Programación Bash

Sumario

Introdución

NOTA: Para un bo entendemento da programación bash é convinte ter coñecemento dos comandos GNU/Linux.
Ligazón de interese: Comandos básicos de xestión e administración Linux-

Á hora de administrar servidores seguramente repitiranse unha serie de tarefas como: crear, eliminar, editar usuarios e grupos, actualizacións varias do sistema e dos servizos, backups... polo tanto será convinte coñecer ferramentas que simplifiquen e axilicen o traballo do administrador do sistema. Algunhas veces atoparemos estas ferramentas, pero seguro que máis dunha vez non debido á particularidade do caso; dun xeito ou outro é convinte coñecer a **programación bash** que permitiranos simplificar as tarefas administrativas mediante guións programados denonimados scripts.
Os scripts bash polo tanto axudan moito, e nalgún caso, son o único camiño para poder levar a cabo a solución dos problemas administrativos de servidores. Baséanse na potencia dos comandos de GNU/Linux -ver,Comandos básicos de xestión e administración Linux- o cal permítelles desenvolver calquera tarefa administrativa.

O primeiro script: Ola Mundo

NOTA: Normalmente os scripts bash soen ter extensión **.sh**, pero non é obrigatorio.

Liña	Código fonte	Explicación do código fonte
1.	#!/bin/bash	Liña necesaria para saber que shell executará o script, neste caso o shell **bash**
2.	# Comentario: Primeiro Script Bash	Facemos un comentario explicativo do programa e/ou código fonte
3.	echo Ola Mundo	Amosa a cadea de texto Ola Mundo

Execución dun script bash

Basicamente existen dúas formas de execución:

1. Sen permisos de execución no script -que tamén funcionan se o script posúe permisos de execución-:
   - Mediante o comando bash:
     ```
     alumno@gnu-linux:~/temporal>cat olamundo.sh
     #!/bin/bash
     # Comentario: Primeiro Script Bash
     echo Ola Mundo
     alumno@gnu-linux:~/temporal>bash olamundo.sh
     Ola Mundo
     ```
   - Tamén podería executarse como **. olamundo.sh** ou **. ./olamundo.sh**
     ```
     alumno@gnu-linux:~/temporal> . olamundo.sh
     Ola Mundo
     alumno@gnu-linux:~/temporal> . ./olamundo.sh
     Ola Mundo
     ```
   - Ou tamén como **source olamundo.sh**
     ```
     alumno@gnu-linux:~/temporal> source olamundo.sh
     Ola Mundo
     ```
2. Dando permisos de execución e executando coa sintaxe **./**
   Esta opción soamente funciona se o script posúe permisos de execución.
   ```
   alumno@gnu-linux:~/temporal>ls -l olamundo.sh
   -rw-r--r-- 1 alumno users 62 dic 28 21:01 olamundo.sh
   alumno@gnu-linux:~/temporal>chmod +x olamundo.sh
   alumno@gnu-linux:~/temporal>ls -l olamundo.sh
   -rwxr-xr-x 1 alumno users 62 dic 28 21:01 olamundo.sh
   alumno@gnu-linux:~/temporal>./olamundo.sh
   Ola Mundo
   ```

NOTA: Aínda que os scripts Bash poden executarse mediante o comando **sh** non é recomendable. Para evitar sorpresas e execucións erróneas mellor empregar un dos métodos anteriormente descritos.

Elementos básicos

Na programación bash existen unha serie de elementos básicos común a outros linguaxes de programación: comentarios, variables, comiñas, parámetros, operadores, control de fluxo, funcións... Imos velos a continuación.

A primeira liña dun script bash

#!/bin/bash : Liña necesaria para saber que shell executará o script, neste caso o shell bash

Comentarios

Todas as liñas que comezen polo símbolo **#**, agás as liñas #! como a primeira liña #!/bin/bash, considéranse comentarios.

Mostrar texto por pantalla

Cando queremos devolver mensaxes por pantalla, que é moi normal nos scripts, empregaremos normalmente dous comandos: O comando echo e o comando printf.

- **echo**

Con *echo* imprimimos por pantalla expresións e variables, vexamos algún exemplo:

```
#!/bin/bash
# Unha expresión:
   echo "Ola"

# Unha variable:
   a="Bos días"
   echo $a

# Para empregar caracteres de escape
# emprégase a opción -e
echo -e "\tOla, bos días!!"

# Normalmente, cada echo imprímese nunha liña,
# para evitalo emprégase a opción -n
echo -n "Ola, "
echo "bos días!!"

# Para buscar patróns no contido dunha variable
   if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]
   then
     echo "$VAR contén a cadea de texto \"txt\""
   fi

# Podemos pasar a saída dun echo no valor dunha variable:
a=`echo "OLA" | tr A-Z a-z`
```

Opcións do comando *echo*:

Opción	Descrición
-n	non fai unha nova liña ao rematar o escrito
-e	activa a interpretación dos parámetros de escape
\b	retroceso
\\	barra invertida
\n	nova liña
\r	retorno de carro
\t	tab horizontal

Opción	Descrición
\v	tab vertical

- **printf**

A función *printf* (*formatted print*), é un *echo* mellorado. Trátase dunha variante da punción *printf()* da linguaxe C. Vexamos algún exemplo:

```
#!/bin/bash
declare -r PI=3,14159265258979           # Así definimos unha constante
declare -r numero=31373

mensaxe1="Ola"
mensaxe2="Bos días"

echo

printf "Pi con dous decimais = %1.2f\n" $PI
echo
printf "Pi con nove decimais = %1.9f\n" $PI

printf "\n"# Unha liña en branco

printf "Constante = \t%d\n" $numero# Insertamos unha tabulación

printf "%s %s\n" "$mensaxe1" "$mensaxe2"
echo

# Imprimir do 1 ao 5 un en cada liña:
for i in $( seq 1 5 )
   do printf "%d\n" "$i"
done
echo

# Imprimir do ao 001 ao 010 separados por unha tabulación:
for i in $( seq 1 10 )
   do printf "%03d\t" "$i"
done
echo
echo
# Traballar con táboas:
# O signo "-" indica alineación á esquerda
# o número, os espazos reservados para ese campo
printf  "%-5s %-10s %-6s\n" Num Nome  Nota
printf  "%-5s %-10s %-2.2f\n" 1 Sara 8,3456
printf  "%-5s %-10s %-2.2f\n" 2 Pedro 6,9989
printf  "%-5s %-10s %-2.2f\n" 3 María 10,0000

exit 0
```

◊ O comando declare permítenos modificar as propiedades das variables.

Variables e comiñas, comiñas e variables

NOTA: Ligazón de interese Variables de Entorno en Linux

Para o entendemento da programación bash é básico entender o funcionamento das comiñas e variables ou viceversa, pois están fortamente ligadas.

Variables

Ver o apartado **comiñas** para comprender as mesmas.

