Tecnotree

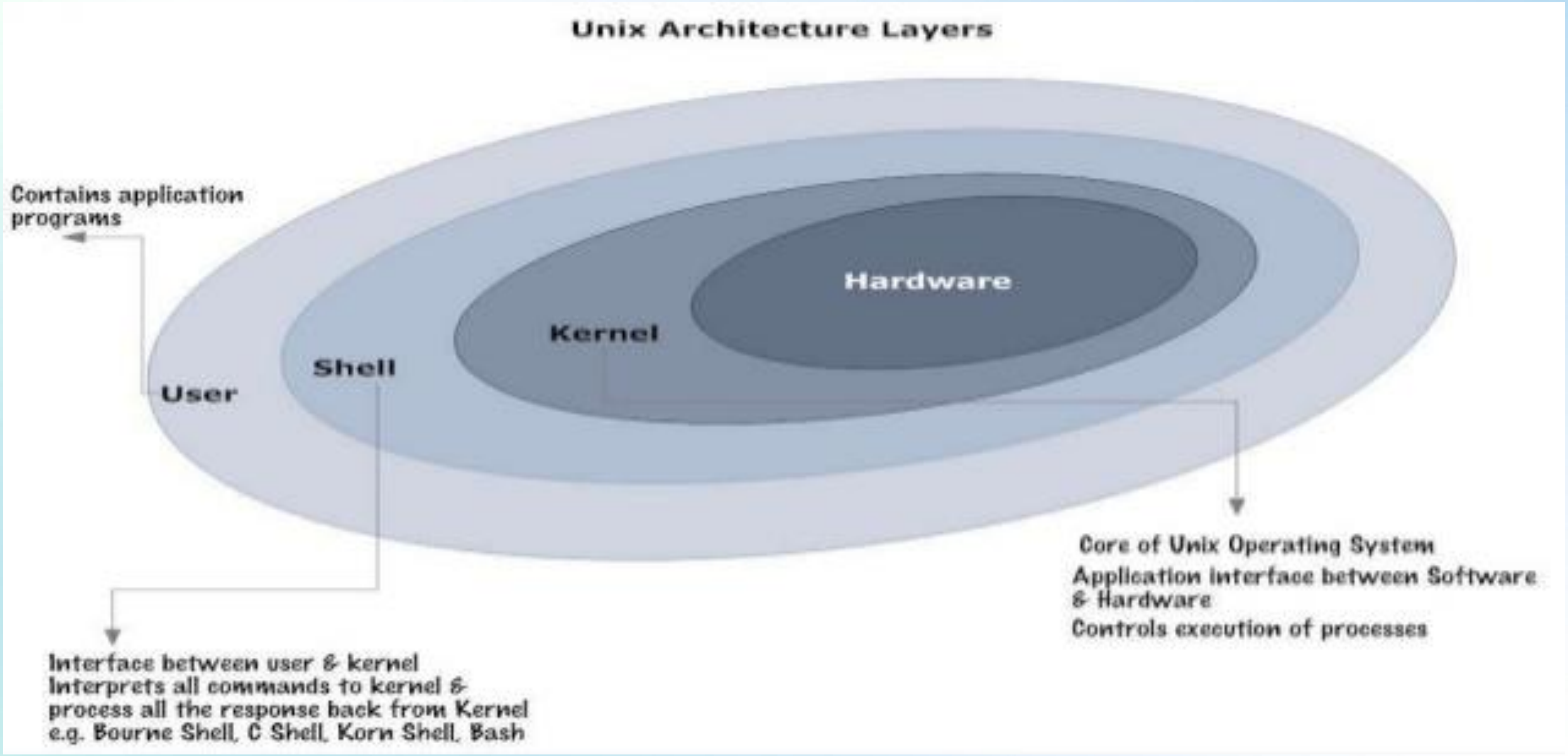
Shell Scripting

4th December, 2019

Manikandan Subramanian

Topics

Kernel and Shell


Unix Architecture Layers
Hardware
Kernel
Shell
User
Contains application programs
Interface between user & kernel
Interprets all commands to kernel & process all the response back from Kernel
e.g. Bourne Shell, C Shell, Korn Shell, Bash
Core of Unix Operating System
Application interface between Software & Hardware
Controls execution of processes

Shell Scripting

- shell scripts / shell programs / shell procedures
- group of UNIX commands stored in a file
- NO restrictions on file extension
- conventional extension → .sh
- She-bang (#!)

  ```
  #!/bin/bash
  #!/bin/sh
  ```

Commonly Used Shells

- **Bourne Shell (sh)** → written by S. R. Bourne, first shell developed for UNIX
- **Korn Shell (ksh)** → written by David Korn, superset of Bourne shell, not widely distributed
- **C Shell (csh)** → written by Bill Joy, (author of *vi* editor), shared much of the C language structure
- **Terminal Based C Shell (tcsh)** → enhanced version of the Berkeley UNIX C shell
- **Bourne Again Shell (bash)** → written by programmers of Free Software Foundation, open source shell from GNU

