Sistemas Operativos
Introducción al bash

GUIÓN SOBRE BOURNE SHELL (bash) – Programación de scripts

Primer script: Hello Wold

1) El primer paso va a ser localizar nuestro interprete Bash en el sistema de ficheros. Para ello teclea el siguiente comando:

```
$ which bash
```

La salida de este comando la utilizaremos en la primera línea de todos nuestros scripts para indicar el intérprete que tiene que ejecutar dicho script. Dicha línea comenzará con **#!** que en este caso no se interpretará como un comentario. Edita un fichero llamado *hello_world.sh* y copia lo siguiente, sustituyendo en la primera línea */bin/bash* por el resultado que has obtenido del comando ejecutado anteriormente (*which bash*).

```
#!/bin/bash
# declarar variable STRING
STRING="Hello World"
#imprimir variable por pantalla
echo $STRING
```

Una vez editado y guardado el fichero, dale permisos de ejecución:

```
$ chmod +x hello_world.sh
```

Ejecuta el script (*./hello_world.sh*) y comprueba que el resultado es correcto (que imprime por pantalla *Hello World*).

Sentencia IF

2) Vamos a escribir un script que chequee si lo que se le pasa como parámetro es o no un directorio. Edita un fichero llamado *ejemploif.sh* con el siguiente contenido:

```
#!/bin/bash
if [ -d $1 ]
then
        echo $1 es un directorio
else
        echo $1 no es un directorio
```

fi

Ejecuta el script pasándole como parámetro el nombre de distintos directorios y ficheros y comprueba su funcionamiento (*./ejemploif.sh nombredirectorio*).
Vamos a explicar un par de cosas sobre el script anterior.

- *$1* es una variable que contiene el primer parámetro pasado a un script (si sustituimos el 1 por otro número accederemos al correspondiente parámetro, consulta apartado *Shell Built-in Variables* del documento *Bash Cheat Sheet* adjunto).
- Las condiciones de las sentencias "if" y los "bucles" se expresan con [ ] o con el comando *test*. Observa que hay un espacio después del "[" y antes de "]", **estos espacios son muy importantes, sin ellos no funcionará el test**. Para conocer los tipos de test que se pueden aplicar consulta los apartados *Checking files, Checking strings, Checking numbers* y *Boolean operators* del documento *Bash Cheat Sheet.*

3) Escribe un script llamado *isexec.sh* que dado un parámetro de entrada se comporte de la siguiente manera (suponemos que se ejecuta el script así: *./isexec.sh parametro1*)

- Si el parámetro de entrada se corresponde con un fichero (no directorio) que existe y que tiene permisos de ejecución para el usuario debe imprimir por pantalla el texto:
  - *parametro1 es un fichero ejecutable*
- En caso contrario (que no sea un fichero, que no exista o que no tenga permisos de ejecución para el usuario) debe imprimir:
  - *parametro1 no es un fichero ejecutable*

Para realizar este script no olvides consultar los apartados mencionados en el punto 2) del documento *Bash Cheat Sheet.* **Entrega este script en la correspondiente tarea del campus virtual de la asignatura**.

Bucle FOR

4) La sintaxis general de un bucle for es la siguiete:

```
for variable in lista
do
        órdenes
        …
done
```

donde lista puede ser algo tan variado como un conjunto de cadenas de caracteres, el conjunto de archivos de un directorio o la salida de un comando.

A continuación, presentamos una serie de ejemplos:

- *for i in 1 dos 3 cuatro*
  - la variable *i* (que se accede a su valor con *$i*) tomará en cada iteración los valores "1", "dos", "3" y "cuatro" respectivamente.

- *for i in **

    - aquí * representa todos los ficheros del directorio donde se ejecute el script, por lo que la variable *i* tomará sucesivamente como valor el nombre de cada archivo del directorio.
- *for i in `ls –R`*
    - ahora la variable *i* tomará como valores los proporcionados por la ejecución de un comando, el introducido entre los acentos graves. En este caso concreto, tomará los nombres de los ficheros que cuelgan del directorio desde donde se ejecuta el script, entrando recursivamente en cada subdirectorio (modificador –R).

Consulta el apartado *Loops* del documento *Bash Cheat Sheet* para más información.

5) Escribe un script llamado *ejemplofor.sh* que recorra todas las entradas (ficheros y directorios) del directorio donde se ejecute el script y por cada entrada imprima el nombre de la entrada (nombre del fichero o directorio) e indique si es un fichero o un directorio. **Entrega este script en la correspondiente tarea del campus virtual de la asignatura**.

