Linux Essentials

Linux Origins

- 1984: The GNU Project and the Free Software Foundation
  - Creates open source version of UNIX utilities
  - Creates the General Public License (GPL)
  - Software license enforcing open source principles
- 1991: Linus Torvalds
  - Creates open source, UNIX-like kernel, released under the GPL

Linux Distributions

- Linux Distributions:
  - Redhat
  - Suse
  - Mandrake
  - Caldera
  - Ubuntu
  - Debian etc
- Shell is an interface between user and OS
- Kernel is the core of OS. It receives requests called system calls from programs and initiates processes that carryout these requests

Logging in to a Linux System

- Two types of login screens: virtual consoles (text-based) and graphical logins (called display managers)
- Login using login name and password
- Each user has a home directory for personal file storage
- Everything is a file (including hardware)

Switching between virtual consoles

- A typical Linux system will run six virtual consoles and one graphical console
- Switch among virtual consoles by typing: **Ctrl-Alt-F*[1-6]***
- Access the graphical console by typing **Ctrl-Alt-F7**

The root **user**

- The root user: a special administrative account
- Also called the superuser
- Do not login as root unless necessary
- **sudo *command*** runs ***command*** as root

Running Commands

- Commands have the following syntax:

  command *options arguments*
- Each item is separated by a space
- Multiple commands can be separated by ;
  - **date** - display date and time
  - **cal** - display calendar

Getting Help

- Don't try to memorize everything!
- Many levels of help
  - **whatis**
  - ***command* --help**
  - **man** and **info**

Help Manuals

- Displays short descriptions of commands
- Often not available immediately after install

```
$ whatis cal
cal (1) - displays a calendar
```

- **The --help Option**
  - Displays usage summary and argument list
  - Used by most, but not all, commands
  - **date --help**

The man Command

- Provides documentation for commands
- Every command has a man "page“
- While viewing a man page
  - Navigate with arrows, **PgUp**, **PgDn**
  - ***/text*** searches for text
  - **n**/**N** goes to next/previous match
  - **q** quits
- Pages are grouped into "chapters"
  - Collectively referred to as the Linux Manual
  - **man [<chapter>] <command>**
  - **The info Command** Similar to **man**, but often more in-depth
  - **info [*command*]**

Browsing the Filesystem

- Files and directories are organized into a single-rooted inverted tree structure
- Filesystem begins at the root directory, represented by / (forward slash)
- Names are case-sensitive
- Paths are delimited by /

Some Important Directories

- **Home Directories:** /root,/home/*username*
- **User Executables:** /bin, /usr/bin, /usr/local/bin
- **System Executables:** /sbin, /usr/sbin, /usr/local/sbin
- **Other Mountpoints:** /media, /mnt
- **Configuration:** /etc
- **Temporary Files:** /tmp
- **Kernels and Bootloader:** /boot
- **Server Data:** /var, /srv
- **System Information:** /proc, /sys
- **Shared Libraries:** /lib, /usr/lib, /usr/local/lib

- **pwd -** Displays the current working directory
- **touch** - create empty files
- **mkdir** creates directories
- **rmdir** removes empty directories
- **rm -r** recursively removes directory trees
- Names are case-sensitive
    - MAIL, Mail, mail, and mAiL

Absolute and Relative Pathnames

- Absolute pathnames begin with a forward slash /
- Relative pathnames do not begin with a slash

Changing Directories

- **cd** changes directories
- To an absolute or relative path:
  - **cd** /home/joshua/work
  - **cd** project/docs
- To a directory one level up:
  - **cd** ..
- To your home directory:
  - **cd**
- To your previous working directory:
  - **cd -**

Copying & moving Files and Directories

- **cp** - copy files and directories
- **mv** - move and/or rename files and directories
- Usage:

  cp/mv [options] ***file destination***
- More than one file may be copied at a time if the destination is a directory:

  cp/mv [options] ***file1 file2 dest***

Users, Groups and Permissions

- Every user is assigned a unique User ID number (UID)
  - UID 0 identifies root
- Users' names and UIDs are stored in /etc/passwd
- Users are assigned a home directory and a program that is run when they log in
- Users cannot read, write or execute each others' files without permission
- Every user in a Linux system belongs at least to one group
- Users are assigned to groups
- Each group is assigned a unique Group ID number (gid)
- GIDs are stored in /etc/group

Permission Types

- Four symbols are used when displaying permissions:
- r: permission to read a file or list a directory's contents
- w: permission to write to a file or create and remove files from a directory
- x: permission to execute a program or change into a directory and do a long listing of the directory
- -: no permission (in place of the r, w, or x)
- File permissions may be viewed using **ls -l**

