OS

2nd Seminar

What is a SHELL?

a command line interpreter or an environment that provides an user interface in the command line for operating systems similar to Unix.
- interactive command language
- scripting language

It is used by the operating system to control system execution using shell scripts .
Users usually interact with a Unix shell using a terminal emulator; more than that, the direct operation through serial hardware connections or Secure Shell (ssh, see Putty) are common for server systems .

What is a SHELL?

All Unix shells provide:
- filename wildcarding
- piping
- here documents
- command substitution
- variables and control structures for condition testing and iteration

In the most generic sense of the term shell is any program that users use to type commands. A shell hides the details of the underlying operating system and handles the technical details of the operating system's kernel interface, which is the lowest level or "innermost" component of most operating systems.

Command files

A script (command file) is a text file that contains:

• Unix commands;

• directives (of the shell command interpreter) to control the execution flow of these commands.

A script behaves, in turn, like a shell command. The name of a script file does not have to respect any syntactic requirements!

We will add the .sh suffix to the script names as a convention of our own to illustrate the content.

Command files

Any shell command can be run :
1. Directly to the prompter in command line
2. The command is written in a script (file) to be run within the script execution
If “script” is the name of a command file in the current directory, it can be run:
1. ./script ... or absolutepath /script ... if the script file has execute rights . For setting rights, in particular those for execution, is used command chmod.
2. script ... if the script has execute rights and the current folder is in PATH
3. sh script ... or sh absolute path /script ... regardless it has or not execution rights.

By ... we denote: arguments , options , files , expressions , redirects : < > >> <& >&

SHELL Programming - Introduction

- Why do we write scripts?

- Speed relative to the C language

SHELL Programming - Intro

```
$ echo '#!/bin/sh' > my-script.sh

$ echo 'echo Hello World' >> my-script.sh

$ chmod 755 my-script.sh

$ ./my-script.sh

Hello World

$
```

SHELL Programming - Intro

```
#!/bin/bash

# This is a comment

echo Hello World                    # This is also a comment

------------------------------------------------------ --------

#!/bin/sh

# This is a comment

echo "Hello World              " # This is also a comment
```

SHELL Programming - Variables

Symbolic name for a memory area

We can assign values, we can read or modify the content

Declaration through assignment: `VAR= value` works ; `VAR = value` does not work

The value of the variable: `echo $VAR`

readonly variables: `VAR=“MyVar”`; readonly VAR

" Deleting" the variable: unset VAR

Readonly variables cannot be "deleted" (until shell process is terminated)

SHELL Programming - Variables

Types of variables:

- **Local variables** − present in the current shell instance. They are not available for programs that are started by the shell. These are set on the command line
- **Environment Variables** − An environment variable is available to any child shell process. Some programs need environment variables to function properly. Typically, a shell script defines only those environment variables that are required by the programs it runs.
- **Shell Variables** − A shell variable is a special variable that is set by the shell and is required by the shell to function properly. Some of these variables are environment variables, while others are local variables.

SHELL Programming - Variables

Special variables

$0 – the name of the command that is being run

$1 - $9 – command line arguments (in some environments ${10} also works)

$* or $@ - all arguments

the special parameter "$ *" takes the entire list as an argument with spaces in between, and the special parameter "$ @" takes the entire list and separates it into separate arguments.

$# - Number of arguments on the command line

$? – the return code of the previous command

SHELL Programming - Variables

```
#!/bin/sh

echo Who are you?

read ME

echo "Hello $ME, nice to meet you"
```

SHELL Programming - Variables

- Scope of variables
- Export

```
$ VAR=hi

$ ./script.sh

$ export VAR

$ ./script.sh
```

SHELL Programming - Variable Substitution

- Variable substitution allows the programmer to manipulate the value of a variable based on its state.
- Possible replacements:
  - **${var}** Replaces the value of *var* .
  - **${var:-word}** If *var* is null or unset, *word* replaces **var .** The *var* value does not change.
  - **${var:=word}** If *var* is null or unset, *var* is set to the value **word** .
  - **${var:?message}** If *var* is null or unset, *message* is displayed on standard error. It is useful to check the correct setting of variables.
  - **${var:+word}** If *var* is set, *word* replaces var. The value of *var* does not change.

