# CSCI 315: Data Structures
# Shell Scripting

Dr. Paul E. West

Department of Computer Science
Charleston Southern University

April 12, 2017

- Shell script is just like batch file in MS-DOS
- Useful to create our own commands
- Can save our lots of time
- Automate some of daily tasks

- Before Starting Linux Shell Script Programming you must know:
    - Kernel
    - Shell
    - Process
    - Redirectors, Pipes, Filters etc

# Kernel

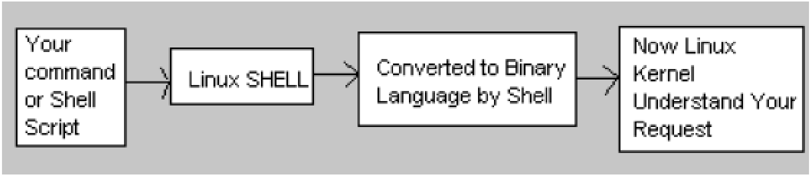
- Kernel is the heart of Linux OS
    - It manages resource of Linux OS
    - print data on printer
    - memory, file management
- Kernel decides who will use this resource, for how long and when.

# Linux Shell

- Computer understands the language of 0's and 1's called binary language
    - Difficult for us to read and write
- In the OS there is a special program called Shell
- Shell accepts your instruction or commands in English and translate it into computers native binary language

# Linux Shell

- This is what the shell does for us



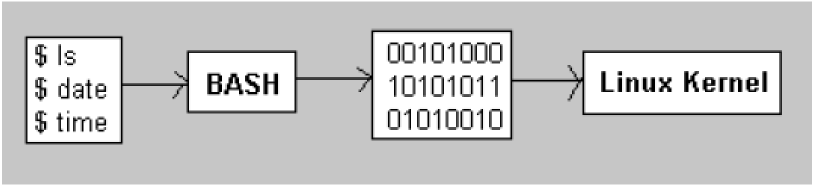
Your command or Shell Script

Linux SHELL

Converted to Binary Language by Shell

Now Linux Kernel Understand Your Request

- You type the command and shell converts it



$ ls
$ date
$ time

BASH

00101000
10101011
01010010

Linux Kernel

- Shell is a command language interpreter
- Popular shells
    - SH : Original shell
    - BSH : Bourne SHell
    - BASH : Bourne Again SHell (Ha!)
    - CSH : Similar to C programming language.
    - TCSH : Turbo C Shell
    - KSH : Korn SHell

- To find your shell type following command
  - echo $SHELL
- Known shells on your system
  - cat /etc/shells
- Your default shell is defined in /etc/passwd
  - How do you display the content of passwd?

# Process

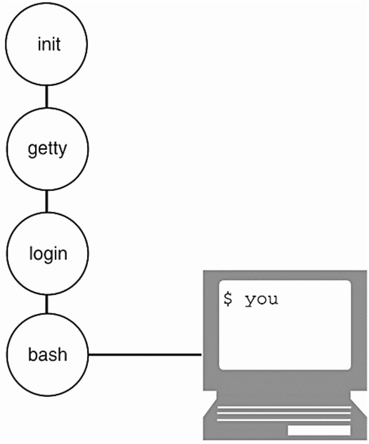
- Process is any kind of program or task carried out by your PC.
- A process is a program to perform some job.
- In Linux when you start process, it gets a number, called PID or process-id
- In Linux, the PID is in range 0 to 65535.

# Why Processes?

- Linux is a multi-user, multitasking OS
- You can run more than two processes simultaneously if you wish.
- An instance of running command is called a process
- Each process has a process-id (PID)

# Login Procedure

- The first process to run is called init, PID #1
- It spawns a getty process
- The /bin/login program is then executed
- After user inputs login/password, your shell (bash) is loaded



# Login Procedure

- The bash process looks for the system file, /etc/profile, and executes its commands
- It then looks in the user's home directory for an initialization file called .bash_profile
- Then it will execute a command for the user's ENV file, usually called .bashrc
- Finally the default prompt, dollar sign ($) (unless you have changed the default), appears on your screen and the shell waits for commands.

