System and Network Engineering - Lecture 5

$ Intro to Bash scripting

Why use bash scripting

Why do we need shell scripts:

- ❏ To avoid repetitive work and automation
- ❏ System engineers use shell scripting for routine backups, daily tasks etc
- ❏ System monitoring
- ❏ Adding new functionality to the shell
- ❏ Shell language is used in many DevOps technologies to execute tasks in the environment

Limitations of bash scripting

Why do we need shell scripts:

- ❑ Prone to costly errors, a single mistake can change the command which might be harmful
- ❑ Slow execution speed
- ❑ Not well suited for large and complex task
- ❑ Provide minimal data structure unlike other scripting languages

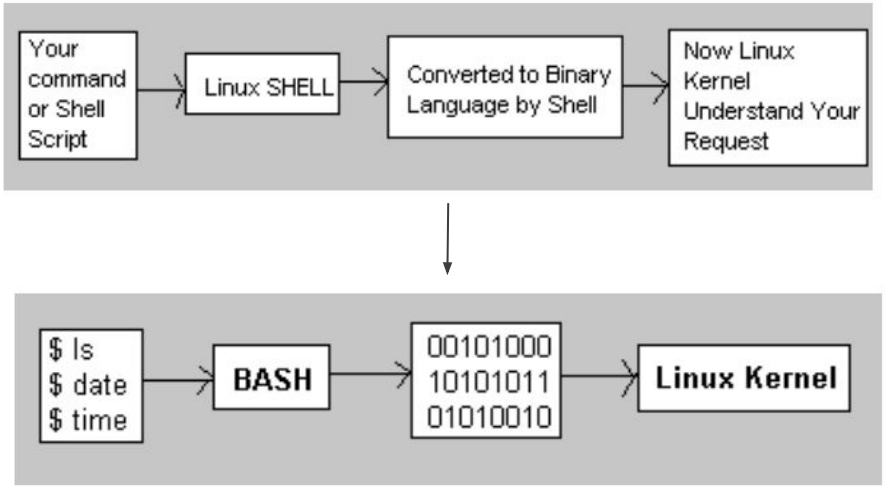
Define a bash script

- You can use plain **Bourne shell** (/bin/sh) when special features of **Bash** (/bin/bash) are not needed
- Bourne shell is proprietary code, whereas Bash is open source
- Note that Bourne shell (/bin/sh) is usually just a symlink to dash (/usr/bin/dash) on Linux systems
  - **dash** is a POSIX-compliant implementation of /bin/sh that aims to be as small as possible

```
saltanov@linuxPC:~$ file $(which bash)
/usr/bin/bash: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=33a5554034feb2af38e8c75872058883b2988bc5, for GNU/Linux 3.2.0
, stripped
saltanov@linuxPC:~$ file $(which dash)
/usr/bin/dash: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, BuildID[sha1]=f7ab02fc1b8ff61b41647c1e16ec9d95ba5de9f0, for GNU/Linux 3.2.0
, stripped
```

Define a bash script

- Bash script is a just a list of consequent commands
- The **#!/bin/bash** tells the shell to invoke /bin/bash to run the script
  - Use it at the beginning of the script file
- Script Execution:
  - **$ bash script_name**
  - **$ chmod u+x First.sh && ./script_name.sh**



Bash scripting usage

- Usage of **echo** command:

```
#!/bin/bash
echo "Printing text with newline"
echo -n "Printing text without newline"
echo -e "\nRemoving \t backslash \t characters\n"
```

```
saltanov@linuxPC:~/123$ ./123.sh
Printing text with newline
Printing text without newline
Removing         backslash       characters
```

- Usage of **comments**:

```
#!/bin/bash
# Add two numeric value
((sum=25+35))

#Print the result
echo $sum
```

```
#!/bin/bash
: '
The following script calculates
the square value of the number, 5.
'
((area=5*5))
echo $area
```

```
saltanov@linuxPC:~/123$ ./123.sh
60
saltanov@linuxPC:~/123$ ./321.sh
25
```

Bash scripting usage

- Using **While** Loop:

```
#!/bin/bash
valid=true
count=1
        while [ $valid ]
        do
          echo $count
          if [ $count -eq 5 ];
          then
          break
          fi
          ((count++))
        done
```

```
saltanov@linuxPC:~/123$ ./123.sh
1
2
3
4
5
```

Bash scripting usage

- Using **For** Loop:

```
#!/bin/bash
for (( counter=10; counter>0; counter-- ))
  do
  echo -n "$counter "
  done
printf "\n"
```

```
saltanov@linuxPC:~/123$ ./123.sh
10 9 8 7 6 5 4 3 2 1
```

- Using **if** statement:

```
#!/bin/bash
n=10
  if [ $n -lt 10 ];
  then
    echo "It is a one digit number"
  else
    echo "It is a two digit number"
  fi
```

```
saltanov@linuxPC:~/123$ ./123.sh
It is a two digit number
```

Bash scripting usage

- Using **if** statement with **logical** operators

```
#!/bin/bash

echo "Enter username"
read username
echo "Enter password"
read password

   if [[ ( $username == "admin" && $password == "secret" ) ]]; then
     echo "valid user"
   else
     echo "invalid user"
   fi
```

```
saltanov@linuxPC:~/123$ ./123.sh
Enter username
user
Enter password
secret
invalid user
```

- Get **User** Input:

```
#!/bin/bash
echo "Enter Your Name"
read name
echo "Welcome $name to LinuxHint"
```

```
saltanov@linuxPC:~/123$ ./123.sh
Enter Your Name
Kirill
Welcome Kirill to LinuxHint
```

