Shell Programming

What Is Kernal ?

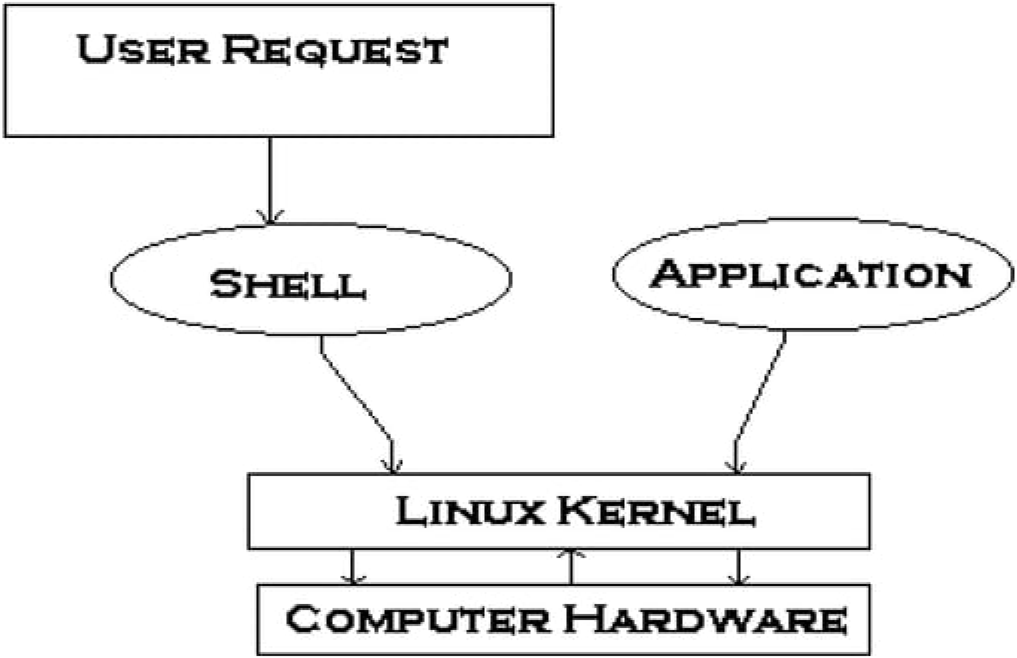
- Kernel is hart of Linux Os.

- It manages resource of Linux Os. Resources means facilities available in Linux. For e.g. Facility to store data, print data on printer, memory, file management etc .

- Kernel decides who will use this resource, for how long and when. It runs your programs (or set up to execute binary files).

- The kernel acts as an intermediary between the computer hardware and various programs/application/shell.

Kernal con.


User Request
Shell
Application
Linux Kernel
Computer Hardware

- It's Memory resident portion of Linux. It performance following task :-
- I/O management
- Process management
- Device management
- File management
- Memory management

What Is a Shell?

- A *shell is a program that takes commands typed by the user and calls the operating system to run those commands.*
- A *shell is a program that acts as the interface between you and the Linux system, allowing you to enter* commands for the operating system to execute.
- Shell accepts your instruction or commands in English and translate it into computers native binary language

Why Use Shells?

- You can use shell scripts to automate administrative tasks.
- Encapsulate complex configuration details.
- Get at the full power of the operating system.
- The ability to combine commands allows you to create new commands
- Adding value to your operating system.

Kind of Shells

- *Bourne Shell*
- *C Shell*
- *Korn Shell*
- *Bash Shell*
- *Tcsh Shell*

Changing Your Default Shell

- Tip: To find all available shells in your system type following command:

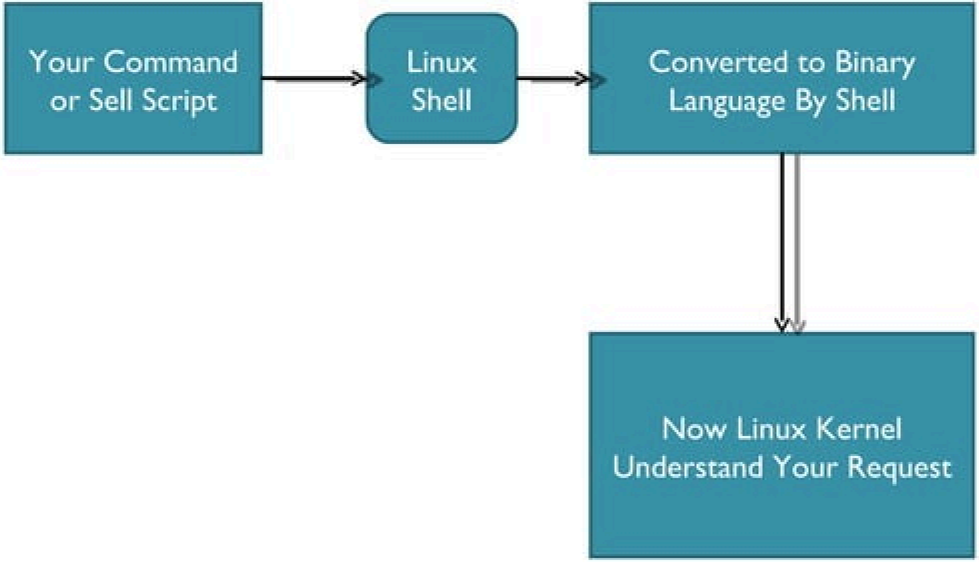
```
$ cat /etc/shells
```

- The basic Syntax :

```
chsh username new_default_shell
```

- The administrator can change your default shell.

This is what Shell Does for US



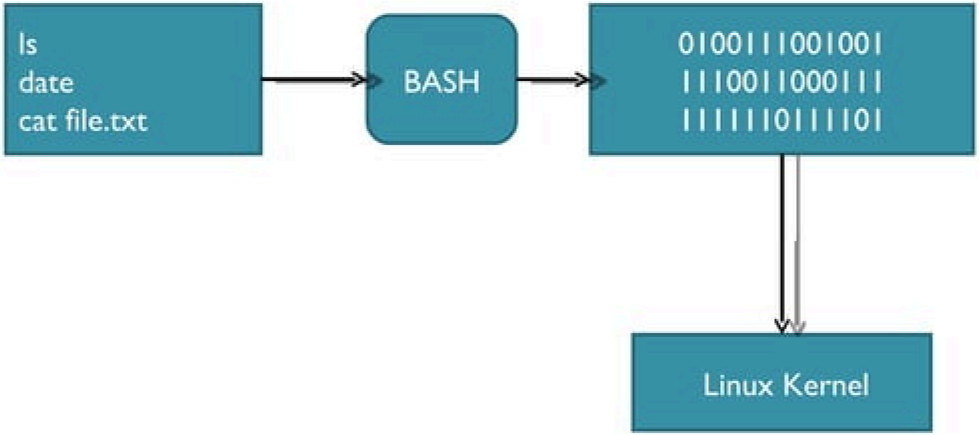
Your Command or Sell Script

Linux Shell

Converted to Binary Language By Shell

Now Linux Kernel Understand Your Request

Example



ls
date
cat file.txt

BASH

0100111001001
1110011000111
1111110111101

Linux Kernel

Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file.

The Shell as a Programming Language

- Now that we've seen some basic shell operations, it's time to move on to scripts.
- There are two ways of writing shell programs.

1. You can type a sequence of commands and allow the shell to execute them interactively.
2. You can store those commands in a file that you can then invoke as a program(shell script).

Shell Scripting

- *Shell script is a series of command(s) stored in a plain text file.*

- *A shell script is similar to a batch file in MS-DOS, but is much more powerful.*

Why to Write 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.
- System Administration part can be also automated.

Practical examples where shell scripting actively used:

1. Monitoring your Linux system.
2. Data backup and creating snapshots.
3. Find out what processes are eating up your system resources.
4. Find out available and free memory.
5. Find out all logged in users and what they are doing.
6. Find out if all necessary network services are running or not.

Create a script

- As discussed earlier shell scripts stored in plain text file, generally one command per line.
  - vi myscript.sh

- Make sure you use .bash or .sh file extension for each script. This ensures easy identification of shell script.

Setup executable permission

- Once script is created, you need to setup executable permission on a script. Why?
  - Without executable permission, running a script is almost impossible.
  - Besides executable permission, script must have a read permission.
- Syntax to setup executable permission:
  - $ chmod +x your-script-name.
  - $ chmod 755 your-script-name.