Crear variables

Para crear unha variable soamente temos que identificala cun nome e igualala ao seu valor, por exemplo:

```
# Comiñas dobres
NOME="usuario"
# Caracteres backticks ou comiñas invertidas
IP=`/sbin/ifconfig eth0`
# Comiñas simples
RUTA_PRINCIPAL='/home/alumno'
# Comiñas dobres
RUTA_SECUNDARIA="$RUTA_PRINCIPAL/tmp"
```

As variables en Bash se definen como NOME=valor (sen espazos antes ou despois do símbolo **=**) e o seu valor emprégase, poñendo o símbolo **$** diante do nome da variable, **$NOME**. Realmente, a anterior, é unha forma simplificada para recoller o valor das variables, sendo a forma xenérica a seguinte: NOME=${valor}. A forma xenérica debe empregarse sempre que a variable vaia seguida de letra, díxito ou guión baixo; noutro caso pode empregarse a forma simplificada. Igual uns exemplos poidan aclarar o comentado:

EXEMPLO A:

```
$ NOME='Ricardo'
$ APELIDO1='Feijoo'
$ APELIDO2='Costa'
$ echo $NOME_$APELIDO1_$APELIDO2
```

A execución deste exemplo devolve a seguinte saída na consola:

```
Costa
```

EXEMPLO B:

```
$ NOME='Ricardo'
$ APELIDO1='Feijoo'
$ APELIDO2='Costa'
$ echo ${NOME}_${APELIDO1}_${APELIDO2}
```

A execución deste exemplo devolve a seguinte saída na consola:

```
Ricardo_Feijoo_Costa
```

- **Matrices**

  Unha matriz é un conxunto de valores identificados polo mesmo nome de variable, onde cada unha das súas celas conta cun índice que a identifica. As matrices deben declararse mediante a cláusula interna ***declare -a***, antes de ser empregadas.
  Bash soporta matrices dunha única dimensión (vectores) cun único índice numérico, pero sen restricións de tamaño nin de orde numérico ou continuidade.
  Os valores das celas poden asignarse de maneira individual ou composta. Esta segunda fórmula permite asignar un conxunto de valores a varias celas do vector.
  Utilizar os caracteis especiais **[@]** ou **[*]** como índice da matriz supón referirse a todos os valores no seu conxunto.
  O mellor é ver un exemplo:

```
#!/bin/bash
##
vector=(un dous tres catro cinco seis)
##
echo "Facer un echo do array: "
echo $vector
echo "Así mellor (utilizando vector[*]): "
echo ${vector[*]}
echo "Ou así (empregando vector[@]): "
echo ${vector[@]}
echo "As diferenzas entre * e @, as de sempre..."
##
echo
echo "Ver o contido do elemento da posición 2: "
echo ${vector[2]}
echo "Eliminamos o elemento da posición 2: "
```

```
unset vector[2]
echo ${vector[*]}

echo "Adxudicamos o valor sete á posición 2 do array: "
vector[2]=sete
echo ${vector[*]}
echo "Vemos que os demáis corren a posición"

echo
echo "O número de elementos é: ${#vector[*]}"

echo "Podemos facer un loop sobre todos os elementos: "
contador=0
for elemento in ${vector[@]}
do
    echo "Elemento $contador: $elemento"
    contador=$(($contador+1))
done
```

- **Traballo con strings**

Se gardamos nunha variable un *string* seguramente nos interese desglosalo en cada un dos caracteres dos que está composto e, como non, coñecer o número de caracteres que ten:

```
#!/bin/bash
# Gardamos na variable "palabra" o primeiro parámetro:
palabra=$1
# Inicializamos i a 0
i=0
# Calculamos o número de caracteres do que está composto a palabra:
nelementos=${#palabra}
# Recorremos cada un dos caracteres da palabra:
while [ $i -lt $nelementos ]
do
caracter=${palabra:$i:1}
i=$(($i+1))
echo "Elemento posición $i: $caracter"
done

exit 0
```

Variables de tempo

Para traballar con datas (anos, meses, días, horas, minutos, segundos) dende *bash* Linux temos o comando date. Trátase dun comando con múltiples parámetros que nos permite obter a data actual, pasada ou futura representada en múltiples formatos.

- Exemplos có comando date

Pensando nos scripts, sería moi interesante poder calcular diferencias de tempo entre dúas datas, para realizar este tipo de cálculos emprégase, normalmente, as Epoch dates:

> **As epoch Unix (ou Unix time ou POSIX time ou Unix timestamp)** é o número de segundos que pasaron dende o 1 de Xaneiro do 1970 (medianoite UTC/GMT), sen contar os segundos intercalares (en ISO 8601: 1970-01-01T00:00:00Z). Falando claramente **o Timestamp é o tempo, en segundos, que pasou dende a media noite do día 1/1/1970 ata o momento actual.** Como se mostra como un enteiro de 32 bits, este número causará problemas o día 19 de Xaneiro do 2018.

Tempo	Timestamp
1 minuto	60 segundos
1 hora	3600 segundos
1 día	86400 segundos
1 semana	604800 segundos
1 mes (30,44 días)	2629743 segundos
1 ano (365,24 días)	31556926 segundos

Tempo	Timestamp

Así, para traballar con tempos nos *scripts* podemos ter moi en conta os seguintes comandos:

◊ Mostrar a data actual do sistema en formato *timestamp*:

```
$ date +%s
1384699849
```

◊ Pasar unha data calquera a formato *timestamp*:

```
$ date -d "Dec 24, 2002 15:15:15" +%s
1040739315
# Empregando outro formato de data:
$date -d "2002/12/24 15:15:15" +%s
1040739315
```

Para calcular canto tempo pasou entre unha e outra data só nos queda restar o *Timestamp* das dúas e saber que o resultado será en segundos.

Variables globlais e locais

Todas as variables nos scripts bash, a non ser que se definan doutro xeito, son **globais**, é dicir, todas unha vez definidas poden ser empregadas en calquera parte do script. Para que unha variable sexa **local**, é dicir, soamente teña senso dentro dunha sección do script -coma nunha función- e non en todo o script debe ser precedida pola sentenza: **local**. Por exemplo:

```
comezo sección
    local NOME=dentro
fin sección
```

onde, NOME soamente toma o valor **dentro** dentro da sección(función) onde se define. Realmente o exemplo anterior debería posuír unha función (ver o apartado [Funcións]).

```
dentro_variable_local() {
    local NOME=dentro
}
```

Quizás a execución dos seguintes exemplos, axuden:

Exemplo A: Sen a sentenza **local**

```
ONDE=script
echo $ONDE
dentro_variable_local() {
    ONDE=dentro
    echo $ONDE
}
dentro_variable_local
echo $ONDE
```

Exemplo B: Coa sentenza **local**

```
ONDE=script
echo $ONDE
dentro_variable_local() {
    local ONDE=dentro
    echo $ONDE
}
dentro_variable_local
echo $ONDE
```

Como se poder ver trala execución do **EXEMPLO A** obtemos a saída:

```
script
dentro
```

```
dentro
```

E trala execución do **EXEMPLO B** obtemos a saída:

```
script
dentro
script
```

Polo tanto conclúese que no **EXEMPLO A** ao non empregar a sentenza **local** a variable dentro da función é **global** e sobrescribe á variable definida antes da función.

Exportar variables

Mediante o comando **export**, por exemplo:

```
export IP=`/sbin/ifconfig eth0`
```

Exportar unha variable nun script significa que quedará cargada en memoria de forma permanente ata que se peche a execución do script.

Exportar variables ten unha gran importancia na execución dos scripts, pois moitas veces interésanos ter as variables cargadas en memoria durante toda a execución do script. Para saber as variables exportadas podemos executar na consola os comandos sen argumentos **env** ou **export**. Para saber tódalas variables, tanto as exportadas como non exportadas, executamos o comando **set**.

Veremos máis adiante na programación modular que é moi común crear un arquivo de variables a exportar para logo recollelo noutro arquivo do script mediante o comando **source**. A verdade é que podemos dicir que forman parella, case unha sen a outra non teñen moita razón de ser: **export-source**.

Recoller variables

Para recoller o valor que contén unha variable podemos empregar varios métodos:

- Mediante comiñas inclinadas, o comando echo e o caracter **$**, por exemplo:
  ```
  `echo $IP`
  ```
- Mediante o comando read, por exemplo:
  ```
  read numero
  ```

  O comando **read** é empregado habitualmente tras unha parada na execución dun script a espera dalgunha resposta, por exemplo nun script que posúe un menú, cunha opción de menú suma, onde necesitamos introducir os sumandos a través do teclado.

