Grado en Ingeniería Informática

Administración de Sistemas

Programación en Shell

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

Universidad Zaragoza

3.1.Introducción

- Descripción
  - Programas interpretados orientados a comandos, procesos y ficheros.
- Objetivos :
  - Realización de programas para automatización de tareas de administración.
  - Modificación de programas shell de aplicaciones y de sistema (de instalación, configuración...).
- Elementos
  - Comentarios(carácter # hasta final de línea), variables, parámetros, comandos condicionales y repetitivos, funciones, recursión.
- Incompatibilidad entre algunos shells diferentes (Bourne y C,...).

3.2. Creación de un programa

- Elección de un tipo de shell
  - En nuestro caso Bourne shell (portabilidad), interpretado por el Bash shell (compatible Bourne).
- Construcción del programa con comandos
  - Directamente en la línea de comandos.
  - Fichero.
- Ejecución de un programa
  - $**sh** *fichero_programa*
  - $*fichero_programa*
    - Por defecto, es ejecutado por un proceso del shell en curso. Si se quiere explicitar un shell de ejecución en particular, se pone en la **primera línea** del fichero el shell a ejecutar con el formato **#!***acceso_completo_shell* :
      - **#!/bin/sh**
    - **Además, en este caso el programa shell debe tener permiso de ejecución** (**chmod u+x** *fichero_programa*).

3.3. Variables predefinidas

- Captura de parámetros del programa :

$0	El nombre del programa shell
$1 a **$9**	Del primero al noveno parámetro
$#	Nº de parámetros
$*	Captura todos los parámetros como una secuencia uniforme
$@	Captura todos los parámetros pero explícitamente separados

- Para utilizar más de 9 parámetros : comando **shift** (sin parámetros). Mueve todos los parámetros una variable a la izquierda ($1 pierde su valor y coge el de $2, $2 el de $3 y así hasta el $9)
- Otras variables predefinidas útiles en programación shell :

$?	Contiene el estado de salida (**ejecución correcta = 0**, o **errónea != 0**) del último comando (o proceso) ejecutado
$$	Contiene el id del proceso en curso
$!	Contiene el id del último proceso enviado como tarea de fondo

3.4. Estado de salida

- Estado en el que terminó la ejecución de un comando o programa shell (recogido en variable **$?**).
- Comandos que devuelven siempre el mismo estado de salida :
  - **true** (sin parámetros) : estado de salida = 0 (correcto).
  - **false** (sin parámetros) : estado de salida = 1 (erróneo).
- Comando **exit** :
  - Termina la ejecución de un programa shell y, si se explicita, devuelve un estado de salida determinado, sino, queda el estado de salida del último comando ejecutado.
    - **exit** [*estado_de_salida*] : **exit** 1.

3.5. El comando **test** (1).

- Objetivo : proveer un estado de salida en función de la verificación de serie de condiciones.
- Tipos de condiciones
  - Longitud de un string.
  - Comparación de dos strings.
  - Comparación de dos números.
  - Verificar el tipo de un fichero.
  - Verificar los permisos de un fichero.
  - Combinar condiciones.
- Dos formatos equivalentes :
  - **test** *expr* : **test** $1 = hola
  - **[** *expr* **]** : **[** $1 -gt $2 -o $1 -eq $2 **]**

3.5. El comando **test** (2).

- Expresiones cadenas de caracteres :

-z string	length of **string** is 0
-n string	length of **string** is not 0
string1 = string2	if the two **strings** are identical
string != string2	if the two **strings** are NOT identical
string	if **string** is not NULL

- Expresiones con enteros :

int1 -eq int2	first int is equal to second
int1 -ne int2	first int is not equal to second
int1 -gt int2	first int is greater than second
int1 -ge int2	first int is greater than or equal to second
int1 -lt int2	first int is less than second
int1 -le int2	first int is less than or equal to second

3.5. El comando **test** (y 3).

- Expresiones para verificar el estado de ficheros :

-r file	file exists and is readable
-w file	file exists and is writable
-x file	file exists and is executable
-f file	file exists and is a regular file
-d file	file exists and is directory
-h file	file exists and is a symbolic link
-c file	file exists and is a character special file
-b file	file exists and is a block special file
-p file	file exists and is a named pipe
-u file	file exists and it is setuid
-g file	file exists and it is setgid
-k file	file exists and the sticky bit is set
-s file	file exists and its size is greater than 0

- Operadores lógicos

!	reverse the result of an expression
-a	AND operator
-o	OR operator
(expr)	group an expression, parentheses have special meaning to the shell so to use them in the test command **you must quote them**

3.6. Comandos condicionales (1)

- Comandos simples :
  - *comando1* **&&** *comando2* : el segundo comando solo se ejecutará si el primero devuelve un estado de salida =0, es decir, ejecución correcta.
  - *comando1* **||** *comando2* : el segundo comando solo se ejecutará si el primero da lugar a un estado de salida erróneo, es decir, != 0.
- Comando if

if *expresion1*
then
comando
comando...
[**elif** *expresion2*
then
comando
comando...]...
[**else**
comando
comando...]
fi

3.6. Comandos condicionales (2)

– Ejemplo :

```
hour=`date | cut -c12-13`
if [ "$hour" -ge 0 -a "$hour" -le 11 ]
then
        echo "Buenos días"
elif [ "$hour" -ge 12 -a "$hour" -le 17 ]
        echo "Buenas tardes"
else
        echo "Buenas noches"
fi
```