Asignación de valores a variables

6) En Shell script no hay tipos de variables, éstas son definidas en el momento que son asignadas, por ejemplo:

```
A=1
B=ANA
```

También podemos asignarles la salida de la ejecución de un comando:

```
A=`comando`
A=$(comando)
A=`ls | wc -w` #A tomará como valor el número de archivos del directorio actual
```

La asignación de variables es alfanumérica y por consiguiente se pueden concatenar variables muy fácilmente:

```
A=Ana
B=Practica1
C=$A/$B          #C toma el valor “ANA/Practica1”
```

Para practicar estos conceptos, copia el siguiente script en un fichero de nombre *filedircount.sh*:

```
#!/bin/bash
for i in *                  # Para cada entrada "i" del directorio donde se ejecuta el script
do
        if [ -d "$i" ]      # Chequea si dicha entrada "$i" es un directorio
        then                # En caso de serlo
                FILES=`ls -l "$i" | wc -l`  # Lista el contenido del directorio "$i" y
                                            # cuenta el número de líneas
```

```
            FILES2=`expr $FILES - 1` # Le restamos uno puesto que una de las líneas es el
                                     # número de bloques que ocupa el directorio
        # si el directorio esta vacío wc da como resultado 0 y al restarle 1 FILES2 valdrá -1
        # en este caso, reasignamos FILES2 a su valor correcto que es 0
            if [ $FILES2 -eq -1 ]
            then
                FILES2=0
            fi
            echo "$i: $FILES2"      # Muestra el nombre del directorio y el número de
                                     # entradas que hay en él
    fi
done
```

Ejecuta el script y comprueba que devuelve, para cada subdirectorio que hay en el directorio donde se ejecute el script, el número de entradas que hay en dicho subdirectorio. Repasa el script línea a línea para comprender que se hace en cada línea y como se hace. Si tienes cualquier duda consulta a tu profesor.

Observa que la variable $FILES contiene el número de entradas del directorio obtenido de la ejecución de un comando. Para poder restarle 1 a dicha variable, lo tenemos que hacer de forma "numérica" y no "alfanumérica" por ese motivo tenemos que usar el comando *expr* (consulta el apartado *Shell Arithmetic* del documento *Bash Cheat Sheet* para más información).

7) Tomando como ejemplo el script *filedircount.sh* anterior, realiza un script denominado *filedircount2.sh* que haga exactamente lo mismo que éste, pero que acepte un parámetro de entrada que sea el directorio el cual analizar (y no el directorio actual como hace *filedircount.sh*).
Su utilización será:

./filedircount2.sh /path/hasta/el/directorio

El script tomará el primer parámetro (en este caso */path/hasta/el/directorio*), comprobará que es un directorio (en caso contrario informará del problema y terminará) y realizará el mismo procesamiento que hace *filedircount.sh* pero con el directorio especificado. **Entrega este script en la correspondiente tarea del campus virtual de la asignatura**.

8) Copia el siguiente script en un fichero llamado *backup.sh*:

```
#!/bin/bash
if [ ! -f "$1" ]    # Chequea si el parámetro de entrada representa un fichero que existe
then                # En caso de no existir, da un mensaje comunicándolo
    echo $1 no existe
else                # En caso de existir el fichero ...
    A=$(ls $1* | wc -w)     # Lista los ficheros que comienzan por el nombre dado como
                            # parámetro y cuenta cuantos hay y lo almacena en A
    if [ $A -ge 9 ]         # Si hay 9 o más, da un mensaje indicando que se ha
                            # alcanzado el máximo número de versiones
```

```
        then
            echo "Se ha superado el número máximo de versiones"
        else                    # Si no se ha alcanzado el maximo ...
            Num=`expr $A + 1`   # Se suma 1 al número de ficheros que comienzan
                                # con ese nombre (A) para que sea la terminación
                                # de la nueva copia
            Version=$1.$Num     # se compone el nombre del nuevo nombre de
                                # fichero (Version) como el nombre original ($1)
                                # seguido por "." y el número actual de copia (Num)
            cp $1 $Version      # Copia el fichero original ($1) con el nombre nuevo
        fi
    fi
```

Este script va a ir guardando hasta 9 versiones de un fichero que se le pase como parámetro de entrada, de tal manera que en el nombre de las versiones aparezca el número de versión a la que corresponde. Repasa el script línea a línea para comprender que se hace en dicha línea y como se hace. Si tienes cualquier duda consulta a tu profesor.
Ejecuta el script varias veces para comprobar que funciona correctamente (según las especificaciones dadas).

