[...]] )

# Integer Comparison

- -eq Equal to
- -ne Not equal to
- -gt Greater than
- -ge Greater than or equal to
- -lt Less than
- -le Less than or equal to

# String Comparison

| | |
|---|---|
| s1 == s2 | Equal to |
| s1 != s2 | Not equal to |
| -z str | str is zero/null string |
| -n str | str is non-zero/not null |

# Logical Operators

- expr1 AND expr2 → expr1 && expr2
- expr1 OR expr2 → expr1 || expr2
- NOT expr → !expr

# File Operators

- -e file — file exists?
- -r file — file exists and readable?
- -w file — file exists and writable?
- -x file — file exists and executable?
- -l file — file exists and a symbolic link?
- -f file — file exists and a regular file?
- -d file — file exists and a directory?
- file1 -nt file2 — file1 newer than file2?
- file1 -ot file2 — file1 older than file2?

# Control Flow

# The if Construct

- if-then-else

```
if [ <some test> ]
then
    <Do this thing>
else
    <Do that thing>
fi
```

# Example

```
#!/bin/bash
# Basic if statement

if [ $1 -gt 100 ]
then
        echo Hey that\'s a large number.
        pwd
fi


echo “outside”
```

# The if Construct

- if-then-elif-else

```
if <command1>
then
        <command set 1>
elif <command2>
then
        <command set 2>
...
else
        <command set N>
fi
```

# case...esac (multi-way selection)

```
case <expression> in
        pattern1) command1 ;;
        pattern2) command2 ;;
        pattern3) command3 ;;
        ...
esac
```

- case matches expression with pattern1 first.
- If matched, it executes command1. Otherwise, proceeds to pattern2 and so on.
- Pattern may be a regex (wildcards + EREs).

# The while loop

```
while <condition>
do
      <commands>
done
```

- Executes <commands> if exit status of <condition> is 0 i.e. successful.

# The until loop

```
until <condition>
do
      <commands>
done
```

- Executes <commands> as long as <condition> is non-zero i.e. fails.

# The for loop

```
for <variable> in <list>
do
      <commands>
done
```

- Every successive item in <list> is assigned to <variable> and <commands> executed.

# The for loop

- Specifying ranges in for loop.
  - {START..END..INCREMENT}
  - seq START INCREMENT END

- C-like flavor of for loop

```
for  (( i=1; i<=5; i++))
do
      echo $i
done
```

# Control Flow break

- To exit for, while and until loops prematurely.

```
while <condition>
do
        <action 1>
        <action 2>
        if <some check>
        then
                break                   (breaks out)
        fi
<action 3>
<action 4>
done
```

# Control Flow continue

- Skips to the next loop iteration.

```
for i in <some list>
do
        <command 1>
        <command 2>
        if <some check>
        then
                continue            (skips to next iteration)
        fi
<command 3>
done
```

# Explore Further

- Google is your best friend.

- Linux shell scripting tutorial http://www.freeos.com/guides/lsst/
- Advanced Bash Scripting http://tldp.org/LDP/abs/html/
- Unix shell scripting http://www.tutorialspoint.com/unix/