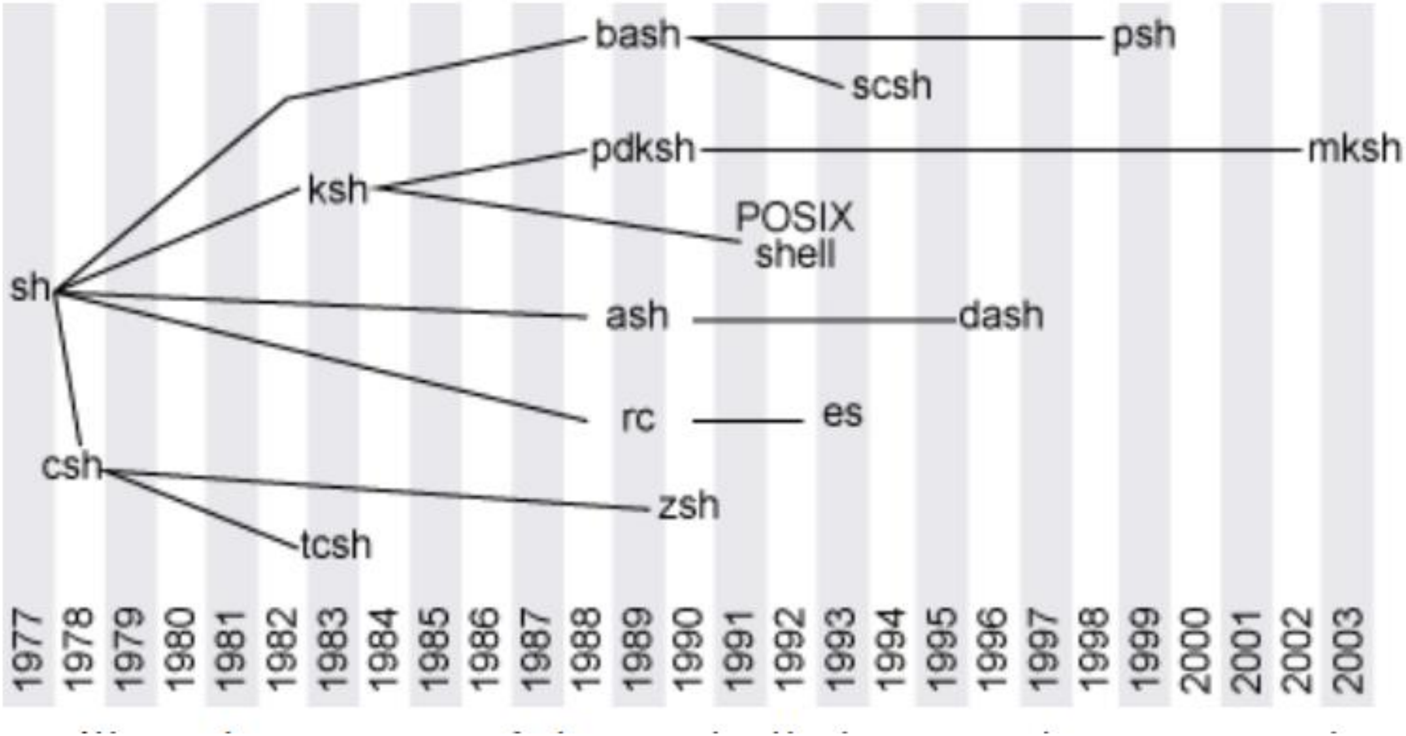
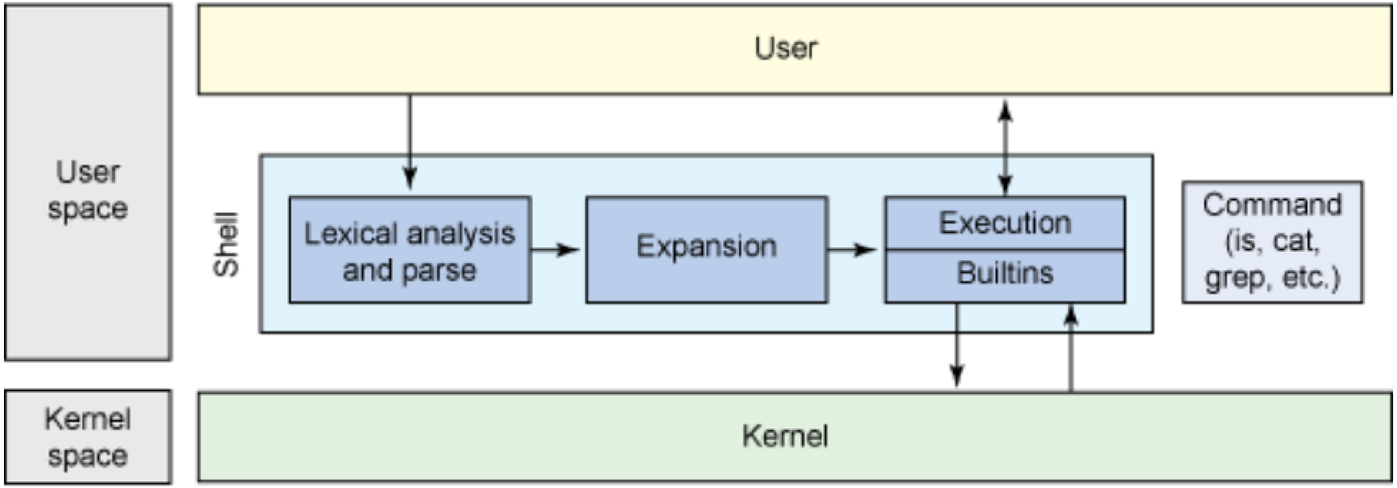
Linux administration with Bash. Lection 1