  - O comando **read** inclue unha opción **-p**, que permite especificar o texto de entrada:

  ```
  read -p "Introduce un número: " numero
  ```

  - Se non poñemos ningunha variable no comando **read** poderemos ler o introducido coa variable de contorno **REPLY**:

  ```
  read -p "Introduce un número: "
  echo "O teu número introducido é $REPLY"
  ```

  - Podemos poñer un temporizador no comando **read** coa opción **-t**. Cando o tempo expira, o comando **read** devolve un **exit** distinto de cero. Vexamos un exemplo:

  ```
  #!/bin/bash
  if read -t 5 -p "Introduce un número: " numero
  then
        echo "O teu número introducido é $numero"
  else
       echo
       echo "Sentímolo, demasiado lento"
  fi
  ```

  - Empregando a opción **-n** podemos indicarlle a **read** que acepte só un número de caracteres. No seguinte exemplo lle indicamos que ese número de caracteres sexa 1:

  ```
  #!/bin/bash
  read -n1 -p "Queres continuar [S/N]? " resposta
  case $resposta in
  ```

```
Y | y)  echo
        echo "Perfecto, continuemos...";;
N | n)  echo
        echo "Ok, ata outra!!"
        exit;;
esac
echo "Este é o final do script"
```

- Se queremos ler dun xeito "silencioso", que non se mostre por pantalla o que o usuario está a escribir (por exemplo para que este teclee un contrasinal), podemos empregar a opción **-s**. Vexamos un exemplo:

```
#!/bin/bash
read -s -p "Introduce o teu contrasinal:  " pass
echo
echo "O teu contrasinal é: $pass"
passl=`openssl passwd -salt proba $pass`

echo
echo "O teu contrasinal encriptado é: $passl"
echo
```

- Para rematar có comando **read** dicir que tamén é posible ler con el o contido dun documento de texto. Cada chamada do comando read pode ler unha liña de texto do arquivo. Cando non existan máis liñas no arquivo o comando **read** sairá devolvendo un estado **exit** distinto de cero. Vexamos un exemplo:

```
#!/bin/bash
contador=1
arquivo="/etc/passwd"
cat $arquivo | while read line
do
        echo "Liña $contador: $line"
        count=$(($count + 1))
done
echo
echo "Fin da lectura do arquivo $arquivo"
```

- Mediante o comando source, por exemplo:

```
source variables.txt
```

Permite cargar o arquivo variables.txt onde temos variables exportadas.

Comiñas

Na programación é moi importante entender o uso das comiñas, así nos scripts bash podemos atopar 3 tipos de comiñas:

- **Simples:** Non interpreta caracteres especiais coma o caracter **$**. Empregadas basicamente para obrigar ao script a ensinar coma texto o contido que se atopa nas mesmas.
- **Dobres:** Interpreta caracteres especiais coma o carácter **$**. Todo o que se atope entre elas e considerado como un só parámetro. Moi empregadas para recoller os valores das variables e combinalos con texto ou outras variables.
- **Invertidas**: Executa o contido dentro das comiñas. Son moi útiles na creación de variables xa que hai veces nas cales interésanos que a sáida da execución dun comando gárdese como valor nunha variable.

Exemplo:

Liña	Código fonte	Explicación do código fonte
1.	#!/bin/bash	Liña necesaria para saber que shell executará o script, neste caso o shell **bash**
2.	a=ls	Definimos variable **a** co valor ls
3.	echo '$a'	Amosa a cadea de texto **$a**
4.	echo "$a"	Amosa o contido da variable **a**, neste caso amosa a cadea de texto **ls**
5.	echo `$a`	Executa o contido dentro das comiñas, neste caso executa **ls**

Parámetros $

Parámetro $	**Explicación da súa función**

Parámetro $	Explicación da súa función
$0	Devolve o parámetro cero: o path do script
$1, $2 ...	Devolve cada un dos parámetros introducinos na execución do script
$#	Devolve o número total de parámetros pasados na execución do script(excluido $0)
$*	Devolve un *string* composto por todos os parámetros menos o $0.
$@	Devolve a lista completa de parámetros(excluido $0), separados por un espazo.
$$	Devolve o Identificador do proceso (PID)
$?	Devolve un valor numérico que indica como rematou o último comando: Todo ben (cero) algo mal (valor distinto de cero).
${!#}	Devolve o último parámetro.

Vexamos un exemplo onde se fai un repaso destes **parámetros $**:

```
#!/bin/bash
# Parámetros $
#
echo "Path completo do script: $0"
echo "Nome do script: `basename $0`"
echo "O PID do proceso de execución do script: $$"
if [ $# -ne 0 ]; then
  echo
  echo "Número de parámetros: $#"
  echo "Primeiro parámetro: $1"
  echo "Último parámetro: ${!#}"
  echo "Toda a lista de parámetros: $*"
  contador=1
  echo "Un parámetro por liña: "
  for param in $@
  do
        echo " - O parámetro \$$contador: $param"
        contador=$(($contador+1))
  done
else
  echo "Introduce algún parámetro!!!"
fi

exit 0
```

Manipulando variables

Enlace muy interesante

Operacións matemáticas

Liña	Código fonte	Explicación do código fonte
1. 2. 3. 4. 5.	#!/bin/bash expr 2 * 2 echo "2 * 2" | bc echo $((2*2)) let "sol = 2 * 2"	Liña necesaria para saber que shell executará o script, neste caso o shell **bash** Fai a operación 2*2 Fai a operación 2*2 Fai a operación 2*2 A variable "sol" ten o valor 2*2

Para facer operacións matemáticas é interesante ver como funciona o comando bc.

◊ Shell scripts de matemáticas.

Vexamos un exemplo:

```
#!/bin/bash
# Definimos dúas variables con números decimais:
```

```
num1=1,5
num2=2,3

# Se queremos sumalos, o único xeito é empregando "bc"
# O problema é que bc emprega o "." como símbolo decimal
# Debemos cambiar a "," polo "."
num1=`echo $num1 | tr -s ',' '.'`
num2=`echo $num2 | tr -s ',' '.'`

echo $num1
echo $num2

# Facemos a suma:
result=`echo "$num1 + $num2" | bc`

# Mostramos o resultado:
echo "$num1 + $num2 = $result"
```

Saída dos *scripts*

Todos os comandos que corren na *shell* empregan un estado de saída para indicar que o proceso foi realizado. O estado de saída é un valor enteiro entre 0 e 255 que o comando devolve ao *shell* cando este remata. Nós podemos capturar este valor e empregalo nos nosos *scripts*.

Linux proporciona a variable especial **$?** onde garda o estado de saída do último comando executado. Podemos ver o valor da variable **$?** inmediatamente despois da execución do comando para chequear o seu valor. Vexamos un exemplo:

```
$ date
Sat Oct 27 20:25:53 CEST 2012
$ echo $?
0
```

Sempre, o número devolto por un comando cando éste se completa ben é o **0**. Se o comando se completa cun erro, entón devolve un número positivo. Vexamos un exemplo:

```
$ cosa
-bash: cosa: command not found
$ echo $?
127
```

Control de Fluxo

- Enlace interesante en linuxcommand.org

Condicións

As estruturas condicionais permiten decidir se se realiza unha acción ou non; esta decisión tómase avaliando unha expresión.

Sentencia simple (if <condicion> then ...)

É a máis básica: **if** *<expresión>* **then** *<sentencia>* onde 'sentencia' só se executa se 'expresión' se avalía como verdadeira.
'2<1' é unha expresión que se avalía falsa, mentres que '2>1' avalíase verdadeira.
Exemplo:

```
alumno@gnu-linux:~/temporal>cat if_simple.sh
    #!/bin/bash
    if [ "linux" = "linux" ]; then
          echo Expresión avaliada como verdadeira
    fi
 alumno@gnu-linux:~/temporal>sh if_simple.sh
Expresión avaliada como verdadeira
```

O código que se executará se a expresión entre corchetes é verdadeira atópase entre a palabra 'then' e a palabra 'fi', que indica o final do código executado condicionalmente.

Neste outro exemplo vemos que se comproba se un comando funcionou ben ou non:

```
#!/bin/bash
if date
then
  echo "O comando traballou ben!!!"
fi
```

Sentencia dobre (if <condicion> then ... else ...)