Changing Permissions - Symbolic Method

- To change access modes:
  - **chmod [-R]** ***mode file***
- Where *mode* is:
  - **u**,**g** or **o** for user, group and other
  - **+** or **-** for grant or deny
  - **r**, **w** or **x** for read, write and execute
- Examples:
  - **ugo+r**: Grant read access to all
  - **o-wx**: Deny write and execute to others

Changing Permissions - Numeric Method

- Uses a three-digit mode number
  - first digit specifies owner's permissions
  - second digit specifies group permissions
  - third digit represents others' permissions
- Permissions are calculated by adding:
  - 4 (for read)
  - 2 (for write)
  - 1 (for execute)
- Example:
  - **chmod 640 myfile**

The Seven Fundamental Filetypes

ls -l symbol	File Type
-	regular file
d	directory
l	symbolic link
b	block special file
c	character special file
p	named pipe
s	socket

Vi - Text Editor

- Use the three primary modes of **vi** and **vim**
- Navigate text and enter Insert mode
- Change, delete, yank, and put text
- Undo changes
- Search a document
- Save and exit

Vi modes

- Three main modes:
- Command Mode (default): Move cursor, cut/paste text, change mode
- Insert Mode: Modify text
- Ex Mode: Save, quit, etc
- **Esc** exits current mode

Modifying a File : Insert Mode

- **i** begins insert mode at the cursor
- Many other options exist
  - **A** append to end of line
  - **I** insert at beginning of line
  - **o** insert new a line (below)
  - **O** insert new line (above)
  - **k** move cursor up
  - **j** move cursor down
  - **h** move cursor left
  - **l** move cursor right
  - **4yy** yanks current line and 3 lines below
  - **dd** is used to delete
  - **P** or **p** is used to paste
  - **cc** is used to cut

Saving a File and Exiting vim
Ex Mode

- Enter Ex Mode with :
  - Creates a command prompt at bottom-left of screen
- Common write/quit commands:
  - **:w** writes (saves) the file to disk
  - **:wq** writes and quits
  - **:q!** quits, even if changes are lost

Undo & Search

- **u** undo most recent change
- **U** undo all changes to the current line since the cursor landed on the line
- **Ctrl-r** redo last "undone" change
- Searching for a pattern (/pat and ?pat)
- Search and replace :%s/e/3/g

Pattern matching – The WILD CARDS

***** - matches zero or more characters

? - matches any single character

[0-9] - matches a range of numbers

[abc] - matches any of the character in the list

[^abc] - matches all except the characters in the list

REDIRECTION

- Linux provides three I/O channels to programs
  - Standard input (STDIN) - keyboard by default
    - < Redirect STDIN to file
  - Standard output (STDOUT) - terminal window by default
    - > Redirect STDOUT to file
  - Standard error (STDERR) - terminal window by default
    - 2> Redirect STDERR to file

- File contents are overwritten by default. >> appends.

Example

- $ ls > myfiles
- $ date >> myfiles
- $ cat < myfiles
- $vi sname
  - Vivek
    ashish
    zebra
    babu

- $ sort < sname > sorted_names
- $ cat sorted_names
- $ tr "[a-z]" "[A-Z]" < sname > cap_names
- $ cat cap_names

Alias

- Aliases let you create shortcuts to commands

```
# alias c='clear'
# alias dir='ls -l'
```

head and tail commands

- **head**: Display the first 10 lines of a file
  - Use **-n** to change number of lines displayed
- **tail**: Display the last 10 lines of a file
  - Use **-n** to change number of lines displayed

Compressing and Archiving files

- gzip and gunzip (.gz)
- bzip2 and bunzip2(.bz2)
- **tar** - standard Linux archiving command

Locate

- Queries a pre-built database of paths to files on the system
  - Database must be updated by administrator
  - Full path is searched, not just filename
  - Locate only finds files by looking up the name in a database.

```
# locate stdio.h
# locate smb.conf
```

find

- Locate uses an internal database to look up indexed files. This database needs to be updated using the command 'updatedb' at regular intervals, so that you'll get more accurate results. Find just does a "normal" search, ie. it just goes through the given path(s) just when you want it to.
- Find command is used to search for files
- Find / -name a.out
- Find . –name “*.txt”

Grep (global regular expression print) : searching for a pattern

- Grep scans its input in a pattern and displays the lines containing the pattern.
  - ^ represents beginning of line
  - $ represents end of line
  - **“abc$” “^abc”**
- # grep “sales” emp.lst display lines containing the string sales from the file emp.lst
- # grep ‘john’ /etc/passwd
- # grep ‘^j’ /etc/passwd
- #grep root /etc/passwd | cut -d: -f7