Shell Evolution



Basic shell architecture

The fundamental architecture of a hypothetical shell is simple (as evidenced by Bourne's shell). As you can see below, the basic architecture looks similar to a pipeline, where input is analyzed and parsed, symbols are expanded (using a variety of methods such as brace, tilde, variable and parameter expansion and substitution, and file name generation), and finally commands are executed (using shell built-in commands, or external commands).



Bash

- Introduction
- Scripting basics
- Q&A

Bash. Scripting. Summary (1)

When not to use shell scripts:

- Resource-intensive tasks, especially where speed is a factor (sorting, hashing, recursion, etc.)
- Procedures involving heavy-duty math operations, especially floating point arithmetic, arbitrary precision calculations, or complex numbers (use C++ or FORTRAN instead)
- Cross-platform portability required (use C or Java instead)
- Complex applications, where structured programming is a necessity (type-checking of variables, function prototypes, etc.)
- Mission-critical applications upon which you are betting the future of the company
- Situations where security is important, where you need to guarantee the integrity of your system and protect against intrusion, cracking, and vandalism
- Project consists of subcomponents with interlocking dependencies
- Extensive file operations required (Bash is limited to serial file access)

Bash. Scripting. Summary (2)

When not to use shell scripts:

- Need native support for multi-dimensional arrays
- Need data structures, such as linked lists or trees
- Need to generate / manipulate graphics or GUIs
- Need direct access to system hardware or external peripherals
- Need port or socket I/O
- Need to use libraries or interface with legacy code
- Proprietary, closed-source applications (Shell scripts put the source code right out in the open for all the world to see.)

P.S. If any of the above applies, consider a more powerful scripting language -- perhaps Perl, Python, Ruby - or possibly a compiled language such as C, C++, or Java.
Even then, prototyping the application as a shell script might still be a useful development step.

Bash. Scripting. Summary (3)

Shells like **bash** have support for programming constructs that can be saved as scripts.

These scripts in turn then become more shell commands. Many Linux commands are scripts.

User profile scripts are run when a user logs on and init scripts are run when a daemon is stopped or started.

This means that system administrators also need basic knowledge of scripting to understand how their servers and their applications are started, updated, upgraded, patched, maintained, configured and removed, and also to understand how a user environment is built.

The goal of this module is to give enough information to be able to read and understand scripts.
And to become a writer of simple scripts.