Neste caso a estrutura é a seguinte: **if** *<expresión>* **then** *<sentencia1>* **else** *<sentencia2>*. Aquí 'sentencia1' execútase se 'expresión' é verdadeira, noutro caso execútase 'sentencia2'.
Exemplo:

```
alumno@gnu-linux:~/temporal>cat if_dobre.sh
#!/bin/bash
if [ "linux" = "unix" ]; then
       echo expresión avaliada como verdadeira
   else
      echo expresión avaliada como falsa
fi

alumno@gnu-linux:~/temporal>sh if_dobre.sh
expresión avaliada como falsa
```

Existe a posibilidade de concatenar varias sentencias **if** do seguinte xeito:

```
alumno@gnu-linux:~/temporal>cat if_concatenados.sh
#!/bin/bash
if [ "linux" = "unix" ]; then
       echo expresión avaliada como verdadeira
  else if [ "linux" = "windows" ]; then
         echo segunda expresión avaliada como verdadeira
  else
       echo expresión avaliada como falsa
  fi
fi

alumno@gnu-linux:~/temporal>sh if_concatenados.sh
expresión avaliada como falsa
```

A saída anterior prodúcese coma resultado de que ningunha das dúas condicións é certa.

Operadores numéricos de comparación

Temos os seguintes operadores aritméticos de comparación:

◊ Menor que : -lt
◊ Maior que : -gt
◊ Menor ou igual que : -le
◊ Maior ou igual que : -ge
◊ Igual que : -eq
◊ Distinto que : -ne

Pero non valen para comparar decimais.

Operadores para a comparación de texto

Temos os seguintes operadores aritméticos de comparación:

◊ Menor que : \<
◊ Maior que : \>
◊ Igual que : = ou ==
◊ Distinto que : !=
◊ String con tamaño maior que 0 : n palabra
◊ String con tamaño igual a 0 : z palabra

Tede en conta que **sort** e **test** (comparación con corchetes) funcionan ao revés con respecto ao trato das maiúsculas e minúsculas.

Podemos utilizar **"-a"** para **AND** y **"-o"** para **OR**. Por exemplo:

```
#!/bin/bash
if [ $1 -ge 3 -a $2 -lt 10 ]; then
...
```

Probar características dun ficheiro

Podemos probar moitas características dun ficheiro có comando [test] e outras opcións. Na seguinte táboa vemos algunhas delas:

Opción	Comproba o ficheiro para ver si
-d	Existe e é un directorio.
-e	Existe.
-f	Existe e é un ficheiro normal (non un directorio).
-r	Existe e pódese ler.
-s	Existe e ten un tamaño maior que 0 bytes.
-w	Existe e pódese escribir.
-x	Existe e pódese executar.
-O	Existe e o dono é o usuario que executa o script.
-G	Existe e o o usuario que executa o script pertence ao grupo dono do arquivo.
arquivo1 -nt arquivo2	O arquivo1 é máis recente que o arquivo2.
arquivo1 -ot arquivo2	O arquivo1 é máis antigo que o arquivo2.

◊ Enlace interesante.

Anidar condicións Test

- Condición1 AND Condición2:

```
[[ condición1 ]] && [[ condición2 ]]
```

- Condición1 OR Condición2:

```
[[ condición1 ]] || [[ condición2 ]]
```

Bucles

Empregados nas situacións onde debemos repetir varias veces a execución do/s mesmo/s comando/s.

for

O comando **for** é un bucle que se emprega cando sabemos as veces que debemos reiterar os comandos, isto é, sabemos o inicio da execución e a finalización do mesmo. Comunmente é empregado para facer contadores.
Exemplo: Contador. Durante 10 veces execútase o bucle amosando a secuencia de números do 1 ao 10

Liña	Código fonte	Explicación do código fonte
1.	#!/bin/bash	Liña necesaria para saber que shell executará o script, aquí o shell **bash**
2.	for i in `seq 1 10`	Bucle for contador: i toma o valor do 1 ao 10
3.	do	Facer os comandos do bucle

4.	{	Comezo do bucle
5.	echo $i	A primeira execución do bucle i toma valor 1, a segunda 2... ata 10
6.	}	Fin comandos do bucle
7.	done	Reiterar bucle ata finalización do contador

```
alumno@gnu-linux:~/temporal>cat bucle_for_contador.sh
#!/bin/bash
for i in `seq 1 10`
do
{
echo $i
}
done
alumno@gnu-linux:~/temporal>sh bucle_for_contador.sh
1
2
3
4
5
6
7
8
9
10
```

- **Podemos ler os valores dunha lista:**

```
root@debian:/scripts# cat proba.sh
#!/bin/bash
for test in Madrid Lugo Pontevedra Barcelona
do
        echo A seguinte provincia é $test
done
root@debian:/scripts# ./proba.sh
A seguinte provincia é Madrid
A seguinte provincia é Lugo
A seguinte provincia é Pontevedra
A seguinte provincia é Barcelona
```

A sentencia **for** asume que cada valor é separado por espazos. Se os valores conteñen espazos teremos algún problema:

```
root@debian:/scripts# cat proba.sh
#!/bin/bash
for test in Madrid Lugo Pontevedra Barcelona A Coruña Ourense
do
        echo A seguinte provincia é $test
done
root@debian:/scripts# ./proba.sh
A seguinte provincia é Madrid
A seguinte provincia é Lugo
A seguinte provincia é Pontevedra
A seguinte provincia é Barcelona
A seguinte provincia é A
A seguinte provincia é Coruña
A seguinte provincia é Ourense
```

Para resolver este problema podemos facelo empregando dobres comiñas:

```
root@debian:/scripts# cat proba.sh
#!/bin/bash
for test in "Madrid" "Lugo" "Pontevedra" "Barcelona" "A Coruña" "Ourense"
do
        echo A seguinte provincia é $test
done
root@debian:/scripts# ./proba.sh
A seguinte provincia é Madrid
A seguinte provincia é Lugo
A seguinte provincia é Pontevedra
A seguinte provincia é Barcelona
```

```
A seguinte provincia é A Coruña
A seguinte provincia é Ourense
```

- **Ler unha lista dunha variable**

```
root@debian:~# cat proba.sh
#!/bin/bash
lista="Madrid Lugo Pontevedra Barcelona A Coruña Ourense"
for test in $lista
do
        echo A seguinte provincia é $test
done
root@debian:~# ./proba.sh
A seguinte provincia é Madrid
A seguinte provincia é Lugo
A seguinte provincia é Pontevedra
A seguinte provincia é Barcelona
A seguinte provincia é A
A seguinte provincia é Coruña
A seguinte provincia é Ourense
```

Para solucionar este problema podemos empregar a variable **IFS** que por defecto os separadores son: o espazo, o tabulador e o retorno de carro.

```
root@debian:~# cat proba.sh
#!/bin/bash
lista="Madrid:Lugo:Pontevedra:Barcelona:A Coruña:Ourense"
IFS=":"  #Configuramos IFS para que o separador sexa :
for test in $lista
do
        echo A seguinte provincia é $test
done
unset IFS  #Para voltar IFS ao seu estado por defecto
root@debian:~# ./proba.sh
A seguinte provincia é Madrid
A seguinte provincia é Lugo
A seguinte provincia é Pontevedra
A seguinte provincia é Barcelona
A seguinte provincia é A Coruña
A seguinte provincia é Ourense
```

Outros exemplos de IFS serían:
- Só retorno de carro: **IFS=$'\n'**
- Retorno de carro, dous puntos e punto e coma: **IFS=$'\n':;**

- **Ler un directorio.**

Podemos empregar o comando **for** para iterar polo contido dun directorio.
Importante recordar que, se empregamos tamén as sentenzas "Test" para comprobar, por exemplo, se estamos a tratar cun directorio ou cun arquivo, hai que poñer a variable entre comiñas dobres para que o script funcione tamén con nomes de arquivos e directorios con espazos.

```
#!/bin/bash
echo "Os directorios existentes: "
for file in /home/usuario/*
do
    if [ -d "$file" ]; then
        echo "$file"
    fi
done
```