SHELL Programming - Array Variables

Special parameter "$ *" takes entire list as an argument with spaces and the special parameter "$ @" takes entire list and separates it in separate arguments

```
array[index]=value

VAR_ARRAY[0]=1

VAR_ARRAY[1]=3
```

For accessing values:

```
${array[index]}

${array[*]} or ${array[@]}
```

SHELL Programming - Operators

Several types of operators are supported:

- Arithmetic: +, -, *, /, %, =, ==, != ` **expr** $a + $b`

  expr is an external program; There must be space between operators and expressions; the full expression must be enclosed between ` ` (backticks)

- Relational: (numbers) -eq, -ne, -gt, -lt, -ge, -le
- Boolean: !, -o (OR), -a (AND)
- Strings: = (equality), !=, -z (zero size), -n (non-zero size), str (empty)
- Test on files: -d, -f, -r, -w, -x, -s (size>0), -e (exists, file or folder), -p, -b, -c,

SHELL Programming - Exercises

Count all lines of code in C files in the directory given as a command line argument, excluding lines that are empty or contain only blank spaces:

```
#!/bin/bash

S = 0
for f in $1/*.c; do
        N=`grep "[^ \t]" $f | wc –l`
        S=`expr $S + $N`
done
echo $S
```

!!! Pay attention to file names that contain spaces

SHELL Programming

Filename wildcards: similar but simpler than regular expressions

Rules:

* - any sequence of characters, including the empty sequence, but not the first dot in the filename

? - any character (1 alone) but not the first dot in the filename

[abc] – List of character options, supports ranges like in regular expressions

[!abc]- Negate list of character options (similar to [^abc] in regex)

Example: list of files whose names start with a letter and have an extension of exactly two characters `ls [a-zA-Z]*.??`

SHELL Programming

What is the result of executing the following code:

```
$cd /

$ ls -ld {,usr,usr/local}/{bin,sbin,lib}
```

SHELL Programming - Decision

- if - fi

```
if [ expression ]
then
   Statements to be executed when expression is true
fi
```

- if - else - fi

```
if [ expression ]
then
   Statements to be executed when expression is true
else
   Statements to be executed when expression is false
fi
```

- if - elif - else - fi

```
if [ expression 1 ]
then
   Statements to be executed if expression 1 is true
elif [ expression 2 ]
then
   Statements to be executed if expression 2 is true
else
   Statements to be executed if no expression is true
fi
```

SHELL Programming - Decision

Similar to multiple if - elif statements

```
case word in
   pattern1)
                Statements to be executed if pattern1 matches
                ;;
   pattern2)
                Statements to be executed if pattern2 matches
                ;;
   *)
                Default condition to be executed
                ;;
esac
```

SHELL Programming - Loops

- while loop: executes the given commands until the given condition remains true
- for loop
- until loop: executes until the condition becomes true
- select loop

- break: terminates the execution of the entire loop
- continue: exits the current loop but not the entire loop

break n **continue n**

SHELL Programming - Exercises

Count all lines of code in C files in the directory given as a command line argument and its subdirectories, excluding lines that are empty or contain only blank spaces

```
#!/bin/bash

S = 0
for f in `find $1 –type f –name “*.c”; do
            N=`grep "[^ \t]" $f | wc –l`
            S=`expr $S + $N`
done
echo $S
```

SHELL Programming - Conditions

What does **test** mean/do ?

To make the writing of the condition look a bit more natural, there is a second syntax, where [is an alias of the command for the test and] marks the end of the test. Be careful, leave spaces around these square brackets or there will be syntax errors.
The basic IF example in the presentation can be rewritten as follows:

```
for A in $@; do
        if [ -f $A ]; then
        echo $A is a file
        elif [ -d $A ]
        then
        echo $A is a dir
        elif echo $A | grep -q "^[0-9]\+$";
then
        echo $A is a number
        else
        echo We do not know what $A is
        fi
done
```

SHELL Programming - Exercises

Read console input until the user supplies a file name that exists and can be read

```
#!/bin/bash

F=””
while [ -z "$F" ] || [ ! -f "$F" ] || [ ! -r "$F" ]; do
		read -p "Provide an existing and readable file path:" F
done
```

```
#!/bin/bash

F=””
while test -z "$F" || ! test -f "$F" || ! test -r "$F"; do
		read -p "Provide an existing and readable file path:" F
done
```

SHELL Programming - Exercises

Write a script that monitors the state of a directory and prints a notification when something has changed

```
#!/bin/bash
D=$1
if [ -z "$D" ]; then
  echo "ERR: No directory for monitoring" >&2
  exit 1
fi
if [ ! -d "$D" ]; then
  echo "ERROR: Directory $D does not exist" >&2
  exit 1
fi
STATE=""
while true; do
		S=""
		for P in `find $D`; do
		if [ -f $P ]; then
				LS=`ls -l $P | sha1sum`
				CONTENT=`sha1sum $P`
			else
				LS=`ls -l –d $P | sha1sum`
				CONTENT =`ls -l $P | sha1sum`
			fi
			S="$S\n$LS $CONTENT"
		done
		if [ -n "$STATE" ] && [ "$S" != "$STATE" ]; then
			echo "Directory state changed"
		fi
		STATE=$S
		sleep 1
```