UNIX

Shell Programming

Introduction

- **System Variables**
  - Set during:
    - Boot
    - Login
- **.profile:**
  - Script executed at login.
  - Alters operating environment of a user.
- **$set**
  - Displays a list of system variables.

Standard shell variables

- **Shell Variables**
  - PATH : Contains the search path string.
  - 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.

Scripts executed automatically

- **.profile script**
  - shell script that gets executed by the shell when the user logs on
  - Used by Bourne shell
- **.cshrc ,.login**
  - Used by C Shell users
  - *.login* and is read when the user logs in.
  - *.cshrc* and is read whenever a new C shell is created
- **.logout script**
  - .logout file can also be created for commands to be executed when you log out.

Example

- **Simple Shell Script: Accept Name & Display Message**

hello.sh

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

- **To execute the shell script**

$sh hello.sh

- **To debug the shell script use –x option**

$sh –x hello.sh

Details

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

- **In the above example, instead of *expr* we can use *let*.**
  - Syntax:
    - let *expressions* or ((*expressions*))
  - In above script res=`expr $no1 + $no2` can be replaced by

    let res=no1+no2

Details

- **Command is enclosed in backquotes (`).**
- **Shell executes the command first.**
  - Enclosed command text is replaced by the command output.
- **Display output of the date command using echo:**

```
$echo The date today is `date`
        The date today is Fri 27 00:12:55 EST 1990
```

- **Issue echo and date commands sequentially:**

```
$echo The date today is; date
```

Example

- **Following instructions print pwd as a string:**

  - **var=pwd**

    echo $var

    Output: pwd

- **Following instructions execute PWD shell command and display the present working directory:**

```
var=`pwd`
        echo $var
        Output:   /usr/deshpavan
```

Details

- **Specify arguments along with the name of the shell program on the command line called as command line argument.**
- **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

Details

- **Arguments are assigned to special variables (positional parameters).**
  - $1 - First parameter , $2 - Second parameter,….
  - Example:

```
echo Program: $0
        echo Number of arguments are $#
        echo arguments are $*
grep "$1" $2
echo "\n End of Script“
```

  - Run script:

```
$ scr1.sh "Unix" books.lst          --$1 is UNIX  , $2 –books.lst
```

Details

- **Logical Operators && and ||:**
  - **&&** operator delimits two commands. Second command is executed only if the first *succeeds*.
  - || operator delimits two commands. Second command is executed only if the first *fails*.
  - Example:

```
$grep `director` emp.lst && echo “pattern found”
$grep `manager` emp.lst || echo “pattern not found”
```

Details

- Syntax

(i) if <condition is true>

then

<execute commands>

else

<execute commands>

fi

(ii) if <condition is true>

then

<execute commands>

fi

- Example

```
if grep "^$1" /etc/passwd 2>/dev/null
  then
        echo "pattern found"
  else
        echo "pattern not found"
fi
```

if Statement

- Syntax:

(iii) if <condition is true>

then

<execute commands>

elif <condition is true>

then

<execute commands>

<… >

else

<execute commands>

fi

- Example

```
if test $# -eq 0; then
  echo "wrong usage " > /dev/tty
  elif test $# -eq 2 ; then
    grep "$1" $2 || echo "$1 not
          found in $2" > /dev/tty
   else
      echo "you didn't enter 2
                  arguments"
fi
```

Relational Operator for numbers

- **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

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

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

- **Check whether user has entered a filename or not:**
  - Example:

```
echo "Enter File Name:\c "
read fn
if [ -z “$fn” ]
then
  echo “You have not entered file name“
fi
```

Example

- **Example:**

```
if test $x -eq $y
                    ≡ if [  $x -eq $y ]
```

- **Example:**

```
If  [ ! -f  fname ]
        then
                echo “file does not exists”
        fi
```

Example

```
echo “Enter the source file name : \c”
read source
#check for the existence of the source file
if test –s “$source”   #file exists & size is > 0
then
    if test ! –r  “$source”
    then
            echo “Source file is not readable”
            exit
    else
         cat $source
    fi
else
    echo “Source file not present”
    exit
fi
```

Case command

- Syntax:

```
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 ;;
5) exit ;;
esac
```

Example

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

Syntax and Example

- Syntax:

```
while <condition is true>

do

        <execute statements>

done
```

e.g.

```
while [ $x -gt 3 ]
        do
                ps -a
                sleep 5
        done
```

```
while true
        do
                ps -a
                sleep 5
        done
```

Example : While

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

break and continue statement

- **Continue:**
  - Suspends statement execution following it.
  - Switches control to the top of loop for the next iteration.

- **Break:**
  - Causes control to break out of the loop.

Example

```
while echo "designation : \c"
                    do
                    read desig
                    case "$desig" in
                       [0-9]) if grep "^$desig" emp.lst >/dev/null
                                    then
```

Syntax

- **Complement of *while* statement.**
- **Loop body executes repeatedly as long as the condition remains *false*.**
  - Example:

```
until false
do
        ps -a
        sleep 5
done
```

for statement

- Syntax:

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

- Eg:

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

```
for var in $PATH  $HOME  $MAIL
    do
        echo "$var"
    done
```

```
for file in *.c
    do
        cc $file
    done
```

Example : for

```
for file in chap20 chap21 chap22 chap23;
    do
    cp $file ${file}.bak
    echo $file copied to $file.bak
done
```

```
for file in 'cat clist'.....
```

```
for file in *.htm *.html;
do
    # do something
done
```

```
for pattern in "$@"; do
    grep "$pattern" emp.lst || echo "$pattern not found"
done
```

Details

- Syntax:

```
for (( expr1; expr2; expr3
    ))
    do
    ..... ... repeat all
    statements between
    do and done until
    expr2 is TRUE

done
```

e.g.

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

Example : Until

```
#script to create a employee file
ans=”y”
until [ $ans = “N” –o $ans = “n” ]
do
        echo “Enter the name :\c”
        read name
        echo “Enter the grade :\c”
        read grade
        echo “Enter the basic :\c”
        read basic
        echo $name: $grade : $basic >>emp
echo “Want to continue (Y/N) :\c”
read ans
done
#end of script
```

Functions in Shell Script

- Use shell functions to modularize the script.
- These are also called as script module
- Normally defined at the beginning of the script.
- Syntax (Function Definition):

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

- Example: Function to create a directory and change directories:
- Use mkcd mydir to call the function. mydir is used as $1 in the function.

```
mkcd()
{
mkdir $1      --$1 is the argument we pass while calling function
cd $1
}
```

Using return statement

- **Used to come out of a function from within.**
  - If called *without* an argument, function return value is the same as *exit* status of the last command executed within the function
  - If called *with* an argument it returns the argument specified.
  - Example:

```
functret()
{
command1
if …….
then
    return 1
else
    return 0
fi
Command2
}
```

Using return statement

```
 Myfunction(){
echo "$*"
echo "The number should be between 1 and 20"
read num
if [ $num –le 1] –a [$num –ge 20]
      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"
```

Using arrays

- **Contains a collection of values accessible by individuals or groups**
  - Subscript of array element indicates their position in the array.
    - arrayname[subscript]
- **First element is stored at subscript 0.**
  - Assign a value in *flowers* array at the first position.
    - Flowers[0]=Rose
- **Assign values in an array with a single command:**
  - $ set -A Flowers Rose Lotus
- **Access individual array elements**
  - ${arrayname[subscript]}

Using arrays

- **To print values from array we can use while loop**

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

- **Access all elements:**

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