- **Estilo C**

Podemos utilizar o comando **for** có estilo da linguaxe de programación C. Vexamos un exemplo:

```
#!/bin/bash
echo "Una secuencia del 1 al 10: "
for (( a=1; a<=10; a++ ))
do
    echo "Número $a"
```

```
done
```

Empregando múltiples variables:

```
#!/bin/bash
echo "Una secuencia del 1 al 10: "
for (( a=1, b=100 ; a<=5 ; a++, b-- ))
do
    echo "Número A: $a"
    echo "Número B: $b"
    echo
done
```

- Un exemplo interesante de utilización de anidamento de bucles **for** é o seguinte, no que recorremos cada un dos "elementos" gardados no arquivo /**etc**/**passwd**:

```
#!/bin/bash
IFS=$?\n?
for entry in `cat /etc/passwd`
do
  echo "Valores na liña: $entry "
  IFS=:
  for value in $entry
  do
    echo " - $value"
    IFS=$'\n'
  done
done
unset IFS
```

- O comando ***break*** permítenos "escapar" fora dun *loop*. Se utilizamos *exit* termínase o script, pero se utilizamos *break* saímos da iteración (*for*, *while*) sen terminar a execución do *script*.

```
#!/bin/bash
#  breaking  out  of  a  for  loop
for  var1  in  1  2  3  4  5  6  7  8  9  10
do
    if  [  $var1  -eq  5  ]
    then
        break
    fi
    echo  ?Iteración: $var1?
done
echo  ?O bucle está completado?
```

- Saír dun bucle interno. Traballando con múltiples bucles queremos saír do interno e seguir có externo.

```
#!/bin/bash
#breaking  out  of  an  inner  loop
for  ((  a  =  1;  a  ?  4;  a++  ))
do
    echo  "Bucle Externo:  $a"
    for  ((  b  =  1;  b  ?  100;  b++  ))
    do
        if  [  $b  -eq  5  ]
        then
            break
        fi
    echo  "      Bucle Interno:  $b"
    done
done
```

- Saír dun bucle externo. Facemos a comparación no bucle interno pero paramos o bucle externo. Aquí o comando *break* inclúe un parámetro enteiro:

break n

onde **n** indica o nivel do bucle externo do que imos saír. Se **n** vale 2, indicamos que

imos parar o bucle inmediatamente superior ao que estamos.

```
#!/bin/bash
#  breaking  out  of  an  outer  loop
for  ((  a  =  1;  a  ?  4;  a++  ))
do
    echo  "Bucle externo:  $a"
    for  ((  b  =  1;  b  ?  100;  b++  ))
    do
        if  [  $b  -gt  4  ]
        then
            break  2
        fi
        echo  "      Bucle interno:  $b"
    done
done
```

- **O comando *continue*.** Este comando permítenos parar prematuramente un bucle, pero non terminalo completamente. Aquí temos un exemplo:

```
#!/bin/bash
#  using  the  continue  command
for  ((  var1  =  1;  var1  ?  15;  var1++  ))
do
    if  [  $var1  -gt  5  ]  &&  [  $var1  -lt  10  ]
    then
        continue
    fi
    echo  "Número de iteración:  $var1"
done
```

Cando se cumplen as condicións da sentenza *if-then* o *shell* executa o comando *continue*, que fai que se salte o resto dos comandos existentes dentro do bucle, pero volve a executarse novamente.

while

O comando **while**:
Exemplo: Contador. Durante 10 veces exécutase o bucle amosando a secuencia de números do 1 ao 10

Liña	Código fonte	Explicación do código fonte
1.	#!/bin/bash	Liña necesaria para saber que shell executará o script, aquí o shell **bash**
2.	i=1	Definimos variable i con valor un
3.	while [$i ?le 10]	Comeza bucle contador onde a variable i toma o valor de 1 a 10: Namentras i sexa menor ou igual a 10
4.	do	Facer
5.	echo Valor de i: $i	Ensina o valor da variable i para cada valor do bucle, sendo o primeiro valor un
6.	i=$(($i+1))	Aumenta unha unidade o valor anterior, se era un, entón agora valor igual a dous
7.	done	Reiterar bucle ata finalización do contador

Para poder saír do while emprégase o comando break

until

O comando **until**:
Exemplo: Contador. Durante 10 veces exécutase o bucle amosando a secuencia de números do 1 ao 10

Liña	Código fonte	Explicación do código fonte
1.	#!/bin/bash	Liña necesaria para saber que shell executará o script, aquí o shell **bash**
2.	i=1	Definimos variable i con valor un
3.	until [$i ?ge 11]	Comeza bucle contador onde a variable i toma o valor de 1 a 10: Ata que i sexa maior ou igual a 11
4.	do	Facer
5.	echo Valor de i: $i	Ensina o valor da variable i para cada valor do bucle, sendo o primeiro valor un
6.	i=$(($i+1))	Aumenta unha unidade o valor anterior, se era un, entón agora valor igual a dous
7.	done	Reiterar bucle ata finalización do contador

Menús

Exemplo: Menú.

Liña	Código fonte	Explicación do código fonte
1.	#!/bin/bash	Liña necesaria para saber que shell executará o script, aquí o shell **bash**
2.	echo Opcion1. Ver directorio actual	Amosa por pantalla Opcion 1. Ver directorio actual
3.	echo Opcion2. Ler /tmp	Amosa por pantalla Opcion2. Ler /tmp
4.	echo Opcion3. Sair	Amosa por pantalla Opcion 3. Sair
5.	echo Elixe opcion:1,2,3?	Escoller opcion
6.	read opcion	A opción escollida gárdase como valor na variable opcion
7.	case $opcion in	Comezo case para facer menu
8.	1) pwd	Se o valor da variable opcion é 1 fanse os seguintes comandos.
9.	;;	Toda opción debe acabar con ;;
10.	2) ls /tmp	Se o valor da variable opcion é 1 fanse os seguintes comandos.
11.	;;	Toda opción debe acabar con ;;
12.	3) exit	Se o valor da variable opcion é 1 fanse os seguintes comandos.
13.	;;	Toda opción debe acabar con ;;
14.	*) echo non elixiches nin 1,2,3	Mensaxe por pantalla no caso de non escoller ningunha das opcións do menú
15.	;;	Toda opción debe acabar con ;;
16.	esac	Fin case para facer menu

- Exit

Ferramentas GNU/Linux moi útiles nos scripts

Procura de patróns en arquivos

Entre as ferramentas máis útiles en Linux están aquelas que buscan palabras en arquivos: grep, fgrep e egrep. Estas ordes buscan liñas que conteñan texto identificado por un patrón nun ou varios obxectivos. Pódense empregar para extraer información dos arquivos, buscar liñas que se relacionen cun elemento particular e para localizar arquivos que conteñan unha palabra clave particular.

As tres ordes que imos describir son moi similares. Diferéncianse na forma de especificar os obxectivos da procura:

- grep
- fgrep
- egrep

Operación con columnas e campos

Moitos arquivos conteñen información organizada en termos de posicións dentro dunha liña. Linux ten varias ferramentas deseñadas específicamente para operar con arquivos organizados en columnas ou campos. Pódense empregar as seguintes ordes que imos ver para extraer e modificar ou reorganizar a información de arquivos estruturados en campos ou por columnas.

- cut permite seleccionar columnas ou campos particulares de arquivos.
- paste crea novos arquivos xuntando liña a liña columnas ou campos de outros existentes.
- join xunta a información de dous arquivos segundo un criterio dado para crear un novo que combina a información de ambos.

Ferramentas de Manipulación de textos

En Linux temos varias ferramentas para manipular arquivos de texto. Entre elas as máis importantes son:

- sed
- awk

xargs

Utilizado para modificar a saída dun comando, entre outras cousas, permite convertir columnas en filas, polo tanto é comunmente empregado conxuntamente con **sed** e **awk** para recoller o que queiramos na saida da execución dun comando en variables no script.