Pipes

- A pipe is a way to connect the output of one program to the input of another program without any temporary file

Command using Pipes	Meaning or Use of Pipes
$ ls | more	Output of ls command is given as input to more command So that output is printed one screen full page at a time.
$ who | sort	Output of who command is given as input to sort command So that it will print sorted list of users
$ who | sort > user_list	Same as above except output of sort is send to (redirected) user_list file
$ who | grep raju	Output of who command is given as input to grep command So that it will print if particular user name if he is logon or nothing is printed (To see particular user is logon or not)

The Shell

- Shell is an interface between user and OS
  - A command (entered at the shell prompt ($) or from a script) is interpreted and if valid is sent to the kernel for execution
- Shell categories
  - Bourne shell and derivatives
    - The Bourne shell, or sh was the default Unix Shell, by Stephen Bourne, of AT&T Bell Labs
    - Pascal like syntax
    - Korn shell (ksh) by David Korn in between sh, csh
    - *Bourne-Again shell (bash) by GNU*, takes features from sh, csh and ksh
  - C shell
    - by Bill Joy, at the University of California, Berkeley
    - C shell (csh) used C like syntax for job control
    - tcsh by Greer (t for Tenex OS)

Different shells

Shell Name	Developed by	Where	Remark
BASH (Bourne-Again SHell)	Brian Fox and Chet Ramey	Free Software Foundation	Default Linux shell
CSH (C SHell)	Bill Joy	University of California	The C shell's syntax and usage are very similar to the C programming language.
KSH (Korn SHell)	David Korn	AT & T Bell Labs	--
TCSH	TCSH is an enhanced but completely compatible version of the UNIX C shell (CSH).		

Shell commands

- To find all available shells in your system
  - $ **cat /etc/shells**
- To find your current shell type following command
  - $ **echo $SHELL**
- To change to another shell
  - $ **/bin/sh**
  - $ **/bin/bash**
  - $ **/bin/csh**
  - $ **/bin/ksh**

What is shell script

- Shell scripts are text files that contain a series of commands or statements to be executed
- Normally shells are interactive
- It means shell accept command from you one by one and execute them.
- you can store this sequence of command to text file and tell the shell to execute this text file instead of entering the commands

Creating Shell Scripts

- Step 1: Use any editor like vi to write shell script with an extension .sh
- First line contains the magic
  - shebang sequence: #!
    - **#!/bin/bash**
- Comment your scripts!
  - Comments start with a #
- Step 2: Make the script executable:
  - chmod +x first.sh
- Step 3: Execute your script as
  - ./first.sh

Example

- $ vi first.sh

```
#!/bin/bash
#
# My first shell script
#
clear
echo "Knowledge is Power"
```

Example

```
$ vi ginfo
  #
  # Script to print user information who currently
  #login , current date & time
  #
  clear
  echo "Hello $USER"
  echo "Today is : " ; date
  echo "Number of user login : " ; who | wc -l
  echo "Calendar"
  cal
  exit 0
```

Variables in shell

- **System variables (Environment)**
  - *Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.*
  - ***env** will display a list of environmental variables*
  - *Do not modify System variable this can some time create problems.*
- **User defined variables (UDV)**
  - *Created and maintained by user. This type of variable defined in lower case letters*

System variables

System Variable	Meaning
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
COLUMNS=80	No. of columns for our screen
HOME=/home/sharan	Our home directory
LINES=25	No. of columns for our screen
LOGNAME=students	students Our logging name
OSTYPE=Linux	Our OS type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PS1=[\u@\h \W]\$	Our prompt settings
PWD=/home/students/Common	Our current working directory
SHELL=/bin/bash	Our shell name
USERNAME=sharan	User name who is currently login

UDV

- To define UDV use following syntax
  Syntax: variable name=value
- *'**value**' is assigned to given '**variable name**' and value must be on right side = sign.*
- *Don't put spaces on either side of the equal sign when assigning value to variable*
- *Example:*

```
$ no=10        # this is ok
$ 10=no        # Error, NOT Ok, Value must be on
               right side of = sign.
```

Exercise

Q.1 How to Define variable x with value 10 and print it on screen.

```
$ x=10
 $ echo $x
```

Q.2 How to Define variable xn with value Rani and print it on screen

```
$ xn=Rani
$ echo $xn
```

Q.3 How to print sum of two numbers, let's say 6 and 3?

```
$ expr 6 + 3
```

Exercise

Q.4 How to define two variable x=20, y=5 and then to print division of x and y (i.e. x/y)

```
$x=20
$ y=5
$ expr x / y
```

