Reliance Tech Services
Anil Dhirubhai Ambani Group

UNIX – Basic Commands

Shell Programming

- A *shell script is a text file that contains one or more commands*
- In a shell script, the shell assumes each line of the text file holds a separate command.
- Some of the other scripting languages are:
  - Perl(Practical Extraction and Reporting Language),
  - pythol,
  - Tcl(Tool Command Language)
  - MS-DOS Batch File

- Script:

```
echo “Good Morning!”
echo “Enter your name?”
read name
echo “HELLO $name How are you?”
```

- To Execute it
  - sh scipt_name
- To Debug it
  - sh –x script_name

- Script:

```
echo “Enter first Number”
read no1
echo “Enter second Number”
read no2
res=`expr $no1 + $no2`
echo “The result is $res”
```

- In above script res=`expr $no1 + $no2` can be replaced by let res=no1+no2
- Add *one* to variable i. Using let statement:
  - let i=i+1 (If no spaces in expression)
  - let "i = i + 1" (enclose expression in “… “ if expression includes spaces)
  - ((i = i + 1))

- Example:Examine th difference between the two codes.

```
var=pwd
echo $var
```

- Lets see this:

```
var=`pwd`
echo $var
```

- Since pwd is enclosed in backquotes it will get replaced by present working directory. echo will display name of current working directory.

Command Line Arguments

- You can pass values to shell programs while you execute shell scripts. These values entered through command line are called as *command line arguments*.
- Arguments are assigned to special variables $1, $2 etc called as positional parameters.
- *special parameters*
  - $0 – Gives the name of the executed command
  - $* - Gives the complete set of positional parameters
  - $# - Gives the number of arguments
  - $$ - Gives the PID of the current shell
  - $! – Gives the PID of the last background job
  - $? – Gives the exit status of the last command
  - $@ - Similar to $*, but generally used with strings in looping constructs

- Example:

```
echo Program: $0

echo Number of arguments are $#

echo arguments are $*
```

- **Conditional Execution using && and ||**
  - The shell provides && and || operators to control the execution of a command depending on the success or failure of previous command.
  - && ensures the second command executes only if the first has succeeded.
  - || will ensure that the second command is executed only if the first has failed.
    - **$grep `director` emp.lst && echo "pattern found"**
    - **$grep `manager` emp.lst || echo "pattern not found"**

- Syntax:

```
if <condition is true>
 then
  <execute commands>
else
 <execute commands>
fi
```

- Example:

```
if  grep “director” emp.lst
then
    echo “Pattern Found”
else
    echo “Pattern Not Found”
    fi
```

- Specify condition either using *test* or *[condition]*
  - Example: test $1 -eq $2 same as [$1 -eq $2]
- Relational Operator for Numbers:
  - 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

- **Example:**

```
if  test $# -lt 1
then
    echo “Please key in the arguments”
else
    if [ $1 -gt  0 ]
    then
        echo “Number is positive”
    elif  test $1 -lt 0; then
        echo “Number is negative”
    else
        echo “Zero”
    fi
    fi
```

Relational Operator for strings and logical operators

- String operators used by *test*:
  - -n str True, if str not a null string
  - -z str True, if str is a null string
  - S1 = S2 True, if S1 = S2
  - S1 != S2 True, if S1 ≠ S2
  - str True, if str is assigned and not null

- Logical Operators
  - -a .AND.
  - -o .OR.
  - ! Not

11

File related operators

- File related operators used by test command
  - -f <file> True, if file exists and it is regular file
  - -d<file> True, if file exist and it is directory file
  - -r <file> True, if file exist and it is readable file
  - -w <file> True, if file exist and it is writable file
  - -x <file> True, if file exist and it is executable file
  - -s <file> True, if file exist and it's size > 0
  - -e <file> True, if file exist

- Example:

```
echo "Enter File Name:\c "
read source
if [ -z “$source” ] ;then
   echo “You have not entered file name“
else
     if test –s “$source”;then   #file exists & size is > 0
          if test ! –r  “$source” ;then
               echo “Source file is not readable”
          exit
     fi ;else
          echo “Source file not present”
          exit
     fi
fi
```

- Example:

```
echo "Enter year"
read year
if (( (year % 400) == 0 ))
then
   echo"$year is a leap year!"
elif (( (year % 4) == 0 ))
then
   if (( (year % 100) != 0 ))
then
   echo "$year is a leap year!"
else
   echo "$year is not a leap year."
fi
else
   echo "$year is not a leap year."
fi
```