Bash. Scripting. Hello World

hello world

Just like in every programming course, we start with a simple ***hello_world*** script. The following script will output Hello World.

```
> echo Hello World
```

After creating this simple script in ***vi*** or with ***echo***, you'll have to ***chmod +x hello_world*** to make it executable. And unless you add the scripts directory to your path, you'll have to type the path to the script for the shell to be able to find it.

```
[student@localhost ~]$ echo echo Hello World > hello_world
[student@localhost ~]$ chmod +x hello_world
[student@localhost ~]$ ./hello_world
Hello World
[student@localhost ~]$
```

Bash. Scripting. She-bang

Let's expand our example a little further by putting ***#!/bin/bash*** on the first line of the script.
The ***#!*** is called a ***she-bang*** (sometimes called ***sha-bang***), where the ***she-bang*** is the first two characters of the script.

```
#!/bin/bash
echo Hello World
```

You can never be sure which shell a user is running. A script that works flawlessly in ***bash*** might not work in ***ksh, csh,*** or ***dash***. To instruct a shell to run your script in a certain shell, you can start your script with a she-bang followed by the shell it is supposed to run in. This script will run in a ***bash*** shell.

```
#!/bin/bash
echo -n hello
echo A bash subshell `echo -n hello`
```

Bash. Scripting. Comments. Variables

Let's expand our example a little further by adding comment lines.

```
#!/bin/bash
#
# Hello World Script
#
echo Hello World
```

Here is a simple example of a variable inside a script.

```
#!/bin/bash
#
# simple variable in script
#
var1=3
echo var1 = $var1
```

Bash. Scripting. Variables. Sourcing a script

Scripts can contain variables, but since scripts are run in their own shell, the variables do not survive the end of the script.

```
[student@localhost ~]$ ./simple_variable_in_script
var1 = 3
[student@localhost ~]$ echo $var1

[student@localhost ~]$
```

But we can force a script to run in the same shell, this is called ***sourcing*** a script (2 ways).

```
[student@localhost ~]$ source ./simple_variable_in_script
var1 = 3
[student@localhost ~]$ echo $var1
3
[student@localhost ~]$
```

```
[student@localhost ~]$ . ./simple_variable_in_script
var1 = 3
[student@localhost ~]$ echo $var1
3
[student@localhost ~]$
```

Bash. Scripting. Variables. Troubleshooting a script

Another way to run a script in a separate shell is by typing bash with the name of the script as a parameter.

```
[student@localhost ~]$ bash simple_variable_in_script
var1 = 4
```

Expanding this to bash -x allows you to see the commands that the shell is executing (after shell expansion).

```
[student@localhost ~]$ bash -x simple_variable_in_script
+ var1=4
+ echo var1 = 4
var1 = 4
[student@localhost ~]$ cat simple_variable_in_script
#!/bin/bash
#
# simple variable in script
#
var1=4
echo var1 = $var1
[student@localhost ~]$
```

Notice the absence of the commented (#) line, and the replacement of the variable before execution of echo.

Bash. Scripting. Conditions and loops. *test[]*

The ***test*** command can test whether something is true or false. Let's start by testing whether 10 is greater than 55.

```
$ test 10 -gt 55 ; echo $?
1
```

The test command returns ***1*** if the ***test*** fails. And as you see in the next screenshot, ***test*** returns 0 when a test succeeds.

```
$ test 56 -gt 55 ; echo $?
0
```

If you prefer true and false, then write the test like this.

```
$ test 56 -gt 55 && echo true || echo false
true
$ test 6 -gt 55 && echo true || echo false
false
```

The test command can also be written as square brackets, the screenshot below is identical to the one above.

```
$ [ 56 -gt 55 ] && echo true || echo false
true
$ [ 6 -gt 55 ] && echo true || echo false
false
```

Bash. Scripting. Conditions and loops. *test[]*

Below are some example tests. Take a look at ***man test*** to see more options for tests.