Exemplo: Nun ficheiro que contén unha columna con nomes de usuarios mediante xargs amosámolo na shell coma unha fila.

```
$ cat usuarios.txt
Usuario1
Usuario2
Usuario3
Usuario4
$ cat usuarios.txt | xargs
Usuario1 Usuario2 Usuario3 Usuario4
```

Outra utilidade típica de **xargs** é a de facer que a saída dun comando se empregue como parámetro doutro comando, vexamos un exemplo no que queremos borrar os arquivos con extensión .txt existentes no ficheiro de traballo:

```
$ ls
arq1.txt  foto.bmp  music.mp3  parq2.txt  qarq3.txt

$ find . -name "*.txt" | xargs rm -rf

$ ls
foto.bmp  music.mp3
```

sort e uniq

sort: Comando que permite ordear, de varios xeitos: orde numérico, alfabético..., as liñas de ficheiros de texto ou a saída de texto da execución dun comando na consola. Por defecto ordea por orde alfabético.
sort -> Exemplo1: Ordear as liñas dun ficheiro de texto por orde alfabético crecente

```
alumno@gnu-linux:~/temporal>cat cousas.txt
rato
teclado
folla
papel
cd
dvd
revista
porta
alumno@gnu-linux:~/temporal>cat cousas.txt | sort
cd
dvd
folla
papel
porta
rato
revista
teclado
```

sort -> Exemplo 2: Ordear as liñas dun ficheiro de texto por orde alfabético decretente

```
alumno@gnu-linux:~/temporal>cat cousas.txt
rato
teclado
folla
papel
cd
dvd
revista
porta
alumno@gnu-linux:~/temporal>cat cousas.txt | sort -r
teclado
revista
rato
porta
papel
folla
dvd
cd
```

uniq: Comando que permite evitar as concurrencias, isto é, permite omitir liñas repetidas. **IMPORTANTE**: *O comando terá éxito soamente se antes actuou o comando* ***sort*** .

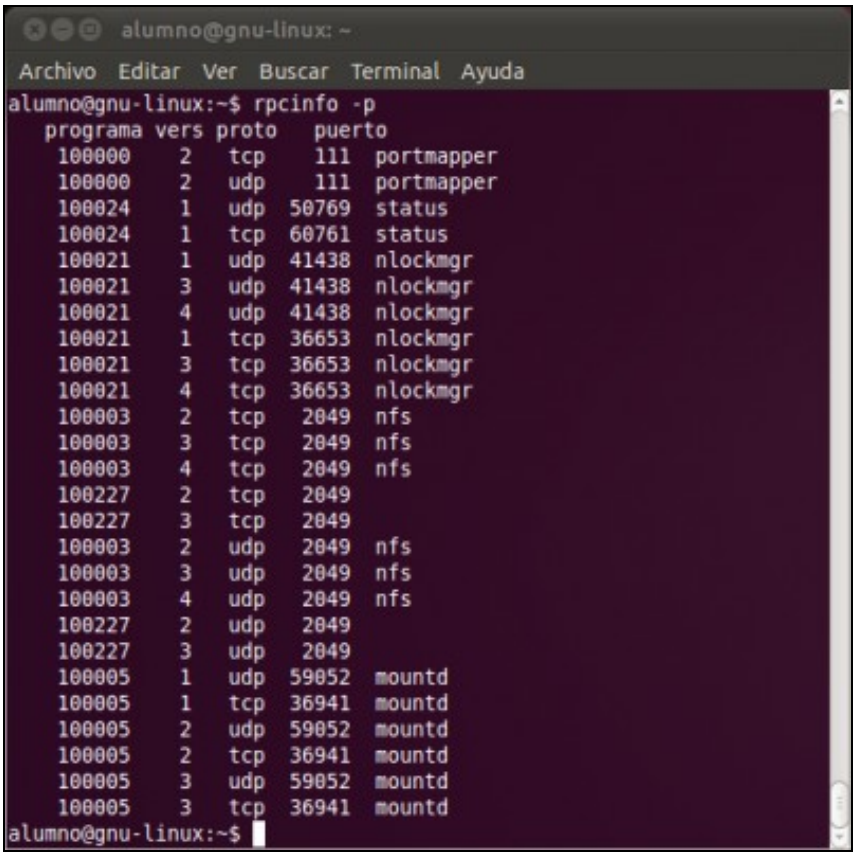
uniq --> Exemplo: Evitar concurrencias nun ficheiro de texto

```
alumno@gnu-linux:~/temporal>cat cousas2.txt
rato
rato
teclado
teclado
folla
folla
folla
papel
papel
cd
cd
dvd
revista
porta
cd
cd
cd
cd
alumno@gnu-linux:~/temporal>cat cousas2.txt | sort
cd
cd
cd
cd
cd
cd
dvd
folla
folla
folla
papel
papel
porta
rato
rato
revista
teclado
teclado
alumno@gnu-linux:~/temporal>cat cousas2.txt | sort | uniq
cd
dvd
folla
papel
porta
rato
revista
teclado
```

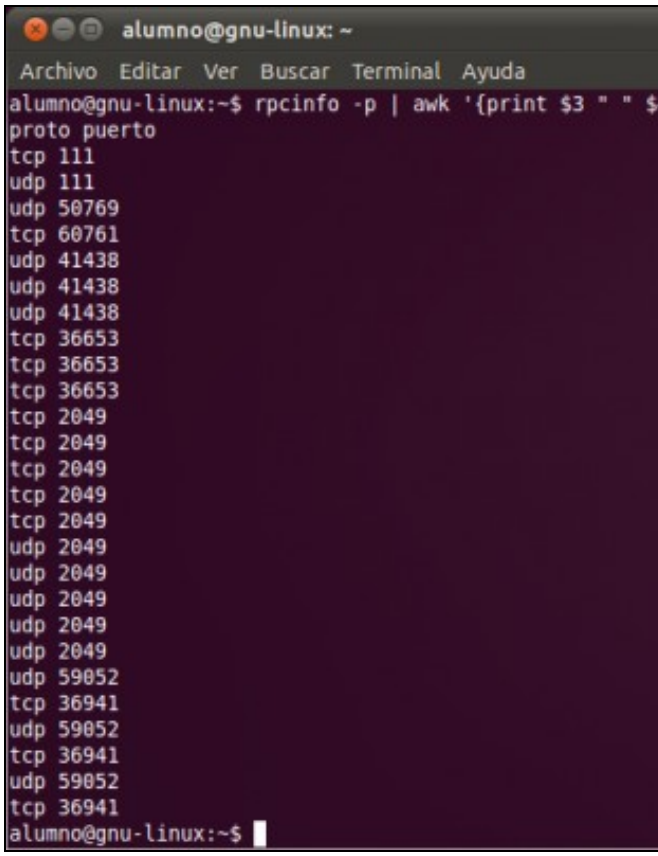
sort e uniq: Exemplo en liña de comandos: Capturar os portos tcp e portos udp do servidor nfs para abrir con iptables

NOTAS:

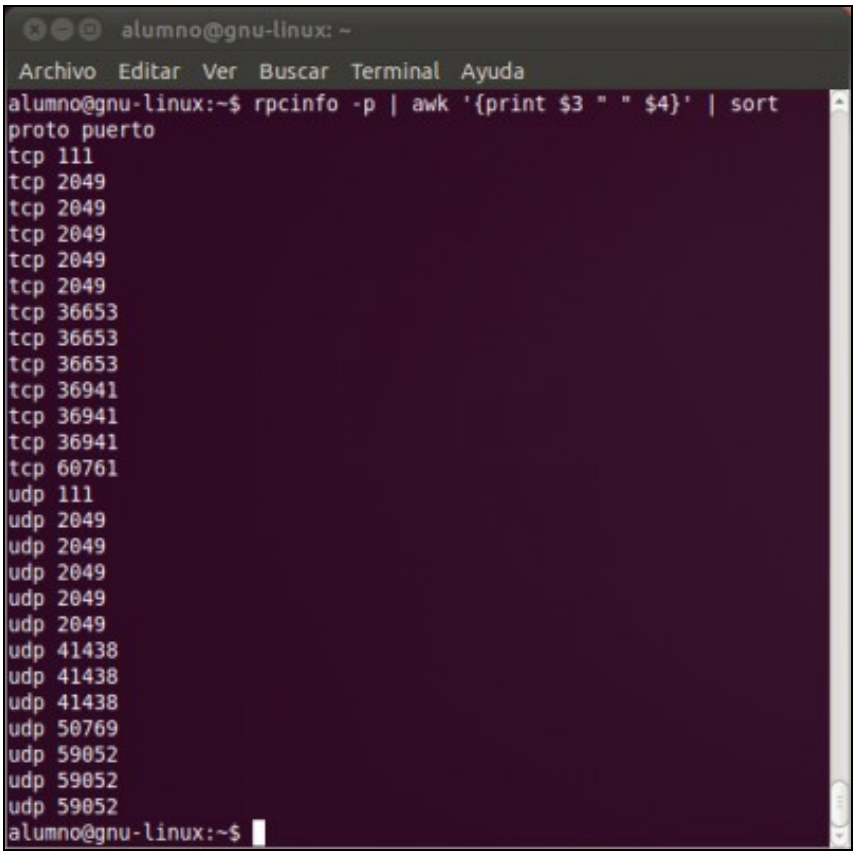
1. **Picar nas imaxes para velas no tamaño orixinal**
2. **Para este exemplo é necesario ten instalado un servidor NFS.**
3. **Para Ubuntu instalar o paquete nfs-kernel-server: apt-get install nfs-kernel-server**

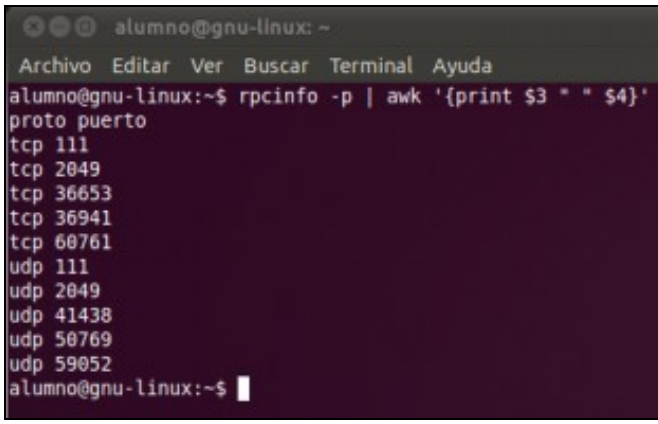


a. Saída do comando rpcinfo -p para saber os portos tcp e udp que abre o servidor NFS



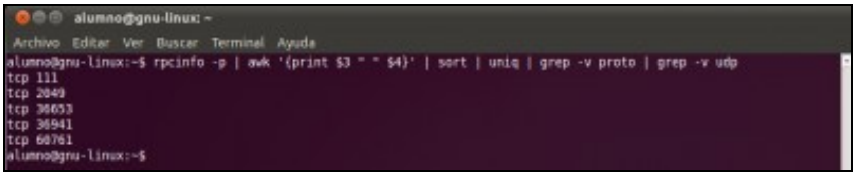
b. Filtrar mediante awk a saída do comando rpcinfo -p para que s as columnas proto e porto.

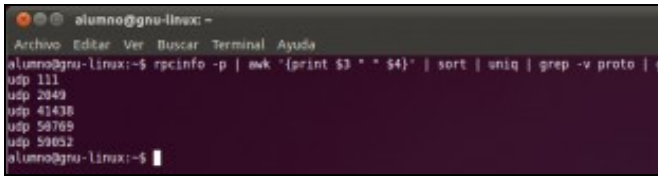




d. Filtrar mediante uniq para evitar as repeticións de portos tcp e servidor NFS.

c. Filtrar mediante sort para obter unha lista ordeada.





e. Portos tcp que abre o servidor NFS.

f. Portos udp que abre o servidor NFS.

Exemplo en script: Versión 0.1: Capturar os portos tcp e portos udp do servidor nfs para abrir con iptables

Liña	Código fonte
1.	#!/bin/bash
2.	
3.	NUMERO_PORTOS_TCP=`rpcinfo -p | awk '{print $3 " " $4}' | sort | uniq | grep -v proto | grep -v udp | awk '{print $2}' | wc -l`
4.	NUMERO_PORTOS_UDP=`rpcinfo -p | awk '{print $3 " " $4}' | sort | uniq | grep -v proto | grep -v tcp | awk '{print $2}' | wc -l`
5.	for i in `seq 1 $NUMERO_PORTOS_TCP`
6.	do
7.	{
8.	PORTOS_TCP=`rpcinfo -p | awk '{print $3 " " $4}' | sort | uniq | grep -v proto | grep -v udp | awk '{print $2}' \| head -$i | tail -1`
9.	iptables -I INPUT -p tcp --dport $PORTOS_TCP -j ACCEPT
10.	}
11.	done
12.	for i in `seq 1 $NUMERO_PORTOS_UDP`
13.	do
14.	{
15.	PORTOS_UDP=`rpcinfo -p | awk '{print $3 " " $4}' | sort | uniq | grep -v proto | grep -v tcp | awk '{print $2}' \| head -$i | tail -1`
16.	iptables -I INPUT -p tcp --dport $PORTOS_UDP -j ACCEPT
17.	}
	done

Liña	Explicación do código fonte
1.	Liña necesaria para saber que shell executará o script, neste caso o shell **bash**
2.	Definición da variable NUMERO_PORTOS_TCP que contén o número de portos tcp que abre o servidor NFS.
3.	Definición da variable NUMERO_PORTOS_UDP que contén o número de portos udp que abre o servidor NFS.
4.	Bucle for que terá lugar tantas veces coma portos tcp abra o servidor NFS
5.	O que fai o bucle.
6.	Abrése chave
7.	Cada vez que se percorre o bucle colle un porto TCP que abre o servidor NFS.
8.	Regra iptables que abre o porto anterior.
9.	Peche de chave
10.	Fin do bucle for
11.	Bucle for que terá lugar tantas veces coma portos udp abra o servidor NFS
12.	O que fai o bucle.
13.	Abrése chave
14.	Cada vez que se percorre o bucle colle un porto UDP que abre o servidor NFS.
15.	Regra iptables que abre o porto anterior.
16.	Peche de chave
17.	Fin do bucle for

wc

O comando wc serve para contar: caracteres, bytes, palabras e liñas dun ficheiro ou da saída en consola da execución dun comando. É comunmente empregado co bucle **for** para facer contadores.

- **Exemplo 1: Contar as liñas dun ficheiro de texto**

  Empregado como comando:

```
alumno@gnu-linux:~/temporal>wc -l cousas.txt
```

```
8 cousas.txt
```

Empregado como filtro:

```
alumno@gnu-linux:~/temporal>cat cousas.txt | wc -l
8
```

- **Exemplo 2: Contar as liñas da saída da execución dun comando**

```
alumno@gnu-linux:~/temporal>ls -l
total 8
-rw-r--r-- 1 alumno users 92 dic 28 19:39 cousas2.txt
-rw-r--r-- 1 alumno users 46 dic 28 19:26 cousas.txt
alumno@gnu-linux:~/temporal>ls -l | wc -l
3
```

- **Exemplo 3: Empregado co bucle for para facer un contador**

```
alumno@gnu-linux:~/temporal>cat cousas.txt
rato
teclado
folla
papel
cd
dvd
revista
porta
alumno@gnu-linux:~/temporal>cat contador.sh
#!/bin/bash
NUM=`cat ~/temporal/cousas.txt | wc -l`
for i in `seq 1 $NUM`
do
{
LINHA=`cat cousas.txt | head -$i | tail -1`
echo "Liña número $i: $LINHA"
}
done
alumno@gnu-linux:~/temporal>sh contador.sh
Liña número 1: rato
Liña número 2: teclado
Liña número 3: folla
Liña número 4: papel
Liña número 5: cd
Liña número 6: dvd
Liña número 7: revista
Liña número 8: porta
```