- Select Case

```
case <expression> in
   <pattern 1> ) <execute commands> ;;
   <pattern 2> ) <execute commands> ;;
      <…>
esac
```

- Example:

```
echo “\n Enter Option : \c”
   read choice
   case $choice in
   1) ls -l ;;
   2) ps -f ;;
   3) date ;;
   4) who ;;
   esac
```

- Example:

```
case `date | cut -d“  “ –f1`  in
   Mon ) <commands> ;;
   Tue ) <commands> ;;
   :
esac
```

- Example:

```
echo “do you wish to continue?”
read ans
case “$ans” in
     [yY] [eE] [sS]) ;;
     [nN] [oO]) exit ;;
       *) “invalid option” ;;
esac
```

- Example:

```
echo; echo "Hit a key, then hit return."
 read Keypress
 case "$Keypress" in
      [a-z] ) echo "Lowercase letter";;
       [A-Z] ) echo "Uppercase letter";;
       [0-9] ) echo "Digit";;
       * ) echo "Punctuation, whitespace, or other";;
esac
```

- While:

```
while <condition is true>
  do
    <execute statements>
  done
```

- Example:

```
num=1
while [  $num –le 10  ]
do
   echo $num
   num=`expr  $num + 1`
done
#end of script
```

- Example:

```
while read name
do
  if  [ -z $name ]
  then
    exit;
  fi
    echo $name | tr "[a-z]" "[A-Z]"
done
```

- Continue:
  - Suspends statement execution following it.
  - Switches control to the top of loop for the next iteration.
- Break:
  - Causes control to break out
- Exit:
  - Exit - used for premature termination of program
  - By default exit returns 0 which is assign to $?
  - You can specify exit with return value
    - ❖**e.g exit 2**

Table 5-1. Exit Status

Value	Description
0	Success.
1	A built-in command failure.
2	A syntax error has occurred.
3	Signal received that is not trapped (see the `trap` command).

- For Loop

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

- Example:

```
for x in 1  2  3
 do
   echo “The value of x is $x”
done
```

- Example:

```
for ((  i = 0 ;  i <= 5;  i++  ))
  do
    echo "Welcome $i times"
  done
```

- Example:

```
for x in `cat first`
  do
    echo $x
  done
```

- Example:

```
for x in  *
  do
    echo $x
  done
```

- Example:

```
for x in  $*
do
   grep “$x” emp.lst || echo “Pattern Not Found”
done
```

- Example:

```
for x in  $@
do
   grep “$x” emp.lst || echo “Pattern Not Found”
done
```

- Example:

```
for x in  *.txt
do
        leftname=`basename $x txt`
        mv $x ${leftname} doc
done
```

- The command **basename** extracts the base file name from the pathname;
  - **Example:** basename /home/trainer/dir1/file1.sh

    File1
  - **Example :** basename file1.sh sh

    File1.
- **When basename is used with a 2nd argument it strips off the string from the 1st argument**

- Example:

```
command=”init”
until [ “$command” = “exit” ]
do
   echo -n “Enter command or \”exit\” to quit: “
   read command
   case $command in
      ls) echo “Command is ls.”;;
      who)echo “Command is who.”;;
      *)if [ $command != “exit” ]
         then
            echo “Why did you enter $command?”
            fi
            ;;
   esac
done
```

- Example:

```
command=”init” # Initialization.
while [ “$command” != “exit” ]
do
   echo -n “Enter command or \”exit\” to quit: “
   read command
   case $command in
      ls)echo “Command is ls.”;;
      who)echo “Command is who.”;;
      *)if [ $command != “exit” ]
         then
               echo “Why did you enter $command?”
          fi
      ;;
   esac
done
```

- Using wildcards in a script
  - Example:
    - **ls /usr/lib/l*z*a**
- Just as if you had entered the ls command at the command line, the asterisks (*) in the argument are expanded by the shell to a list of those files in the /usr/lib directory that start with *l (ell)*, *have a z*, *and* end with an *a*.