UNIX Commands

- ls
- cat
- echo
- pwd
- cd
- mkdir
- rmdir
- rm
- cp
- mv

- wc
- sort
- uniq
- head
- tail
- grep
- touch
- ps
- sleep
- kill

- more
- man
- diff
- chmod
- find
- <
- >
- 2>
- >>
- |

- which
- basename
- ssh
- scp
- sftp
- $HOME (~)
- $USER
- $PATH
- export

Variables

- **Format:** `${variable}` or `$Variable`
- **Assign:** `set/unset`
- **Input:** `read var1 [var2 .. varN]`

  each word entered is assigned to each variable
  last variable gets rest of input line

read.sh

- **Command substitution:** backquotes (` `` `) or `$()`
- **Sequential commands:** semicolon (`;`)
- **Command grouping:** `()`
- **Line Comment:** #
- **Here document:** `<<STRING` (Eg: `<<EOF`)
- **Arithmetic Expression:** `expr ${var1} + ${var2}`

Operators

	Meaning	Numeric	String
Relational	Greater than	-gt	>
	Greater than or equal	-ge	>=
	Less than	-lt	<
	Less than or equal	-le	<=
	Equal	-eq	==
	Not equal	-ne	!=
	String length = 0		-z str
	String length > 0		-n str

	Condition	Meaning
Logical	-a	Logical AND
	-o	Logical OR
	!	Logical NOT
File	-e file	file exists
	-d file	file is a directory
	-f file	file is a regular file
	-s file	file size > 0
	-r file	file is readable
	-w file	file is writable

Control Statement - IF, CASE

The IF statement takes decisions depending on the condition that evaluates true

Syntax: if.sh

```
if [ condition ]
then
        commands...
elif [ condition ]
then
        commands...
else
        commands...
fi
```

The CASE statement matches an expression for a matching choice

Syntax: case.sh

```
case <expression> in
   value-1)
       commands...
       ;;
   value-2)
       commands...
       ;;
   *)
       commands...
       ;;
esac
```

Loop Statement - WHILE, UNTIL

WHILE executes the commands repeatedly as long as the condition remains **TRUE**

Syntax: while.sh

```
while [ condition ]
do
        commands...
done
```

Eg:

```
cnt=1
while [ ${cnt} -le 10 ]
do
        echo ${cnt}
        cnt=`expr ${cnt} + 1`
done
```

UNTIL executes the commands repeatedly as long as the condition remains **FALSE**

Syntax: until.sh

```
until [ condition ]
do
        commands...
done
```

Eg:

```
cnt=20
until [ ${cnt} -le 10 ]
do
        echo ${cnt}
        cnt=`expr ${cnt} - 1`
done
```

Loop Statement - FOR

FOR <list> executes the commands repeatedly for every item in the list

Syntax: for.sh

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

Eg:

```
for f in `ls *.sh`
do
        echo ${f}
done
```

FOR <condition> executes the commands repeatedly as long as the condition remains TRUE

Syntax: for2.sh

```
for (( expr; condition; expr ))
do
        commands...
done
```

Eg:

```
for (( cnt=20; cnt>10; cnt-- ))
do
        echo ${cnt}
done
```

Loop Statement - SELECT

SELECT constructs simple menu from word list

SELECT loops till user presses CTRL+D / CTRL+C

User enters sequence number corresponding to the word

$REPLY is reserved variable that contains user entered value

Syntax:

select.sh

```
select <variable> in <list>
do
        commands...
done
```

Eg:

```
PS3="ENTER UR CHOICE: "
select f in `ls *.sh`
do
        echo "Entered No: ${REPLY}"
        more ${f} 2>/dev/null
done
```

Loop Control - BREAK, CONTINUE

BREAK terminates the loop immediately

CONTINUE causes a jump to the next iteration of the loop

Syntax:

break-continue.sh

```
break
continue
```

Eg:

```
for (( i=1; i<=10; i++ ))
do
        if [ ${i} -le 3 ]; then
                echo "skip"
                continue
        fi

        echo ${i}

        if  [ ${i} -ge 8 ]; then
                echo "end"
                break
        fi
done
echo "after loop: ${i}"
```

Parameters / Positional Parameters

`$*`	All positional parameters as one string
`$@`	All positional parameters as set of strings
`$#`	The number of positional parameters
`$1-$9`	Positional parameters 1 through 9
`$0`	Name of the currently executed shell script
`$?`	Returns status of last command/script executed
`$$`	Process ID of current process

Shift

- copies the content of a positional parameter to its lower positional parameter
- content of $2 is copied to $1, $3 to $2 and so on
- using SHIFT command, we can use more than 9 positional parameters

Try this:

```
set 1 2 3 4 5 6 7 8 9 10 11 12 13
echo "Parameters: $*"
echo "5th  Param: $5"
echo "9th  Param: $9"

shift
echo "AFTER shift"
echo "Parameters: $*"
echo "9th  Param: $9"

shift 3
echo "AFTER shift 3"
echo "Parameters: $*"
echo "9th  Param: $9"
```

Functions

- A shell function is similar to a shell script; stores a series of commands
- A shell function is executed in the same shell that its called from
- The parameters of the main script are not directly available inside a function
- Variables defined within functions are global; unless prefixed with `local`
- Functions are defined at the beginning of a shell script
  Also kept in a separate file (like header files)
- To include functions from a file (say common_func.sh)

```
. common_func.sh OR source common_func.sh
```

Syntax:

```
function-name()
{
        commands...
}
```

math.sh function.sh

Debugging

- Check for syntax error only; don’t execute the commands

```
$ bash -n <filename>
```

- Print every command before executing them

```
$ bash -v <filename>
```

- Print every command after command-line processing

```
$ bash -x <filename>
```

- Options can also be set via she-bang line

```
#! /bin/bash -v
```

- `echo` command can also be used for debugging within script

Questions

Tecnotree

Thank you