# The Environment

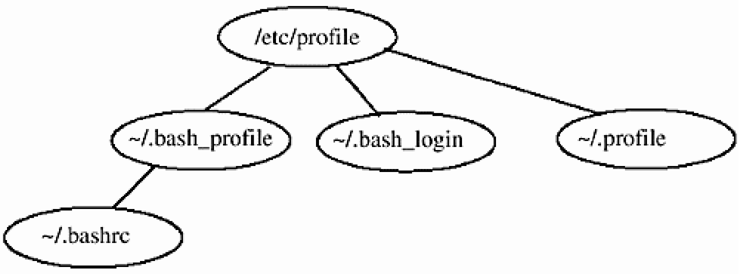
- The environment of a process consists of
  - variables
  - open files
  - working directory
  - functions
  - resource limits
  - ...
- The configuration for the user's shell is defined in the shell initialization files

# Initialization Files

- The bash shell has a number of startup files that are sourced
- Sourcing a file causes all settings in the file to become part of the current shell
- The initialization files are sourced depending on whether the shell is a login shell, an interactive shell, or a non-interactive shell (a shell script)
- Some files may be empty

# Initlization Files

- When you log on, before the shell prompt appears, /etc/profile is sourced
- It is a system wide initialization file
- Next, if it exists, the .bash_profile in the user's home directory is sourced



# Variables

- In Linux, there are two types of variables
  - System variables
    - Created and controlled by system
    - Defined in CAPITAL LETTERS
  - User defined variables (UDV)
    - Created and maintained by user.
    - Defined in lower case letters
- The capitalization isn't really enforced, just good practice

# System Variables

- To see system variables, type
  env
- BASH - Shell name
- BASH_VERSION - shell version name
- COLUMNS - No. of columns for screen
- HOME - home directory
- LINES - No. of lines for screen
- PS1 - Prompt setting 1
- PWD - Current working directory

# User Variables

- To define UDV use following syntax
    - Syntax: variablename=value
    - no=10
    - 10=no
    - To define variable called 'vech' having value Bus
        - vech=Bus
    - To define variable called n having value 10
        - n=10
- This is all very shell specific
- sh,csh, etc. have their own syntax

# Variable Naming Rules

- Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character (same as C++ variable names)
- Don’t put spaces on either side of the equal sign when assigning value to variable.
- no=10
- no =10
- no= 10

# Variable Naming Rules

- Variables are case-sensitive, just like filename in Linux.
    - no=10
    - No=11
    - NO=20
    - nO=2
    - Type echo $variablename to see the differences
- $ is used to get the value of a variable
    - No is just a word, $No is the value of the variable No

# Variable Naming Rules

- You can define NULL variable as follows
- vech=
- vech=""
- NULL is basically the empty string
- vech with value NULL is different from not having variable vech defined at all
- Type echo $vech to print it’s value
- Do not use ?,* etc, in your variable names.

# Arithmetic Operations

- Syntax: expr op1 operator op2
- op1 and op2 are any Integer Number
- operator
  - +, -, /, %(modular), * Multiplication
- expr 6 + 3 will work
- expr 6+3 will not work!!
  - Space between number and OP is required!!
- Why? Think about filenames

# Permissions

- Files have permissions in Unix
  - First 10 chars in ls -l output
- 3 Types of perms: Read, Write, eXecute
- 3 types affected: User, Group, Other
- chmod (change mode) used to change permissions
- Two ways of using chmod: absolute and relative

# chmod

- Relative: chmod changes files
  - chmod u+r files add read perms for users
  - chmod go-wx files remove write and execute perms from group and other (no error if group and other didn't have w and x before)
- Absolute: chmod 3-digit-code files
  - Consider read, write, execute as bits in binary number. 3 digits total for user, group, and other.
  - chmod 751 files
    7 = 4 + 2 + 1 r & w & x for user 5 = 4 + 1 which means r & x for group,
    1 = x for other
  - I prefer absolute, no question as to result
  - Either is acceptable for HW/exams