Run a script (execute a script)

- Now your script is ready with proper executable permission on it. Next, test script by running it.
  - bash your-script-name
  - sh your-script-name
  - ./your-script-name
- Examples
  - $ bash bar
  - $ sh bar
  - $ ./bar

Example

```
$ vi first
```

```
#
# My first shell script
#
clear
echo “This is my First
script”
```

```
$ chmod 755 first


$ ./first
```

Variables in Shell

- In Linux (Shell), there are two types of variable:
  - **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.
  - **User defined variables (UDV)** - Created and maintained by user. This type of variable defined in lower letters.

User defined variables (UDV)

- To define UDV use following syntax:
  - variable name=value
  - $ no=10
- **Rules for Naming variable name**
  - Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character.
  - Don't put spaces on either side of the equal sign when assigning value to variable.
  - Variables are case-sensitive.
  - You can define NULL variable
  - Do not use **?,*** etc, to name your variable names.

Print or access value of UDV

- To print or access UDV use following syntax :
  - $variablename.
- Examples:
  - $vech=Bus
  - $ n=10

  - $ echo $vech
  - $ echo $n

Cont…

- Don’t try
    - **$ echo vech**
    - it will print vech instead its value 'Bus‘.

    - **$ echo n**
    - it will print n instead its value '10‘.
- You must *use* $ *followed by variable name.*

Class work

1. Define variable x with value 10 and print it on screen.
2. Define variable xn with value SUST and print it on screen.
3. print sum of two numbers, let's say 6 and 3 .

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`

The read Statement

- Use to get input (data from user) from keyboard and store (data) to variable.
- *Syntax:*
  - read variable1, variable2,...variableN

```
echo "Your first name please:"
read fname

echo "Hello $fname, Lets be friend!"
```

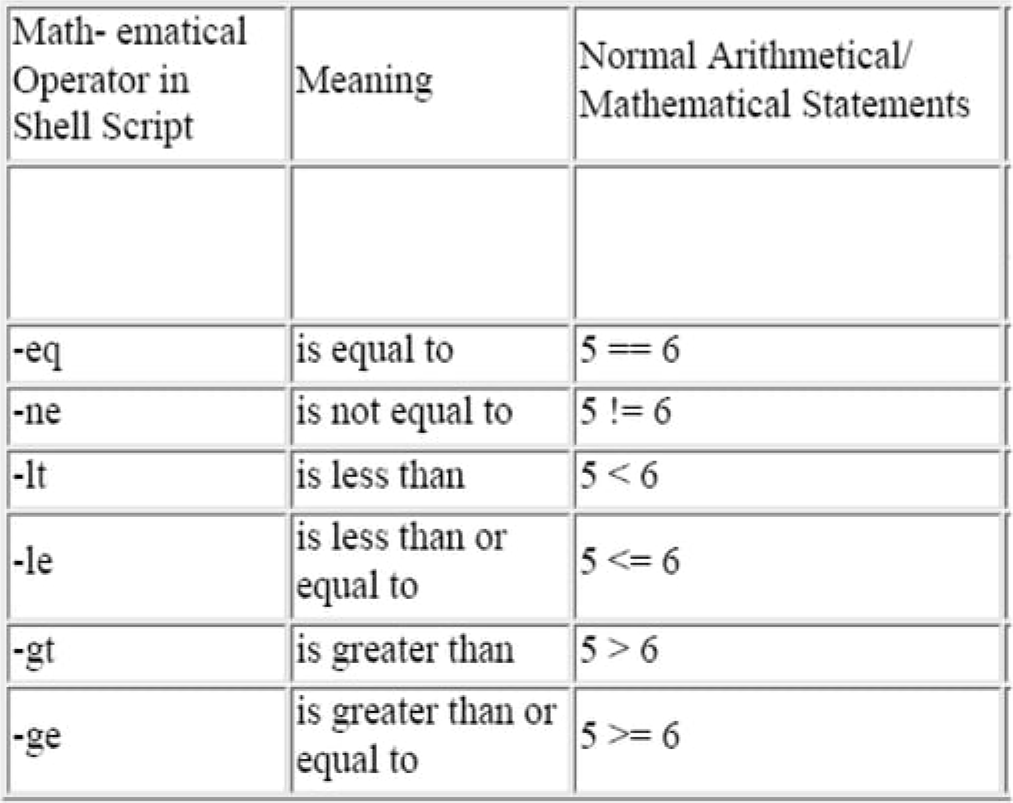
Shorthand

Shorthand	Meaning
$ 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'.
$ 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
$ ls [abc]*	will show all files beginning with letters a,b,c

if condition

Syntax:

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



Math- ematical Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements
-eq	is equal to	5 == 6
-ne	is not equal to	5 != 6
-lt	is less than	5 < 6
-le	is less than or equal to	5 <= 6
-gt	is greater than	5 > 6
-ge	is greater than or equal to	5 >= 6

Example

- $ vim myscript.sh

```
 read choice

if [ $choice -gt 0 ]; then
echo "$choice number is positive"
else
echo "$ choice number is negative"
fi
```

Nested if-else-fi

- $ vi nestedif.sh

```
echo "1. Unix (Sun Os)"
echo "2. Linux (Red Hat)"
echo -n "Select your os choice [1 or 2]? "
read osch

if [ $osch -eq 1 ] ; then

    echo "You Pick up Unix (Sun Os)"

else
      if [ $osch -eq 2 ] ; then
            echo "You Pick up Linux (Red Hat)"
      else
            echo "What you don't like Unix/Linux OS."
     fi
fi
```

Loops in Shell Scripts

- Bash supports:

2. for loop.
3. while loop.

- **Note** that in each and every loop:
  - First, the variable used in loop condition must be initialized, then execution of the loop begins.
  - A test (condition) is made at the beginning of each iteration.
  - The body of loop ends with a statement that modifies the value of the test (condition) variable.

for Loop

Syntax:

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
```

Example

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

for Loop

- *Syntax:*

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

Example

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

Nesting of for Loop

- $ vi nestedfor.sh

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

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

  echo “”
don
```

while loop

- *Syntax:*

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

Example

```
i=1
while [ $i -le 10 ]
 do
   echo "$n * $i = `expr $i \* $n`"
   i=`expr $i + 1`
 done
```

The case Statement

- *Syntax:*

```
case $variable-name in
   pattern1 ) command…..;;
  pattern2) command…..;;
  pattern N) command….;;
  .
 *) command  ;;
esac
```

Example

```
 read var
  case $var in
   1) echo “One”;;
   2) echo “Two”;;
   3) echo “Three”;;
   4) echo “Four”;;
   *) echo "Sorry, it is bigger than Four";;
esac
```

Functions

- Function is series of instruction/commands.
- Function performs particular activity in shell.
- *Syntax:*

```
function-name ( )
{
   Function body
}
```

Example

```
today()


today()
  {
  echo “Today is `date`”
 return
  }
```

Example

```
function cal()
    {
    n1=$1
    op=$2
    n2=$3
    ans=0
    if [ $# -eq 3 ]; then
      ans=$(( $n1 $op $n2 ))
      echo "$n1 $op $n2 = $ans"
      return $ans
    else
      echo "Function cal requires atleast three args"
    fi
     return
    }
```

Cont…

```
cal 5 + 10
cal 10 - 2
cal 10 / 2
echo $?
```

Example

```
while :
do
     clear
     echo "--------------------------------------"
     echo " Main Menu "
     echo "--------------------------------------"
     echo "[1] Show Todays date/time"
     echo "[2] Show files in current directory“
     echo "[3] Show calendar"
     echo "[4] Start editor to write letters"
     echo "[5] Exit/Stop"
     echo "=========================="
     echo -n "Enter your menu choice [1-5]: "
     read yourch
```

Cont…

```
case $yourch in
    1) echo "Today is `date` , press a key. . ." ; read ;;
    2) echo "Files in `pwd`" ; ls -l ; echo "Press a key. . ." ; read ;;
    3) cal ; echo "Press a key. . ." ; read ;;
    4) vi ;;
    5) exit 0 ;;
    *) echo "Opps!!! Please select choice 1,2,3,4, or 5";
    echo "Press a key. . ." ; read ;;
esca
done
```

Class work#2

--------menu---------

Working with Files

- "On a UNIX system, everything is a file; if something is not a file, it is a process.“
- *Directories*: files that are lists of other files.
- *Special files*: the mechanism used for input and output. Most special files are in /dev, we will discuss them later.

- *Links*: a system to make a file or directory visible in multiple parts of the system's file tree. We will talk about links in detail.
- *(Domain) sockets*: a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.

- *Named pipes*: act more or less like sockets and form a way for processes to communicate with each other, without using network socket semantics.

Symbol	Meaning
-	Regular file
d	Directory
l	Link
c	Special file
S	Socket
P	Named pipe
C	character (unbuffered) device file special
b	block (buffered) device file special