Q.5 Modify above and store division of x and y to variable called z

```
$ x=20
$ y=5
$ z=`expr x / y`
$ echo $z
```

Point out error if any in following script

-
  ```
  $ vi variscript
  #
  # Script to test MY knowledge about variables!
  #
  myname=Sharan
  myos = TroubleOS
  myno=5
  echo "My name is $myname"
  echo "My os is $myos"
  echo "My number is myno, can you see this
  number"
  ```

Shell arithmetic

- *Syntax:*
  expr op1 math-operator op2

  Examples:
  ```
  $ expr 1 + 3
  $ expr 2 - 1
  $ expr 10 / 2
  $ expr 20 % 3
  $ expr 10 \* 3
  $ echo `expr 6 + 3`
  ```

- **$ echo "expr 6 + 3"** # It will print expr 6 + 3
  $ echo 'expr 6 + 3' # It will print expr 6 + 3

- **$ echo "Today is date"**
  Can't print message with today's date.
  $ echo "Today is `date` “   #It will print today's date

Exit Status

- If return *value is zero* (0), command is successful
- If return *value is nonzero*, command is not successful or some sort of error executing command/shell script.
- To determine exit Status you can use **$?** special variable of shell.
- Eg: **$ rm unknownfile**
  cannot remove `unknownfile': No such file or directory
  $ echo $?
  it will print nonzero value to indicate error.
- **$ ls**
  $ echo $?
  It will print 0 to indicate command is successful.

The read Statement

- Use to get input (data from user) from keyboard and store (data) to variable
- $ vi sayH

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

Wild cards (Meta Characters)

Wild card /Shorthand	Meaning	Examples	
*	Matches any string or group of characters.	$ ls *	will show all files
		$ ls a*	will show all files whose first name is starting with letter 'a'
		$ ls *.c	will show all files having extension .c
		$ ls ut*.c	will show all files having extension .c but file name must begin with 'ut'.
?	Matches any single character.	$ ls ?	will show all files whose names are 1 character long
		$ ls fo?	will show all files whose names are 3 character long and file name begin with fo
[...]	Matches any one of the enclosed characters	$ ls [abc]*	will show all files beginning with letters a,b,c

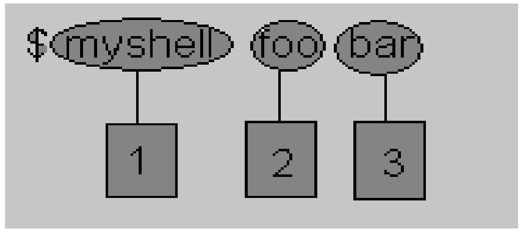
Command Line Processing

- $ ls foo
  $ cp y y.bak
  $ mv y.bak y.okay
  $ tail -10 myf
  $ mail raj
  $ sort -r -n myf
  $ date
  $ clear

Command	No. of argument to this command (i.e $#)	Actual Argument
ls	1	foo
cp	2	y and y.bak
mv	2	y.bak and y.okay
tail	2	-10 and myf
mail	1	raj
sort	3	-r, -n, and myf
date	0	
clear	0	

- Lets take ls command **$ ls -a /***
  This command has 2 command line argument -a and /* is another.

- **$ myshell foo bar**



- $\boxed{1}$ Shell Script name i.e. myshell

  $\boxed{2}$ – First command line argument passed to myshell i.e. foo

  $\boxed{3}$ – Second command line argument passed to myshell i.e. bar

- In shell if we wish to refer this command line argument we refer above as follows
  $\boxed{1}$ myshell it is $0 $\boxed{2}$ foo it is $1

  $\boxed{3}$ bar it is $2
- Here **$#** (built in shell variable) will be 2 (Since foo and bar only two Arguments), Please note at a time such 9 arguments can be used from $1..$9, You can also refer all of them by using $* (which expand to `$1,$2...$9`). Note that $1..$9 i.e command line arguments to shell script is known as "*positional parameters*".

Exercise

- Try to write following for commands
  Shell Script Name ($0),
  No. of Arguments (i.e. $#),
  And actual argument (i.e. $1,$2 etc)
  $ sum 11 20
  $ math 4 - 7
  $ d
  $ bp -5 myf +20
  $ Ls *
  $ cal
  $ findBS 4 8 24 BIG

Shell Script Name	No. Of Arguments to script	Actual Argument ($1,..$9)				
$0	*$#*	*$1*	*$2*	*$3*	*$4*	*$5*
sum	2	11	20			
math	3	4	-	7		
d	0					
bp	3	-5	myf	+20		
Ls	1	*				
cal	0					
findBS	4	4	8	24	BIG	

Linux Variables

- $? = Exit status variable.
- $0 = The name of the command you used to call a program.
- $1 = The first argument on the command line.
- $2 = The second argument on the command line.
- $n = The nth argument on the command line.
- $* = All the arguments on the command line.
- $# The number of command line arguments.