# How To Write a Shell Script

- The First Line
- The first column of the first line of the script will indicate the program that will be executing the code in the script. #!/bin/bash
- The #! is called a magic number and is used by the kernel to identify the program that should be interpreting the code in the script.
- The #! is pronounced shebang.
- Etymology:
  - ! has always been shorten to bang.
  - Probably came from SHarp Bang, haSH Bang, or SHEll Bang
- This line must be line 1 of your script.
- No spaces before #

## Write a Shell Script

-
  ```
  $ cat > first
  #!/bin/bash
  # My first shell script
  #
  clear
  echo "Knowledge is Power"
  ```
  Press Ctrl + D to finish typing and save.
- # indicates a comment, like // in C/C++

# Executing A Script

- Try first
- Didn’t work? Why? Path
- OK, now try
- ./first
- But that didn’t work, why? Permissions (90+% of problem in Linux.)
- Type
- chmod +x first
- ./first

# Qoutes

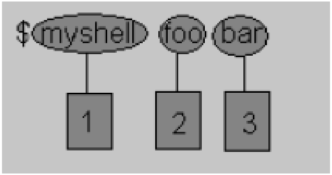
```
Double Quotes " — Anything enclosed in double quotes removed meaning of
                  those characters (except \ and $).
'Single quotes' — Enclosed in single quotes remains unchanged.
‘Back quote‘ — To execute command.
‘ allows commands within strings
Remember: $ is how you access the value of a variable!
```

# Practice

```
date="August 31, 1976"
echo "Today is date"
echo "Today is $date"
echo ‘Today is $date’
echo "Today is ‘date‘’
echo "expr 6 + 3"
echo ’expr 6 + 3’
```

# Command Line Arguments

**$ myshell foo bar**



1 Shell Script name i.e. myshell

2 First command line argument passed to myshell i.e. foo

3 Second command line argument passed to myshell i.e. bar

# Command Line Arguments

- In the shell, if we want to refer to the command line arguments
- myshell is $0
- f00 is $1
- bar is $2 (etc.)
- Number of arguments in a command line
- $# (useful for loops and error checks)
- All of command line arguments
- $* (useful for passing on to other commands)

# Practice

```
$ cat > demo
#!/bin/bash
#
# Script that demos, command line args
#
echo "Total number of command line argument are $#"
echo "$0 is script name"
echo "$1 is first argument"
echo "$2 is second argument"
echo "All of them are :- $*"
```

## Now Run

```
./demo Hello World
```

# Exit Status

- In Linux when a command is executed, it returns a value called the exit status
- return value is zero (0), command was successful,
- return value is nonzero (>0), command was not successful
- To determine this exit status we use the $? variable of shell (? For what just happened).
- (Windows has a similar concept called error levels)

# Exit Status Practice

- expr 1 + 3
- echo $?
- echo Welcome
- echo $?
- wildwest canwork?
- echo $?
- date
- echo $?
- echon $?
- echo $?

# The read Statement

- Use to get input from keyboard and store them to variable.
- Syntax: read varible1 varible2 varibleN
- Create the following script

  ```
  cat > sayH
  #!/bin/bash
  #Script to read your name from key-board
  #
  echo "Your first name please:"
  read fname
  echo "Hello $fname, Lets be friend!"
  ```

# Parameters for read

- read answer
    - Reads a line from standard input and assigns it to the variable answer.
- read first last
    - Reads a line from standard input to the first whitespace or newline, putting the first word typed into the variable first and the rest of the line into the variable last.
- read
    - Reads a line from standard input and assigns it to the built-in variable, REPLY.

# Parameters for read

- read -a arrayname
  - Reads a list of words into an array called arrayname (just FYI).
- read -e
  - Used in interactive shells with command line editing in effect; e.g., if editor is vi, vi commands can be used on the input line.
- read -p prompt
  - Prints a prompt, waits for input, and stores input in REPLY variable.
- read -r line
  - Allows the input to contain a backslash.

