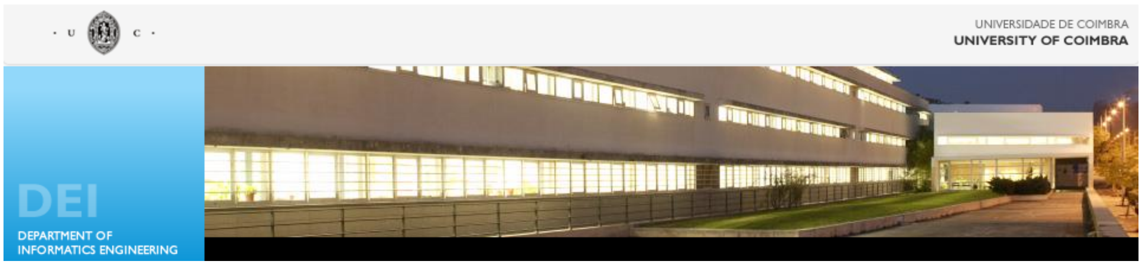

Operating Systems [2021-2022]

Tutorial - Shell Scripting

__Note__: this tutorial will __not__ be done during the practical classes, although questions will be clarified by the teacher.

Introduction

Shell Scripting can be seen as an alternative way to interact with the operating system through the shell. Instead of inserting the commands one by one, these can be written into scripts and interpreted sequentially. Moreover, these scripts support common features found on programming languages, like variables and control instructions. This tutorial presents the basic concepts and syntax of Shell Scripting. Since different shells can present some differences regarding the syntax, this tutorial is focused on the **bash** shell.

Concepts and Syntax

Shell scripts are often files with the extension *.sh*, and to be interpreted these files need to have executable and readable permissions. These permissions can be added using the **chmod** command, like: `chmod a+rx <script file>`.

The first line of any script is a special comment that indicates its target shell. This is required because different shells may present differences regarding the syntax. For the bash shell, this first line must be `#!/bin/bash`. The **#** symbol can be used afterwards to insert comments in any line.

Variables can be useful to retain information that is necessary in a later stage of the script. To define them the following syntax is used: `<variable>=<value>`. Notice that there are no spaces before and after the =, otherwise the shell will interpret the variable name as a command. Another important aspect is that the type of the variable is not defined. The shell stores all variables as strings but distinguishes text from numeric data. In order to access a variable its name must be proceeded by a **$**, like: ***$***`<variable>`. This symbol is also used to access information about the arguments passed to the script, namely:

- `$0 to $9` – Access each argument individually;
- `$*` – Access all arguments;
- `$#` – Get the total number of arguments.

The **if** statement is often used in the scripts, and has the following syntax:

```
if <condition> then
      <…>
elif <condition> then
      <…>
else
      <…>
fi
```

Regarding the condition, the following types are accepted:

- Comparisons with numeric variables, strings or files, which are defined inside **[…]**. This type can also be used in any part of the script through the **test** command;
- Execution of commands in a subshell, where the condition is based on the exit code of the commands considering zero as true, which are defined inside **(…)**. One can define the commands without using the (), but in this case they are not executed in a subshell;
- Arithmetic operations, which are defined inside **((…))**, considering a non-zero result as true. This type is not standard, being only supported by some shells, including the bash;
- More complex comparisons based on the ones mentioned on the 1st point, like checking if a string matches a regular expression, which are defined inside **[[…]]**. Once again, this type is not standard, but it is supported by the bash.

Case statements are also supported, with the following syntax:

```
case <variable> in
      <value1>) <…>
                ;;
      <value2>) <…>
                ;;
      <valueN>) <…>
                ;;
             *) <…>
                ;;
esac
```

Notice that the ***)** option is the default scenario.

Regarding the loops, Shell Scripting supports **while** and **for**. The while has the following syntax:

```
while <condition>
do
      <…>
done
```

The condition described above supports the same types accepted by the if statement. The for loop has two different syntaxes:

```
for <variable> in <list>
do
      <…>
done
```

```
for ((exp1; exp2; exp3))
do
      <…>
done
```

The left syntax iterates over a list of values, and this list can mix different types of data. The right syntax is the "traditional" one, incrementing a variable between a start and end value.

In order to perform the comparisons with numeric variables, strings and files, different operators are required. Regarding the numeric types, the operator is defined between two values or variables, like: `<num1> <op> <num2>`. The available ones are:

- **-eq** – equal
- **-ne** – not equal
- **-lt** – less than
- **-le** – less than or equal
- **-gt** – greater than
- **-ge** – greater than or equal

The "traditional" operators (e.g., < or >) can only be used inside **((…))**. Although the bash supports this usage, it is not standard.

Regarding the strings, the operator is defined in the same way as for numeric types, and the following ones are available:

- **=** – equal
- **!=** – not equal
- **-n** – string is not null
- **-z** – string is null

The file operators are defined before the path to a specific file, like: `<op> <file>`. Some of the available operators are:

- **-e** – exists
- **-f** – is a regular file (not a directory)
- **-d** – is a directory
- **-r** – has read permissions
- **-w** – has write permissions
- **-x** – has execute permissions

Executing commands in subshells (child processes of the current one) is often used to obtain their output on the script like the return of a function. This can be done by defining the commands inside **$(…)**. For example, the output can be stored in a variable like this: `<variable>=$(<commands>)`. Subshells are also useful for isolation and parallel execution purposes.