[-d foo]	***Does the directory foo exist ?***
[-e bar]	***Does the file bar exist ?***
['/etc' = $PWD]	***Is the string /etc equal to the variable $PWD ?***
[$1 != 'secret']	***Is the first parameter different from secret ?***
[55 -lt $bar]	***Is 55 less than the value of $bar ?***
[$foo -ge 1000]	***Is the value of $foo greater or equal to 1000 ?***
["abc" < $bar]	***Does abc sort before the value of $bar ?***
[-f foo]	***Is foo a regular file ?***
[-r bar]	***Is bar a readable file ?***
[foo -nt bar]	***Is file foo newer than file bar ?***
[-o nounset]	***Is the shell option nounset set ?***

Tests can be combined with logical AND and OR.

```
$ [ 66 -gt 55 -a 66 -lt 500 ] && echo true || echo false
true
$ [ 66 -gt 55 -a 660 -lt 500 ] && echo true || echo false
false
$ [ 66 -gt 55 -o 660 -lt 500 ] && echo true || echo false
true
```

Bash. Scripting. Conditions and loops. *If then else*

The ***if then else*** construction is about choice. If a certain condition is met, then execute something, else execute something else. The example below tests whether a file exists, and if the file exists then a proper message is echoed.

```
#!/bin/bash
if [ -f isit.txt ] then echo isit.txt exists!
else echo isit.txt not found!
fi
```

If we name the above script 'choice', then it executes like this.

```
$ ./choice
isit.txt not found!
$ touch isit.txt
$ ./choice isit.txt exists!
$
```

Bash. Scripting. Conditions and loops. *if then elif.*

You can nest a new ***if*** inside an ***else*** with ***elif***. This is a simple example.

```
#!/bin/bash
count=42
if [ $count -eq 42 ]
then
		echo "42 is correct."
elif [ $count -gt 42 ]
then
		echo "Too much."
else
		echo "Not enough."
fi
```

Bash. Scripting. Conditions and loops. *for loop.*

The example below shows the syntax of a classical ***for loop*** in bash:

```
for i in 1 2 4
do
        echo $i
done
```

An example of a for loop combined with an embedded shell:

```
#!/bin/bash
for counter in `seq 1 20`
do
        echo counting from 1 to 20, now at $counter
        sleep 1
done
```

Bash. Scripting. Conditions and loops. *for loop.*

The same example as above can be written without the embedded shell using the bash {from..to} shorthand.

```
#!/bin/bash
for counter in {1..20}
do
        echo counting from 1 to 20, now at $counter
        sleep 1
done
```

This for loop uses file globbing (from the shell expansion). Putting the instruction on the command line has identical functionality.

```
$ ls
count.ksh  go.ksh
$ for file in *.ksh ; do cp $file $file.backup ; done
$ ls
count.ksh  count.ksh.backup  go.ksh  go.ksh.backup
```

Bash. Scripting. Conditions and loops. *while loop.*

Below a simple example of a ***while loop***.

```
i=100;
while [ $i -ge 0 ] ;
do
        echo Counting down, from 100 to 0, now at $i;
        let i--;
done
```

Endless loops can be made with
while true or ***while :*** ,
where the ***colon*** is the equivalent of
no operation in the ***bash*** shell.

```
#!/bin/bash
# endless loop while :
do
        echo hello
        sleep 1
done
```

Below a simple example of an ***until loop***

```
let i=100;
until [ $i -le 0 ] ;
do
        echo Counting down, from 100 to 1, now at $i;
        let i--;
done
```

Bash. Scripting. Conditions and loops. Example.

Write a script that counts the number of files ending in ***.txt*** in the current directory

```
#!/bin/bash
let i=0
for file in *.txt
do
		let i++
done
echo "There are $i files ending in .txt"
```

Wrap an ***if*** statement around the script so it is also correct when there are zero files ending in ***.txt***

```
#!/bin/bash
ls *.txt > /dev/null 2>&1
if [ $? -ne 0 ]
then echo “Directory contains 0 *.txt files"
else
	let i=0
	for file in *.txt
	do
		let i++
	done
	echo " Directory contains $i *.txt files "
fi
```

Q&A

Thank you!