# read Example

```
#!/bin/bash
# Scriptname: nosy
echo -e "Are you happy?"
read answer
echo "$answer is the right response."
echo -e "What is your full name?"
read first middle last
echo "Hello $first"
echo -n "Where do you work?"
read
echo "I guess $REPLY keeps you busy!"
read -p "Enter your job title: "
echo "I thought you might be an $REPLY."
echo -n "Who are your best friends?  "
read -a friends
echo "Say hi to ${friends[2]}."
```

# If Then

- Syntax:
  if condition
  then
  command1 if condition is true or if exit status of condition is 0 (zero)
  ...
  ...
  fi

# Practice

-
  ```
  cat > showfile #!/bin/bash
  #
  #Script to print file
  #
  if cat $1
  then
  echo -e "File $1, found and successfully echoed"
  fi
  ./showfile foo
  ```

# test Command

- test command is used to see if an expression is true
  if it is true it return zero(0)
  returns nonzero(>0) for false.
  Syntax: test expression [OR expression ]

# Practice

-
  ```
  #!/bin/bash
  #
  # Script to see whether argument is positive
  #
  if test $1 -gt 0
  then
  echo "$1 is positive"
  fi
  ```

# Practice

- ispostive 5
- ispostive -45
- ispostive

# test Command

- For Mathematical comparisons use following operators in Shell Scripts
    - -eq : is equal to
    - -ne : not equal to
    - -lt : less than
    - -le : less than or equal to
    - -gt : greater than
    - -ge : greater than or equal to

# test Command

- string1 = string2
- string1 != string2
- string1
    - string1 is NOT NULL or not defined
- -n string1
    - string1 is NOT NULL and does exist
- -z string1
    - string1 is NULL and does exist

# test Command

- -s file : Non empty file
- -f file : Is File exist or normal file and not a directory
- -d dir : Is Directory exist and not a file
- -w file : Is writeable file
- -r file : Is read-only file
- -x file : Is file is executable

# Logical Operators

- ! expression
    - Logical NOT
- expression1 -a expression2
    - Logical AND
- expression1 -o expression2
    - Logical OR

# if else fi

```
if condition
then
command1 if condition is true or if exit status
of condition is 0(zero)
...
...
else
command2 if condition is false or if exit status
of condition is >0 (nonzero)
...
...
fi
```

# Practice

```
cat > isnump_n
#!/bin/bash
#
# Script to see whether argument is positive or negative
#
if test $# -ne 1
then
echo "$0 : You must supply one integer"
exit 1
fi
if test $1 -gt 0
then
echo "$1 is positive"
else
echo "$1 is negative"
fi
```

# Practice

- isnump_n 5
- isnump_n -45
- isnump_n
- isnump_n 0

# Multilevel if else fi

if condition
then
condition is zero (true - 0)
execute all commands up to elif statement
elif condition1
condition1 is zero (true - 0)
execute all commands up to elif statement
elif condition2
condition2 is zero (true - 0)
execute all commands up to elif statement
else
None of the above condtion,condtion1,condtion2 are true (i.e. all of the above nonzero or false)
execute all commands up to fi
fi

# Multilevel if else fi

```
$ cat > elf
#!/bin/bash
#
# Script to test if..elif...else
#
if [ $1 -gt 0 ] # note space around [ and ]
then
echo "$1 is positive"
elif [ $1 -lt 0 ]
then
echo "$1 is negative"
elif [ $1 -eq 0 ]
then
echo "$1 is zero"
else
echo "Oops! $1 is not number, give number"
```

# Practice

- ./elf 1
- ./elf -2
- ./elf 0
- ./elf a

# For loop

- More like a for-each loop
- Syntax:
  for { variable name } in { list }
  do
  execute one for each item in the list until the list is not finished (And repeat all statement between do and done)
  done