9) Tomando como ejemplo el script *backup.sh* anterior (que hace hasta 9 copias de un fichero dado, nombrando cada copia con su número de orden de copia), realiza un script *backup2.sh* que dada una lista de ficheros como parámetro de entrada (no solo uno), cree una copia de cada fichero añadiéndole al comienzo del nombre del fichero copia, la fecha de realización de la copia en formato YYMMDD_ (siendo YY los dos últimos dígitos del año, MM dos dígitos del mes y DD dos dígitos del día. Para poder implementar esto mira el funcionamiento del comando *date* con *man*). **Entrega este script en la correspondiente tarea del campus virtual de la asignatura**.
Su utilización será:

*./backup2.sh *.c *.cpp *.h *.hpp*

El script tomará la lista de ficheros dada y uno a uno los copiará anteponiendo la fecha.
Si en el directorio dónde se ejecuta el comando anterior estuvieran los siguientes ficheros:

main.cpp
soporte.cpp
soporte.h

tras su ejecución debería aparecer (suponiendo la fecha de ejecución 10 de marzo de 2014):

140310_main.cpp
140310_soporte.cpp
140310_soporte.h
main.cpp
soporte.cpp
soporte.h

10) Realiza una mejora del script anterior llamada *bakcup3.sh*, de forma que las copias de seguridad se almacenaran en un directorio backup (si no existe créalo automáticamente la primera vez) para que no se mezclen con los ficheros originales. Dentro del directorio backup se debe crear un directorio por cada copia de seguridad que se haga (si no existe), es decir, por cada vez que se ejecute el script (con nombre igual al formato de fecha YYMMDD) y dentro de ese directorio hacer la copia de los ficheros fuente, ahora sin cambiarles el nombre. Con estas modificaciones, el resultado (en forma de árbol de directorios) sería algo así:

dir_de_trabajo/
|---- *main.cpp*
|---- *soporte.cpp*
|---- *soporte.h*
|---- *backup/*
| |---- *140310/*
| | |---- *main.cpp*
| | |---- *soporte.cpp*
| | |---- *soporte.h*
| |---- *140311/*
| | |---- *main.cpp*
| | |---- *soporte.cpp*
| | |---- *soporte.h*

Observa que le ejecución de este script dos veces en el mismo día no generaría dos copias de seguridad, sino que la segunda sobrescribiría la primera. Puedes pensar en una solución para mantener todas las copias (incluso varias el mismo día), pero manteniendo una ordenación de copias por directorios. **Entrega este script en la correspondiente tarea del campus virtual de la asignatura**.

11) (Opcional) Realiza un script llamado *clasifica.sh* que, dados como parámetros de entrada un directorio con ficheros de música en formato MP3 y otro directorio para almacenar la salida (outputdir), clasifique por artistas y renombre los ficheros MP3 con el nombre de la canción (outputdir podrá o no existir, si no existe el script deberá crearlo). En outputdir se deberá crear (si no existe) un directorio por cada autor y dentro de dicho directorio de deberán copiar todas las canciones de dicho autor renombradas con el título de la canción. El autor de la canción aparece en una línea del fichero MP3 precedido por la cadena “autor:”, y el título de la canción aparece en otra línea del fichero MP3 precedido por la cadena “titulo:”.

Para obtener el autor y el título a partir del fichero MP3 se recomienda usar los comandos *grep* y *cut* (consulta en *man* el funcionamiento de ambos o pregunta a tu profesor).

Supongamos este contenido en el directorio de trabajo:

dir_de_trabajo/
|---- *canciones/*
| |---- *001.mp3*
| |---- *002.mp3*
| |---- *003.mp3*
| |---- *004.mp3*

dentro de los ficheros MP3 deberán aparecer dos líneas como, por ejemplo, las siguientes:

```
…
autor:Iron_Maiden
…
titulo:The_nunmber_of_the_beast
…
```

si ejecutamos el script en el directorio de trabajo de la siguiente forma:

```
$ ./clasifica.sh canciones outputdir
```

el resultado final debería ser algo parecido a esto (si esos fueran los autores y títulos de las canciones del directorio):

dir_de_trabajo/
|---- canciones/
| |---- 001.mp3
| |---- 002.mp3
| |---- 003.mp3
| |---- 004.mp3
|---- outputdir/
| |---- ACDC/
| | |---- Thunderstruck.mp3
| |---- Iron_Maiden/
| | |---- The_nunmber_of_the_beast.mp3
| |---- Metallica/
| | |---- Nothing_else_matters.mp3
| | |---- The_unforgiven.mp3