- Use shell functions to modularize the script.
  - These are also called as script module
  - Normally defined at the beginning of the script.
  - There are two very basic rules to remember when dealing with functions:
    - **You cannot use a function until it is defined. Thus all function definitions should appear either at the top of the script or in a start-up file such as ~/.profile.**
    - **Functions can be nested to any depth, as long as the first rule is not violated.**
  - Syntax (Function Definition):

```
functionname(){
        commands
        }
```

- Example:

```
i_upper_case()
{
   echo $1 | tr ‘[a-z‘] ['A-Z‘]
}
name="fred"
i_upper_case $name

 OR

small_name="fred smith"
large_name=`i_upper_case "$small_name"`
# Quoted parameter
echo "Large Name = [$large_name]"
```

- Declaring a function before use on the command-line using the built-in function declare:
- Example:

```
declare -f diskusage
diskusage ()
{
    df;
    df -h;
    du -sch
}
```

- Example:

```
Myfunction(){
    echo “$*”
    echo “The number should be between 1 and 20”
    read num
    if [ $num –le 1] –a [$num –ge 20]; then
          return 1;
    else
          return 0;
    fi
    echo “You will never reach to this line”
    }
    echo “Calling the function Myfunction”
    if   Myfunction “Enter the number”
    then
        echo “The number is within range”
    else
       echo the number is out of range”
```

- Example

```
greet () {
        echo "Hello. "
        getName
}
getName () {
        echo "Welcome to Reliance Communications"
}
greet
```

```
flowers[0]=Rose
flowers[1]=Lotus
flowers[2]=Mogra
i=0
while [ $i -lt 3 ]
do
        echo ${flowers[$i]}
        i=`expr $i+1`
done
```

To access all elements:

```
${array_name[*]}
${array_name[@]}
```

- Shift:
  - Reassigns positional parameters in effect shifting them to the left one at a time
- Example:

```
variable="one two three four five”
 set -- $variable  # Sets positional parameters to the contents of
   "$variable“.
 first_param=$1
 second_param=$2
 shift ; shift
# Shift past first two positional params.
remaining_params="$*”
 echo ; echo "first parameter = $first_param"
 echo "second parameter = $second_param“
echo "remaining parameters = $remaining_params”
```

Reliance Tech Services
Anil Dhirubhai Ambani Group

UNIX – Basic Commands

AWK

AWK

- Named after Aho, Weinberger, Kernigham.
- As powerful as any programming language.
- It can access, transform and format individual fields in a record – it is also called as a report writer. It can accept regular expressions for pattern matching, has “C” type programming constructs, variables and in-built functions. Based on *pattern matching* and *performing action*.
- Limitations of the *grep* family are:
  - **No options to identify and work with fields.**
  - **Output formatting, computations etc. is not possible.**
  - **Extremely difficult to specify patterns or regular expression always.**
- AWK overcomes all these drawbacks.

- Syntax:
  - awk <options> ‘line specifier {action}’ <files>
- Example:

  awk '{ print $0 }' emp.lst
  - This program prints the entire line from the file *emp.lst*.
  - $0 refers to the entire line from the file *emp.data*.
- Example:

  awk '/director/ {print}' emp.lst

- Variable List:
  - **$0:** Contains contents of the full current record.
  - **$1..$n:** Holds contents of individual fields in the current record.
  - **NF:** Contains number of fields in the input record.
  - **NR:** Contains number of record processed so far.
  - **FS:** Holds the field separator. Default is *space* or *tab*.
  - **OFS:** Holds the output field separator.
  - **RS:** Holds the input record separator. Default is a new line.
  - **FILENAME:** Holds the input file name.
  - **ARGC:** Holds the number of Arguments on Command line
  - **ARGV:** The list of arguments

- Example
  - **awk -F " | " '{ print $1 $2 $3 }' emp.lst**
    - **This prints the *first*, *second* and *third* column from file**
  - **awk '{ print }' emp.data**
  - Prints all lines (all the fields) from file *emp.data*.
- Fields are identified by special variable $1, $2, ….;
- Default delimiter is a contiguous string of spaces.
- Explicit delimiter can be specified using -F option
  - **Example: awk -F " | " '/sales/{print $3, $4}' emp.lst**
  - **awk -F"|" '$1=="1002" {printf "%2d,%-20s",NR,$2}' emp.lst**
  - **awk -F " | " '$5 > 7000 { print $1, $2 * $3 }' emp.lst**

- Line numbers can be selected using NR built-in variable.
  - **awk -F " | " 'NR ==3, NR ==6 {print NR, $0}' emp.lst**
  - **awk '{ print NF, $1, NR }' emp.data**
  - **awk '$3 == 0' emp.data**
  - **awk '{ print NR, $0 }' emp.data**
  - **awk ' $1 == "Susie" ' emp.data**
- Relational and Logical Operators can also be used.
  - **$awk - F "|" '$3 ==" director" || $3 == "chairman"{printf "%-20s",$2}' emp.lst**
  - **$awk -F "|" '$6>7500 {printf "%20s", $2}' emp.lst**
  - **$awk -F "|" '$3 == "director" || $6>7500 {print $0}' emp.lst**