Shell Scripting also offers some commands that allow interaction with the user. In order to write something on the shell the **echo** command is used. It receives a string and displays it, and if this string contains variables previously defined their values are presented. For example, assuming the previous definition of the variable *NAME* (`NAME="DEI"`), the execution of the line `echo "The name is $NAME"` should write to the console the sentence "The name is DEI". Moreover, echo also supports backslash-escaped characters (if the **-e** flag is used) and all the redirection operators. In order to ask information through the shell, the **read** command can be used, followed by the name of the variable that is going to store the inputted string, like: *read <variable>*. This variable does not need to be previously declared.

Exercise 1

Create a shell script that receives two numbers as arguments, sums them and displays a sentence saying if the resulting value is a positive or negative number.

a) Create a file named "ShellScriptExercise.sh" and give it readable and executable permissions using the following command: ***chmod a+rx ShellScriptExercise.sh***

b) Start by indicating in the first line that the target shell is the bash. Also save the received arguments into new variables, just to improve the code legibility.

```
#!/bin/bash

NUMBER1=$1

NUMBER2=$2

echo "$NUMBER1 + $NUMBER2 = ???"
```

c) Implement the sum operation and display it.

```
NUMBER3=$(($NUMBER1+$NUMBER2))

echo "$NUMBER1 + $NUMBER2 = $NUMBER3"
```

d) Evaluate if the resulting value is positive or negative and present the conclusion with a sentence.

```
if (( $NUMBER3 > 0 ))
then
   echo "$NUMBER3 is a Positive Number!"
else
   echo "$NUMBER3 is a Negative Number!"
fi
```

Exercise 2

Create a shell script that displays periodically the CPU and memory usage of a specific running process. The timestamp associated with the values should also be displayed. The information must be written simultaneously to a CSV file. The script receives as arguments the PID of the running process and the name of the file where the data is going to be saved. Before starting the data collection, the script must verify if the given PID is valid and if the CSV file exists. If the file does not exist it must be created with the proper header, and if it exists the user must be able to decide if he wants to clear its contents. The script should run until the process being watched ends, and the data collection should be executed every 5 seconds.

a) Start by doing the operations described on items a) and b) of the previous exercise (adapt them for this one).

b) Verify if the received PID is valid by checking if the process is running. This validation can be done using the command **ps -p <pid>**, because it returns 0 as exit code if the process exists. Moreover, the output of the command can be redirected to */dev/null*, which will avoid it being displayed in the shell.

```
if ( ! ps -p $PID > /dev/null )
then
   echo -e "\nProcess $PID is not running!"
   exit 1
else
   echo -e "\nProcess $PID is running!"
fi
```

c) Check if the received CSV file exists and is a regular file, using the **-f** operator. Create a new file, with the proper header, if it does not exist or if the user chooses explicitly to clear the contents of the existing one.

```
CREATE_LOG=0

if [ ! -f $LOG_FILE ]
then
   CREATE_LOG=1
else
   echo -e "\nClear the log file? (y/n)"
   read CLEAN

   if [ $CLEAN = "y" ] || [ CLEAN = "Y" ]
   then
      CREATE_LOG=1
   fi
fi

if [ $CREATE_LOG -eq 1 ]
then
   echo "TIMESTAMP,CPU,MEMORY" > $LOG_FILE
   echo -e "\n$LOG_FILE was created."
else
   echo -e "\n$LOG_FILE already exists."
fi
```

d) Define the data collection instructions inside a while loop that is active until the watched process ends. To obtain the current date and time of the system the command **date +"%Y-%m-%d %T")** can be used. The CPU and memory usage can be obtained with the **ps** command using the **-o** flag to define the required fields: **%cpu=** for the CPU usage and **%mem=** for the memory usage (the **=** is only to discard the column header). In order to ensure the 5 seconds interval between each collection the **sleep** command may be used.

```
echo -e "\nCollecting CPU and Memory usage for process $PID...\n"

while ( ps -p $PID > /dev/null )
do
   TIMESTAMP=$(date +"%Y-%m-%d %T")
   CPU=$(ps -p $PID -o %cpu=)
   MEM=$(ps -p $PID -o %mem=)

   echo "$TIMESTAMP,$CPU,$MEM" >> $LOG_FILE

   echo "Timestamp: $TIMESTAMP"
   echo "CPU: $CPU%"
   echo -e "Memory: $MEM%\n"

   sleep 5s
done
```

e) The complete code of the script is available next.

```
#!/bin/bash

PID=$1

LOG_FILE=$2

if ( ! ps -p $PID > /dev/null )
then
   echo -e "\nProcess $PID is not running!"
   exit 1
else
   echo -e "\nProcess $PID is running!"
fi

CREATE_LOG=0

if [ ! -f $LOG_FILE ]
then
   CREATE_LOG=1
else
   echo -e "\nClear the log file? (y/n)"
   read CLEAN

   if [ $CLEAN = "y" ] || [ CLEAN = "Y" ]
   then
      CREATE_LOG=1
   fi
fi

if [ $CREATE_LOG -eq 1 ]
then
   echo "TIMESTAMP,CPU,MEMORY" > $LOG_FILE
   echo -e "\n$LOG_FILE was created."
else
   echo -e "\n$LOG_FILE already exists."
fi

echo -e "\nCollecting CPU and Memory usage for process $PID...\n"

while ( ps -p $PID > /dev/null )
do
   TIMESTAMP=$(date +"%Y-%m-%d %T")
   CPU=$(ps -p $PID -o %cpu=)
   MEM=$(ps -p $PID -o %mem=)

   echo "$TIMESTAMP,$CPU,$MEM" >> $LOG_FILE

   echo "Timestamp: $TIMESTAMP"
   echo "CPU: $CPU%"
   echo -e "Memory: $MEM%\n"

   sleep 5s
done
```