Para probar puedes descargarte el fichero *canciones.zip* del campus virtual (en la tarea de entrega), que contiene una carpeta "canciones" con ocho ficheros MP3 que cumplen las características especificadas en el enunciado.

Si realizas este script, entrégalo en la correspondiente tarea del campus virtual de la asignatura.

11) (Opcional) Realiza un script llamado *findbigfiles.sh* que, dados como parámetros de entrada un directorio y un número de bytes, muestre por pantalla todos los ficheros que cuelgan del directorio dado como parámetro y sus subdirectorios (recursivamente) cuyo tamaño en bytes sea mayor que el especificado en el segundo parámetro, además de indicar el tamaño (en bytes) de cada uno.

Un ejemplo de ejecución sería:

```
$ ./findbigfiles.sh midir 1024
```

siendo la salida similar a lo siguiente:

```
midir/fichero1.ext: 1500 bytes
midir/fichero2.ext: 2500 bytes
midir/misubdir1/fichero3.ext: 1250 bytes
midir/misubdir2/misubdir3/fichero4.ext: 150000 bytes
```

Para la implementación de este script te pueden ser de mucha utilidad los comandos *find* y *stat*. Estudia el uso de dichos comandos con *man* y mira si te pueden ser de utilidad. **Si realizas este script, entrégalo en la correspondiente tarea del campus virtual de la asignatura**.

Bash Cheat Sheet

(http://cis.stvincent.edu/html/tutorials/unix/bshellref)

This file contains short tables of commonly used items in this shell. In most cases the information applies to both the Bourne shell (sh) and the newer bash shell.

Tests (for ifs and loops) are done with [ ] or with the test command.

Checking files:

```
-r file     Check if file is readable.
-w file     Check if file is writable.
-x file     Check if we have execute access to file.
-f file     Check if file is an ordinary file (as opposed to a directory, a device
special file, etc.)
-s file     Check if file has size greater than 0.
-d file     Check if file is a directory.
-e file     Check if file exists.  Is true even if file is a directory.
```

Example:

```
if [ -s file ]
then
   #such and such
fi
```

Checking strings:

```
s1 = s2     Check if s1 equals s2.
s1 != s2    Check if s1 is not equal to s2.
-z s1       Check if s1 has size 0.
-n s1       Check if s2 has nonzero size.
s1          Check if s1 is not the empty string.
```

Example:

```
if [ $myvar = "hello" ]
then
   echo "We have a match"
fi
```

Checking numbers:

Note that a shell variable could contain a string that represents a number. If you want to check the numerical value use one of the following:

```
n1 -eq n2      Check to see if n1 equals n2.
n1 -ne n2      Check to see if n1 is not equal to n2.
n1 -lt n2      Check to see if n1 < n2.
n1 -le n2      Check to see if n1 <= n2.
n1 -gt n2      Check to see if n1 > n2.
n1 -ge n2      Check to see if n1 >= n2.
```

Example:

```
if [ $# -gt 1 ]
then
   echo "ERROR: should have 0 or 1 command-line parameters"
fi
```

Boolean operators:

```
!     not
-a    and
-o    or
```

Example:

```
if [ $num -lt 10 -o $num -gt 100 ]
then
   echo "Number $num is out of range"
elif [ ! -w $filename ]
then
   echo "Cannot write to $filename"
fi
```

Note that ifs can be nested. For example:

```
if [ $myvar = "y" ]
then
   echo "Enter count of number of items"
   read num
   if [ $num -le 0 ]
   then
      echo "Invalid count of $num was given"
   else
      ... do whatever ...
   fi
fi
```

The above example also illustrates the use of read to read a string from the keyboard and place it into a shell variable. Also note that most UNIX commands return a true (nonzero) or false (0) in the shell variable status to indicate whether they succeeded or not. This return value can be checked. At the command line echo $status. In a shell script use something like this:

```
if grep -q shell bshellref
then
   echo "true"
else
   echo "false"
fi
```

Note that -q is the quiet version of grep. It just checks whether it is true that the string shell occurs in the file bshellref. It does not print the matching lines like grep would otherwise do.