## Practice

```
cat > testfor
for i in 1 2 3 4 5
do
echo "Welcome $i times"
done
```

- More useful then you think for file management
- Variable could be part of a filename or a command (e.g. hw$i.ys)

```
cat > mtable
#!/bin/bash
#
#Script to test for loop
##
if [ $# -eq 0 ]
then
echo "Error - Number missing form command line argument"
echo "Syntax : $0 number"
echo "Use to print multiplication table for given number"
exit 1
fi
n=$1
for i in 1 2 3 4 5 6 7 8 9 10
do
echo "$n * $i = `expr $i \* $n`"
done
```

# Practice

- ./mtable 7
- ./mtable

## C-style Loop

```
for (( c=1; c<=5; c++ )); do
        loop body
   done
can use  { } instead of do done on modern Bash, but undocumented
More on (( )) in a few slides
FYI: break and continue are supported by all Bash loops
```

# While Loop

- Syntax:
  while [ condition ]
  do
  command1
  command2
  command3
  ..
  ....
  done
  There is also an until loop that is the same as while not.

```
cat > nt1
#!/bin/bash
#
#Script to test while statement
##
if [ $# −eq 0 ]
then
echo "Error − Number missing form command line argument"
echo "Syntax : $0 number"
echo " Use to print multiplication table for given number"
exit 1
fi
n=$1
i=1
while [ $i −le 10 ]
do
echo "$n ∗ $i = ‘expr $i \∗ $n‘"
i=‘expr $i + 1‘
done
```

- Execute the following ./nt1 7

# case statement

- Syntax:

```
case $variable–name in
pattern1) command
...
..
command;;
pattern2) command
...
..
command;;
patternN) command
...
..
command;;
*) command
...
...
command;;
esac
```

# Practice

```
cat > car
#!/bin/bash
# if no vehicle name is given
# i.e. -z $1 is defined and it is NULL
#
# if no command line arg
if [ -z $1 ]
then
rental="*** Unknown vehicle ***"
elif [ -n $1 ]
then
# otherwise make first arg as rental
rental=$1
fi
case $rental in
"car") echo "For $rental Rs.20 per k/m";;
"van") echo "For $rental Rs.10 per k/m";;
"jeep") echo "For $rental Rs.5 per k/m";;
"bicycle") echo "For $rental 20 paisa per k/m";;
*) echo "Sorry, I can not get a $rental for you";;
esac
```

# Practice

- car van
- car car
- car Maruti-800

# let using (( ))

- Though the command is let, the usual use is with (( )). The goal is to simplify some basic math and variable usage.
- while [ $a -lt $LIMIT ] #spacing is required
- becomes
- while (( a <= LIMIT )) #spacing is optional
- -o becomes || and aa becomes &&
- a=

  ```
  ‘expr \$a + 1‘
  ```

  becomes ((a += 1))
- z = $(( a + b )) is the more general form

# declare

- With declare variables can be given a type:
- declare -i n
- n=6/3 # n will now be 2, no expr or let

# seq

- seq is an older command, but useful:
- seq [OPTION]... LAST
- seq [OPTION]... FIRST LAST
- seq [OPTION]... FIRST INCREMENT LAST
- FIRST and INCREMENT default to 1, options for formatting and padding

```
for i in `seq 1 10`; do
    loop body
done
```

# {} expansion

- or i in {1..10..2} # 1 to 10 increment by 2
- No spaces, no variables allowed
- Unless you use eval, but that is weird
- {f..k} becomes f g h i j k
- 1.{0..9} becomes 1.0 1.1 1.2 ... 1.9
- {A..Z}{0..9} generates: A0 A1 ... A9 B0 B1 .. Z9
- {{A..Z},{a..z}} generates: A B .. Z a b .. Z
- Can have something before or after { }

- There is a lot.
- I do not expect you to know everything on a test.
- It is best to understand how much power shell scripting is giving you:
  - program chaining/redirection
  - program argument manipulation
  - can interact with anything
- When you don't know something, Google!