3.6. Comandos condicionales (y 3)

- Comando **case** (patrones de substitución de ficheros) :

```
case valor in
   patrón1) comando1...
            comando2;;
   patrón2) comando3;; ...
esac
```

  - Ejemplo :

```
case $1 in
   hola)          echo hola que tal;;
   adios)         echo porque adios
                  echo si no has dicho hola;;
   T* | *T)       echo esta palabra empieza o termina con T;;
   [a-z]?)        echo comienza minúscula y dos letras;;
   *)             echo lo siento no conozco esta palabra;;
esac
```


3.7. Comandos repetitivos (1)

- Comando **for** :

```
for variable in palabra1 palabra2 ... palabran
do
   lista_de_comandos
done
```

- Ejemplo :

```
count=0
for param in $*   # para cada parámetro
do                # visualiza el numero de parámetro y su valor
   count=`expr $count + 1`
   echo el parametro $count es $param
done
```

3.7. Comandos repetitivos (2)

- Comando **while** :

```
while  comando
do
   lista_de_comandos
done
```

- Ejemplo :

```
count=10
while [  $count   -ge  0  ]
do
   echo $count
   count=`expr  $count  -  1`
done
```

- Comando **until** :

```
until  comando
do
   lista_de_comandos
done
```

- Ejemplo :

```
count=10
until   [  $count   -lt  0  ]
do
   echo $count
   count=`expr  $count  -  1`
done
```


3.7. Comandos repetitivos (3)

- Comando **break** : sale inmediatamente de un bucle, o de un número definido de bucles anidados (for, while o until).
  - **break**
  - **break** *número_de_bucles*
- Comando **continue** : sale de la iteración en curso para comenzar la siguiente en el mismo bucle, o en bucles anidados si parámetro.
- Redirección en bucles : afecta a todos los comandos del bucle, salvo que se hagan redirecciones explicitas dentro del bucle.

```
for directorio in $*
do
  ls n* 2> ficheros_no_encontrados
   ls $directorio
done 2> directorios.no.encontrados
```

3.7. Comandos repetitivos (y 4)

- Las variables **$@** y **$*** : lista de todos los parámetros pasados en un programa shell. Una diferencia importante :
  - Programa ejemplo *parametros* :

```
for param in $*
do
  echo $param
done
```

  - El shell reemplaza **$*** por los valores de $1 $2 $3... sin comillas ni otro metacarácter de quoting. Si uno de los valores tiene espacios u otros separadores de campo, estos son interpretados por el shell.
    - **$** *parametros* "1 2" 3
    - 1
    - 2
    - 3
  - El shell reemplaza **"$@"** por los valores de "$1" "$2" "$3"... Las comillas previenen de una interpretación de separadores de campo por el shell.
    (se reemplaza $* por **"$@"** en el programa parametros) **$** *parametros* "1 2" 3

```
1 2
3
```

- La variable predefinida **IFS** contiene los caracteres separadores. (por defecto : espacio y tabulación).

3.8. Lectura y escritura

- El comando **read** : toma una línea de la entrada estandar y afecta cada uno de sus campos a una variable. La última variable absorbe todos los campos que quedan. Devuelve un estado de salida 0, salvo que encuentre un fin de fichero o un <CTRL-D>.
  - **read** lista_de_variables
- Extensiones del comando **echo** :

-n	don't add the terminating new line character
-e	enable the ability to understand backslash escape characters
\a	alert (bell)
\b	backspace
\c	don't display the trailing newline
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\nnn	the character with ASCII number nnn (octal)

3.9. Funciones (1)

- La llamada de otro programa shell genera un proceso más. En cambio si utilizamos funciones, éstas se ejecutan EN EL MISMO PROCESO de llamada, utilizando el mismo contexto de ejecución. Su sintaxis es :

  *nombre_función***()**
  {
  comando
  comando...
  }

- Ejemplo :

```
buscar()
{
   grep  $1  /etc/passwd  >  /dev/null
}
```


3.9. Funciones (2)

- **Parámetros** : **$1 $2 ... $9** toman los parámetros pasados a la función (invalidando los parámetros de programa) y **$*** la lista de todos los parámetros pasados a la función. **$0** sigue con el nombre del programa shell.
- **El estado de terminación** es el del último comando de la función o el explicitado por el comando de terminación :

  return [*número _estado_salida*]

- (Bash shell : **Variables locales** a una función: comando **local** *variable_local*[*=valor*])

3.9. Funciones (y 3)

- Funciones shell recursivas : ejemplo de la función de visualización del orden inverso :

```
#!/bin/bash
recurre()
{
   if [ $# -ne 0 ]
   then
      shift
      recurre $*
      echo -n $1 " "
   fi
}

visualiza()
{
   recurre $*
   echo $1
}

visualiza $*
```


3.10. Visibilidad de variables y funciones

- Las variables exportadas :
  - Son copiados por valor en los contextos de los descendientes.
  - Se heredan : se pasa implícitamente de los hijos a los nietos...
  - Si una variable heredada de un proceso padre se modifica y quiere tenerse en cuenta dicha modificación para su descendencia, debe de explicitarse de nuevo como exportada (con el comando **export**).
- Dos métodos para hacer visible, al shell en curso, las variables y funciones definidas en ficheros :
  - Definición de variables y funciones en fichero *.profile* del usuario. Accesibles para el shell que se ejecuta al entrar en sesión.
  - Definición de funciones (o variables) en un fichero ejecutado con el comando "." . Accessible para el shell que lo ejecuta (no crea subproceso) :

$. *fichero*