Decision making and loops : Introduction

Expression	Meaning to us	Your Answer	BC's Response
5 > 12	Is 5 greater than 12	NO	0
5 == 10	Is 5 is equal to 10	NO	0
5 != 2	Is 5 is NOT equal to 2	YES	1
5 == 5	Is 5 is equal to 5	YES	1
1 < 2	Is 1 is less than 2	Yes	1

In Linux Shell Value 0	Meaning	Example
Zero Value (0)	Yes/True	0
NON-ZERO Value	No/False	-1, 32, 55 anything but not zero

Decision making and loops : Introduction

- It means when ever there is any type of comparison in Linux Shell It gives only two answer one is YES and NO is other.

In Linux Shell Value	Meaning	Example
Zero Value (0)	Yes/True	0
NON-ZERO Value	No/False	-1, 32, 55 anything but not zero

- Remember both bc and Linux Shell uses *different ways to show True/False values*

Value	Shown in bc as	Shown in Linux Shell as
True/Yes	1	0
False/No	0	Non - zero value

if condition

- if condition which is used for decision making in shell script, If given condition is true then command1 is executed.
  Syntax:

```
    if condition
    then
            command1 if condition is true or if exit
            status of condition is 0 (zero)
            … …
    fi
```

test command or [expr]

- test command or [expr] is used to see if an expression is true, and if it is true it return zero(0), otherwise returns nonzero for false.
  Syntax:
  test expression OR [expression]
- $ vi ispostive

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

Mathematical operators in shell script

Mathematical Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements	But in Shell	
			For test statement with if command	For [expr] statement with if command
-eq	is equal to	5 == 6	if test 5 -eq 6	if [5 -eq 6]
-ne	is not equal to	5 != 6	if test 5 -ne 6	if [5 -ne 6]
-lt	is less than	5 < 6	if test 5 -lt 6	if [5 -lt 6]
-le	is less than or equal to	5 <= 6	if test 5 -le 6	if [5 -le 6]
-gt	is greater than	5 > 6	if test 5 -gt 6	if [5 -gt 6]
-ge	is greater than or equal to	5 >= 6	if test 5 -ge 6	if [5 -ge 6]

Logical Operator

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

if...else...fi

- If given condition is true then command1 is executed otherwise command2 is executed.
- *Syntax:*

```
if condition
then
      condition is zero (true - 0) execute all   commands
up to else statement
else
      if condition is not true then execute all
      commands up to fi
fi
```

Nested if-else-fi

- You can write the entire if-else construct within either the body of the if statement or the body of an else statement. This is called the nesting of ifs.
- ***Syntax:***

```
if condition
then
    if condition
    then
            ..... .. do this
    else
            .... .. do this
    fi
else
    ... ..... do this
fi
```

Multilevel if-then-else

- ***Syntax**:*

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

Loops in Shell Scripts

- *Computer can repeat particular instruction again and again, until particular condition satisfies. A group of instruction that is executed repeatedly is called a loop.*
- *Bash supports:*
  - *for loop*
  - *while loop*

for 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

for Loop

- *Syntax:*

  for ((expr1; expr2; expr3))
  do

  ....

  repeat all statements between do and done until expr2 is TRUE

  done

✓ In above syntax BEFORE the first iteration, ***expr1*** is evaluated. This is usually used to initialize variables for the loop. All the statements between do and done is executed repeatedly UNTIL the value of ***expr2*** is TRUE. AFTER each iteration of the loop, ***expr3*** is evaluated. This is usually use to increment a loop counter.

Nested for

- $ vi nestedfor.sh

```
for (( i = 1; i <= 5; i++ ))      ### Outer for loop ###
do

      for (( j = 1 ; j <= 5; j++ )) ### Inner for loop ###
      do
                echo -n "$i "
      done

  echo "" #### print the new line ###

done
```

while loop

- *Syntax:*

```
while [ condition ]
do
    command1
    command2
    command3 .. ....
done
```

The case Statement

- *case $variable-name in*

```
pattern1)  command
           … ..
           command;;
pattern2)  command
           … ..
           command;;
patternN)  command
           … ..
           command;;
*)         command
           … ..
           command;;
esac
```

The *$variable-name* is compared against the patterns until a match is found. The shell then executes all the statements up to the two semicolons that are next to each other. The default is *) and its executed if no match is found