- Computations can also be performed in AWK

```
$awk -F "|" '$3 == "director" || $6>7500 {
    kount = kount+1
    printf "%3d%-20s\n", kount,$2}' emp.lst
```

- Programming can be done in separate file.
  - Example: awk1

```
$3 == "Director" {print $2,$4,$5}
```

  - To execute the AWK command using the file

```
awk -F "|" –f  awk1  emp.lst
```

- BEGIN and END Section:
- In case, something is to be printed before processing the first line begins, BEGIN section can be used.
  - Normally if you want to print any header lines or want to set field separator then use BEGIN section
- Similarly, to print at the end of processing, END section can be used.
  - And If you want to display total or any summarized information at the end of the report then use END section
- These sections are optional. When present, they are delimited by the body of the awk program.
  - Format: (i) BEGIN {action} (ii) END {action}

- Example:

```
BEGIN {
      printf “\n\t Employee details \n\n”
}
$6>7500{
      # increment sr. no. and sum salary
   kount++; tot+=$6
   printf “%d%-20s%d\n”, kount, $2, $6
}
END {
      printf “\n The Avg. Sal. Is %6d\n”, tot/kount
}
```

```
$awk -F “|” -f emp.awk emp.lst
```

- It is possible to store an entire awk command into a file as a shell script, and pass parameters as arguments to the shell script. The shell will identify these arguments as $1, $2 etc based on the order of their occurrence on the command line.
- Within awk, since $1, $2 etc. indicate fields of data file, it is necessary to distinguish between the positional parameters and field identifiers. This is done by using single quotes for the positional parameters used by awk.
  - **Positional parameter should be single-quoted in an *awk* program.**
    - **Example: $3 > '$1'**

- To make AWK interactive we can use the getline **statement**
- Example:

```
BEGIN{
        printf "\nEnter Cut-off basic pay:"
        getline cobp < "/dev/tty"
        printf "\n\t Employee List \n\n"
}
        $6 > cobp {
                printf "%3d %-20s %-12s %d \n",++kount,$2,$3,$6
        }
```

- Awk can handle one-dimenasional arrays.
- Example:

```
BEGIN{ FS="|"
        printf "\n%38s\n","Basic   Da   Hra Gross"
}
/sales|marketing/{
        da=0.25*$6
        hra=0.50*$6
        gp=$6+hra+da
        total[1]+=$6
        total[2]+=da
        total[3]+=hra
        total[4]+=gp
        kount++
}
END{
```

Shell Variables

- There are several variables set by the system - some during booting and some after logging in. These are called the system variables, and they determine the environment one is working in. The user can also alter their values.
- The set statement can be used to display list of system variables.
  - **Set**
- Shell Variables
  - PATH : Contains the search path string.
    - **Determines the list of directories (in order of precedence) that need to be scanned while you look for an executable command.**
    - **Path can be modified as: $ PATH=$PATH:/usr/user1/progs**
    - **This causes the /usr/user1/progs path to get added to the existing PATH list.**

- HOME : Specifies full path names for user login directory.
- TERM : Holds terminal specification information
- LOGNAME : Holds the user login name.
- PS1 : Stores the primary prompt string.
- PS2 : Specifies the secondary prompt string.

- Example:

```
# Checks for environment variables.
# Uncomment the following line to remove the variable.
#unset DISPLAY
if [ “$DISPLAY” == “” ]
then
  echo “DISPLAY not set, using :0.0 as default.”
  DISPLAY=”:0.0”
fi
#unset SHELL
if [ “$SHELL” == “” ]
Then
  echo “Using /bin/bash, which is the shell you should use.”
  SHELL=/bin/bash
fi
```

```
#unset USER
if [ “$USER” == “” ]
then
   echo -n “Please enter your username: “
read USER
fi
#unset HOME
if [ “$HOME” == “” ]
then
# Check for Mac OS X home.
   if [ -d “/Users/$USER” ]
   then
      HOME=”/Users/$USER”
# Check for Linux home.
   elif [ -d “/home/$USER” ]
then
```

```
    HOME="/home/$USER"
  else
    echo -n "Please enter your home directory: "
    read HOME
    echo
  fi
fi
# Display all the values.
echo "DISPLAY=$DISPLAY"
echo "SHELL=$SHELL"
echo "USER=$USER"
echo "HOME=$HOME"
```