Alternative OS – Lecture 9

1. Shell types
2. A simple shell script
3. Basic elements of a shell script
4. **if** control structure
5. Conclusions

1. Shell types

What is a *shell*? A *shell* is a program that provides you with a user interface. With a shell you can type in commands and run other programs. In other words, a *shell* is a program that reads your commands, and executes them.

Every time you log in a *default shell* is started. It will read its configuration files, it may set up some *variables* and *search paths,* it may also run some commands, and do other things specified in those startup configuration files.

A *shell* is similar to MS-DOS command.com file. Another name for *shell* is a *command interpreter*.

There are many different types of *shells*:

- Bourne shell (**sh**)
- C shell (**csh**)
- Korn shell (**ksh**)
- Bourne Again Shell (**bash**)
- T-C shell (**tcsh**)
- The extended C Shell (**cshe**)
- The Plan 9 shell (**rc**)

You can choose your default shell when you configure your user account. You can also run other shells over your present shell, just like you run any other command.

The original Unix shell was the Bourne shell '**sh'.** In that time many of the Unix users were also **C** language programmers. That's why another shell very soon grew very popular. It was the *C Shell* written by Bill Joy at the University of California at Berkley. The *C Shell* sported a **C** language-like syntax, so comfortable for **C** programmers. Besides, it offered significant functionality improvement over the old good **sh**.

The most popular shells of the present time are **bash** (the improved version of the Bourne Shell) and **tcsh** (the improved version of **csh**).

You can always find out what shell you are running giving the command:

% **echo** $SHELL

2. A simple shell script

We've been working in a shell since we began learning Unix. What is so special about shell, when there is an attractive Windows-like environment?

- Well, when system crashes the graphical environment is usually not available.
- Another reason is that when connecting to a remote host over the Internet, either the connection speed does not permit usage of the graphical environment, or the *firewall* does not allow it.
- **However, maybe the main reason is that the *shell* in reality is a programming language! It can automate many routine tasks.**

We already learnt how to automate some tasks using *wildcards*. Now we shall learn a more powerful tool of automation – the *shell scripting,* as we call programming for a Unix shell.

Let us start with a very simple *shell script*. Call it **shinfo**

```
#!/bin/sh
echo "Your login shell is: $SHELL"
```

You can use an editor like **vi** to create the file **shinfo.** Then you make it executable for the *user* and the *group*:

% **chmod** ug+x *shinfo*

Now you can execute the file by typing:

% ./**shinfo**

Your login shell is: /bin/tcsh

We have just created our own Unix command called **shinfo** that displays the type of the used shell!

Writing shell scripts is simple. Let's look at the sample script above.

Every shell script must start with the *"shebang"* line: #!**shell_type.** As you remember there is a plenty of Unix shells. As a matter of fact there are over 30 of them. Most of those shells have different syntax. That is, a program written for one shell will not run under another shell. That is often the case for the two most popular shells: **bash** and **csh**. The *"shebang"* line is a workaround for this problem. It simply tells the system what shell must *interpret* our script. In the given example the script must be interpreted by the *Bourne Shell:* #!**/bin/sh**.

Examples:

The shell	**The "shbang"**
C Shell	#!**/bin/csh**
Bourne Shell	#!**/bin/sh**
Bourne Again Shell	#!**/bin/bash**

As a matter of fact our simple script will run perfectly well under most of the shells including **bash.**

The next line is a shell command:

echo "Your login shell is: *$SHELL*"

This command contains two elements:

- The **echo** command itself displays the argument line.
- The *argument* is a *quotation* that contains:
  - A constant string 'Your working shell is:'
  - And the variable replacement instruction '*$SHELL'*

The *shell* will work in following way. When the string in double quotes is encountered, all *variable replacement instructions* will be replaced by their

values. The '*$SHELL'* is a system variable that contains your *login shell* path. That's why the output of this command will be:

Your login shell is: /bin/tcsh

3. Basic elements of a shell script

a. The "shebang"

This is always the very first line of the script. It lets the *Kernel* know what shell will be interpreting this script.

b. Comments

Comment is a line starting with #, except for the "shbang". The shell does not interpret these lines. Those lines should help other humans who would like to read your script to understand it.

Example:

 # This is a comment
 # Use comments in your scripts!

c. Output display

We use **echo** command to display the output.

Remark: *Wildcards* must e escaped with a *backslash* \ or matching quotes.

Example:

echo *"Hello World!"*
echo *"I think, that means I exist!"*

d. Variables

Variables are used to store values that can be changed. All variables in shell programming have data type *strings*, and they don't have to be declared, unlike they have to be declared in *Pascal* or *C*.

Assigning a value to a variable:

Examples:

a="hello world"
x=2341

Remark: If a value contains spaces, like in *"hello world",* the line must be put between matching quotes.

Extracting the value from a variable:

A dollar sign is used to extrac .

Examples:

echo *$a*
echo *"The variable x has value $x"*

A simple shell script:

```
#!/bin/sh
# assign a value:
a="hello world"
# display the value of a
echo "I say: $a!"
```

Sometimes variable names can be confused with the rest of the text.

Example:

```
#!/bin/sh
var="bat"
#Correct:
echo I am a $var"man"
echo "I am a ${var}man"
```

```
echo "I am a $var\man"
#Incorrect:
echo "I am a $varman"
echo "I am a $var{man}"
echo I am a \$varman
echo I am a '$var'man
```

```
I am a batman
I am a batman
I am a batman
I am a
I am a bat{man}
I am a $varman
I am a $varman
```

Normally, the variables introdly when the script is running. We can use the keyword **export** to make our variable accessible even after the script is stopped.

Example:

VARIABLE_NAME=value
export VARIABLE_NAME

PATH=/bin:/usr/bin:.
export PATH

Variables available only *within the scope of the script* are called *local variables*. Variables available throughout the shell, like those we **export**, are called *global variables*.

e. User Input

The **read** command takes a line of input from the user and assigns it to one or more variables. The variable names are given on the right hand side of the **read** command. Each variable is assigned a separate word from the input line.

Example:

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

```
echo "Enter your full name:"
read a b c
echo "Your first name is $a"
echo "Your middle name is $b"
echo "Your  surname is $c"
```

4. if control structure

The **if** statement is used to test if the condition is *true* (exit status is 0, success), and execute either the" then" part or the" else" part.

Syntax:

```
if ....; then
   ....
elif ....; then
   ....
else
   ....
fi
```

The *semicolon* is used to separate the command that tests a condition from the ***then*** *operator*.

In the most cases a very special command is used to test statements. The command is called " test", and it is called with square brackets **[ ]**.

Examples:

[-f "somefile" **]**
Test if somefile is a file.

[-x "/bin/ls" **]**
Test if /bin/ls exists and is executable.

[-n "*$var*" **]**
Test if the variable $var contains something

["*$a*" **=** "*$b*" **]**
Test if the variables "$a" and "$b" are equal

A simple script:

```
#!/bin/sh
if [ "$SHELL" = "/bin/bash" ]; then
  echo "your login shell is the bash (Bourne again shell)"
else
  echo "your login shell is not bash but $SHELL"
fi
```

5. Conclusions

One of the main advantages of Shells is that they allow us automate routine tasks. Make the machine do boring job for us.

Shell scripting is a very simple programming language. But it also has its own syntax that we have to follow.

For example, we start our shell scripts always with the “shebang” line that tells the Kernel what shell must execute this script. Different shells are not always compatible!