Bash scripting usage

- Using **else if** statement:

```
#!/bin/bash

echo "Enter your lucky number"
read n

        if [ $n -eq 101 ];
        then
          echo "You got 1st prize"
        elif [ $n -eq 510 ];
        then
          echo "You got 2nd prize"
        elif [ $n -eq 999 ];
        then
          echo "You got 3rd prize"
        else
          echo "Sorry, try for the next time"
        fi
```

```
saltanov@linuxPC:~/123$ ./123.sh S
Enter your lucky number
123
Sorry, try for the next time
saltanov@linuxPC:~/123$ ./123.sh
Enter your lucky number
999
You got 3rd prize
```

Bash scripting usage

- Using **Case** Statement:

```
#!/bin/bash

echo "Enter your lucky number"
read n
        case $n in
        101)
        echo "You got 1st prize" ;;
        510)
        echo "You got 2nd prize" ;;
        999)
        echo "You got 3rd prize" ;;
        *)
        echo "Sorry, try for the next time" ;;
        esac
```

```
saltanov@linuxPC:~/123$ ./123.sh
Enter your lucky number
510
You got 2nd prize
saltanov@linuxPC:~/123$ ./123.sh
Enter your lucky number
111
Sorry, try for the next time
```

Bash scripting usage

- Get **Arguments** from Command Line:

```
#!/bin/bash
echo "Total arguments : $#"
echo "1st Argument = $1"
echo "2nd argument = $2"
```

```
saltanov@linuxPC:~/123$ ./123.sh
Total arguments : 0
1st Argument =
2nd argument =
saltanov@linuxPC:~/123$ ./123.sh hello there
Total arguments : 2
1st Argument = hello
2nd argument = there
```

- Get **Arguments** from command line with **names**:

```
#!/bin/bash
        for arg in "$@"
        do
          index=$(echo $arg | cut -f1 -d=)
          val=$(echo $arg | cut -f2 -d=)
          case $index in
          X) x=$val;;

          Y) y=$val;;

          *)
        esac
        done
((result=x+y))
echo "X+Y=$result"
```

```
saltanov@linuxPC:~/123$ ./123.sh X=11 Y=12
X+Y=23
```

Bash scripting usage

- Combine **String** variables:

```
#!/bin/bash

string1="Linux"
string2="Hint"
echo "$string1$string2"
string3=$string1+$string2
string3+=" is a good tutorial blog site"
echo $string3
```

```
saltanov@linuxPC:~/123$ ./123.sh
LinuxHint
Linux+Hint is a good tutorial blog site
```

- Get **substring** of **String**:

```
#!/bin/bash
Str="Learn Linux from LinuxHint"
subStr=${Str:6:5}
echo $subStr
```

```
saltanov@linuxPC:~/123$ ./123.sh
Linux
```

Bash scripting usage: Function

- A function is a block of code that performs a specific task and which is reusable
- Functions concept reduces the code length
- Two ways to define a function:

```
function function_name
{
    commands/statements
}
```

```
function_name()
{
    commands/statements
}
```

- When using one liner use {... ;} - **function function_name { commands/statements ; }**

Bash scripting usage: Function

- **Variables Scope**:

```
#!/bin/bash

var1='A'
var2='B'

my_function () {
  local var1='C'
  var2='D'
  echo "Inside function: var1: $var1, var2: $var2"
}

echo "Before executing function: var1: $var1, var2: $var2"

my_function

echo "After executing function: var1: $var1, var2: $var2"
```

```
saltanov@linuxPC:~/123$ ./321.sh 10 200
Before executing function: var1: A, var2: B
Inside function: var1: C, var2: D
After executing function: var1: A, var2: D
```

- When a local variable is set inside the function body with the same name as an existing global variable, it will have precedence over the global variable.
- Global variables can be changed from within the function

Bash scripting usage: Function

- **Return Values**:
  - Unlike functions in "real" programming languages, Bash functions don't allow you to return a value when called. When a bash function completes, its return value is the status of the last statement executed in the function, 0 for success and non-zero decimal number in the 1 - 255 range for failure.
  - The **return** status can be specified by using the **return keyword**, and it is assigned to the variable **$?**. The **return** statement **terminates** the function. You can think of it as the function's **exit status** .

```
#!/bin/bash

my_function () {
  echo "some result"
  return 55
}

my_function
echo $?
```

```
saltanov@linuxPC:~/123$ ./321.sh
some result
55
```

Bash scripting usage: Function

- **Return Values**:
  - assign the result of the function to a global variable
  - send the value to stdout using echo or printf

```
#!/bin/bash

my_function () {
  func_result="some result"
}

my_function
echo $func_result
```

```
#!/bin/bash

my_function () {
  local func_result="some result"
  echo "$func_result"
}

func_result="$(my_function)"
echo $func_result
```

Bash scripting usage: Function

- **Passing Arguments to Bash Functions**:
  - The passed parameters are $1, $2, $3 … $n, corresponding to the position of the parameter after the function's name.
  - The $# variable holds the number of positional parameters/arguments passed to the function.
  - The $* and $@ variables hold all positional parameters/arguments passed to the function.
  - When double-quoted, "$*" expands to a single string separated by space (the first character of IFS) - "$1 $2 $n".
  - When double-quoted, "$@" expands to separate strings - "$1" "$2" "$n".
  - When not double-quoted, $* and $@ are the same.

Bash scripting usage: Function

- **Passing Arguments to Bash Functions**:

```
#!/bin/bash

Rectangle_Area() {
area=$(($1 * $2))
echo "Area is : $area"
}

Rectangle_Area 10 20
```

```
saltanov@linuxPC:~/123$ ./123.sh
Area is : 200
```