I/O Redirection:

```
pgm > file      Output of pgm is redirected to file.
pgm < file      Program pgm reads its input from file.
pgm >> file     Output of pgm is appended to file.
pgm1 | pgm2     Output of pgm1 is piped into pgm2 as the input to pgm2.
n > file        Output from stream with descriptor n redirected to file.
n >> file       Output from stream with descriptor n appended to file.
n >& m          Merge output from stream n with stream m.
n <& m          Merge input from stream n with stream m.
<< tag          Standard input comes from here through next tag at start of line.
```

Note that file descriptor 0 is normally standard input, 1 is standard output, and 2 is standard error output.

Shell Built-in Variables:

```
$0              Name of this shell script itself.
$1              Value of first command line parameter (similarly $2, $3, etc)
$#              In a shell script, the number of command line parameters.
$*              All of the command line parameters.
$-              Options given to the shell.
$?              Return the exit status of the last command.
$$              Process id of script (really id of the shell running the script)
```

Pattern Matching:

```
*               Matches 0 or more characters.
?               Matches 1 character.
[AaBbCc]        Example: matches any 1 char from the list.
[^RGB]          Example: matches any 1 char not in the list.
[a-g]           Example: matches any 1 char from this range.
```

Quoting:

```
\c              Take character c literally.
`cmd`           Run cmd and replace it in the line of code with its output.
"whatever"      Take whatever literally, after first interpreting $, `...`, \
'whatever'      Take whatever absolutely literally.
```

Example:

```
   match=`ls *.bak`        Puts names of .bak files into shell variable match.
   echo \*                 Echos * to screen, not all filename as in:  echo *
   echo '$1$2hello'        Writes literally $1$2hello on screen.
   echo "$1$2hello"        Writes value of parameters 1 and 2 and string hello.
```

Grouping:

Parentheses may be used for grouping, but must be preceded by backslashes since parentheses normally have a different meaning to the shell (namely to run a command or commands in a subshell). For example, you might use:

```
if test \( -r $file1 -a -r $file2 \) -o \( -r $1 -a -r $2 \)
then
   do whatever
fi
```

Case statement:

Here is an example that looks for a match with one of the characters a, b, c. If $1 fails to match these, it always matches the * case. A case statement can also use more advanced pattern matching.

```
case "$1" in
   a) cmd1 ;;
   b) cmd2 ;;
   c) cmd3 ;;
   *) cmd4 ;;
esac
```

Shell Arithmetic:

In the original Bourne shell arithmetic is done using the expr command as in:

```
result=`expr $1 + 2`
result2=`expr $2 + $1 / 2`
result=`expr $2 \* 5`               #(note the \ on the * symbol)
```

With bash, an expression is normally enclosed using [ ] and can use the following operators, in order of precedence:

```
* / %       (times, divide, remainder)
+ -         (add, subtract)
< > <= >=   (the obvious comparison operators)
== !=       (equal to, not equal to)
&&          (logical and)
||          (logical or)
=           (assignment)
```

Arithmetic is done using long integers.

Example:

```
   result=$[$1 + 3]
```

In this example we take the value of the first parameter, add 3, and place the sum into result.

Order of Interpretation:

The bash shell carries out its various types of interpretation for each line in the following order:

```
brace expansion         (see a reference book)
~ expansion             (for login ids)
parameters              (such as $1)
variables               (such as $var)
command substitution    (Example:  match=`grep DNS *` )
arithmetic              (from left to right)
word splitting
pathname expansion      (using *, ?, and [abc] )
```

Other Shell Features:

```
$var          Value of shell variable var.
${var}abc     Example: value of shell variable var with string abc appended.
#             At start of line, indicates a comment.
var=value     Assign the string value to shell variable var.
```

```
cmd1 && cmd2   Run cmd1, then if cmd1 successful run cmd2, otherwise skip.
cmd1 || cmd2   Run cmd1, then if cmd1 not successful run cmd2, otherwise skip.
cmd1; cmd2     Do cmd1 and then cmd2.
cmd1 & cmd2    Do cmd1, start cmd2 without waiting for cmd1 to finish.
(cmds)         Run cmds (commands) in a subshell.
```

See a good reference book for information on traps, signals, exporting of variables, functions, eval, source, etc.