Liña	Código fonte	Explicación do código fonte
1.	#!/bin/bash	Liña necesaria para saber que shell executará o script, aquí o shell **bash**
2.	NUM=`cat ~/temporal/cousas.txt | wc -l`	Definición variable NUM do contador, igual ao nº de liñas arquivo cousas.txt
3.	for i in `seq 1 $NUM`	Bucle for contador: dende o 1 ata o valor da variable NUM
4.	do	Facer os comandos do bucle
5.	{	Comezo do bucle
6.	LINHA=`cat cousas.txt | head -$i | tail -1`	Definición variable LINHA. Cada vez que se percorre o bucle colle unha liña.
7.	echo "Liña número $i: $LINHA"	Amosar por pantalla o texto Liña número valor_variable_i: texto_da_liña
8.	}	Fin comandos do bucle
9.	done	Reiterar bucle ata finalización do contador

join

Manexo de arquivo .csv

Manexaremos a información de arquivos csv con **csvkit**. O programa csvkit é unha ferramenta desenrolada en Python que facilita a manipulación da información contida nun arquivo con formato csv.

A instalación de **csvkit** faise do seguinte xeito:

$ apt-get intall python $ apt-get install python-pip $ pip install csvkit

Traballaremos có arquivo "archivo.csv" que ten o seguinte contido: $ cat archivo.csv nombre,oficina,direccion,nacimiento,sueldo Steve Blenheim,238-923-7366,"95 Latham Lane, Easton, PA 83755",11/12/56,20300 Betty Boop,245-836-8357,"635 Cutesy Lane, Hollywood, CA 91464",6/23/23,14500 Igor Chevsky,385-375-8395,"3567 Populus Place, Caldwell, NJ 23875",6/18/68,23400 Norma Corder,397-857-2735,"74 Pine Street, Dearborn, MI 23874",3/28/45,245700 Jennifer Cowan,548-834-2348,"583 Laurel Ave., Kingsville, TX 83745",10/1/35,58900 Jon DeLoach,408-253-3122,"123 Park St., San Jose, CA 04086",7/25/53,85100 Karen Evich,284-758-2867,"23 Edgecliff Place, Lincoln, NB 92743",11/3/35,58200 Fred Fardbarkle,674-843-1385,"20 Parak Lane, Duluth, MN 23850",4/12/23,780900 Lori Gortz,327-832-5728,"3465 Mirlo Street, Peabody, MA 34756",10/2/65,35200 Paco Gutierrez,835-365-1284,"454 Easy Street, Decatur, IL 75732",2/28/53,123500 Ephram Hardy,293-259-5395,"235 CarltonLane, Joliet, IL 73858",8/12/20,56700 James Ikeda,834-938-8376,"23445 Aster Ave., Allentown, NJ 83745",12/1/38,45000 Barbara Kertz,385-573-8326,"832 Ponce Drive, Gary, IN 83756",12/1/46,268500 Lesley Kirstin,408-456-1234,"4 Harvard Square, Boston, MA 02133",4/22/62,52600 William Kopf,846-836-2837,"6937 Ware Road, Milton, PA 93756",9/21/46,43500 Sir Lancelot,837-835-8257,"474 Camelot Boulevard, Bath, WY 28356",5/13/69,24500 Jesse Neal,408-233-8971,"45 Rose Terrace, San Francisco, CA 92303",2/3/36,25000 Zippy Pinhead,834-823-8319,"2356 Bizarro Ave., Farmount, IL 84357",1/1/67,89500 Arthur Putie,923-835-8745,"23 Wimp Lane, Kensington, DL 38758",8/31/69,126000 Popeye Sailor,156-454-3322,"945 Bluto Street, Anywhere, USA 29358",3/19/35,22350 Jose Santiago,385-898-8357,"38 Fife Way, Abilene, TX 39673",1/5/58,95600 Tommy Savage,408-724-0140,"1222 Oxbow Court, Sunnyvale, CA 94087",5/19/66,34200 Yukio Takeshida,387-827-1095,"13 Uno Lane, Ashville, NC 23556",7/1/29,57000 Vinh Tranh,438-910-7449,"8235 Maple Street, Wilmington, VM 29085",9/23/63,68900

Funcións

As funcións son moi útiles na programación, permiten reutilizar, ordear e estruturar código para logo empregalo cunha chamada ao nome da función.
Existen dous modos de crear funcións nos *bash shell scripts*:
- Empregando a sentenza *function*:

```
function name {
    commands
}
```

- Simulando outras linguaxes de programación:

```
name() {
    commands
}
```

- **Como empregar funcións.**

Vexamos un exemplo de como empregar funcións en *shell scripts*.

```
#!/bin/bash
function func1 {
    echo "Este é un exemplo dunha función"
}
count=1
while [ $count -le 5 ]
do
    func1
    count=$(($count+1))
done
echo "Este é o final do script"
```

- **Como devolver valores dende unha función.**

O *shell bash* trata as funcións como se fosen *mini-scripts*. Así que veremos que existen tres xeitos distintos de retornar valores dende as funcións:

- O *exit status* por defecto.

· O *exit status* dunha función será o do último comando existente na función.
· Logo de que remate a función, podemos empregar a variable **$?** para determinar o *exit status*.
· O problema é que así non se pode controlar se algún comando faia dentro da función se este non é o último.
· Recorda que se a variable **$?** ten un valor **igual a 0** quere dicir que o comando non faiou e se esta variable ten un valor **distinto de 0** o comando non se executou correctamente por algún motivo.

- Empregando o comando *return*.

· Podemos empregar o comando *return* para que unha función remate cun estado *exit* determinado.
· Este comando nos permite especificar un enteiro (0 - 255) como estado *exit* dunha función.
· Recorda recoller o *exit status* da función "xusto" no intre en que a función remata, pois se executamos logo outro comando calquera o valor da varible **$?** será o dese novo comando e non o da nosa función.
Vexamos un exemplo:

```
#!/bin/bash
function db1 {
        read -p "Escribe un número: " value
        echo "Dobrando ese valor"
        return $(($value*2))
}

db1
echo "O novo valor é $?"
```

- Coa propia saída da función.

· Deste xeito adxudicaremos a saída da función ao valor dunha variable.
Vexamos un exemplo:

```
#!/bin/bash
function db1 {
        read -p "Escribe un número: " value
        echo $(($value*2))
}

result=`db1`
echo "O valor dobrado é $result"
```

- **Pasando parámetros ás funcións.**

Como xa se falou antes, as funcións compórtanse como pequenos *scripts*.
Nas funcións pódense empregar as variable de contorno: $1, $2, $# e, como non, $0 que será o nome da propia función.
Vexamos un exemplo:

```
root@debian:/scripts# cat func.sh
#!/bin/bash
function sumar {
        if [ $# -eq 0 ] || [ $# -gt 2 ]
        then
                echo 1
        elif [ $# -eq 1 ]
        then
                echo $(($1 + $1))
        else
                echo $(($1 + $2))
        fi
}

echo -n "Sumando 10 e 15: "
result=`sumar 10 15`
echo $result
echo -n "Sumando un só número, 23: "
result=`sumar 23`
echo $result
echo -n "Chamando a función sen argumentos: "
result=`sumar`
echo $result
echo -n "Intentando sumar tres números: "
result=`sumar 10 20 30`
echo $result

root@debian:/scripts# ./func.sh
Sumando 10 e 15: 25
Sumando un só número, 23: 46
Chamando a función sen argumentos: 1
Intentando sumar tres números: 1
```

Non debería ser difícil empregar as variables $1 e $2 do *script* e pasarllas como argumentos á función, ¿non?

- **Pasando e devolvendo *arrays* con funcións.**

Vexamos un exemplo de como realizar estas operacións:

```
$ cat prueba.sh
#!/bin/bash
function arraydblr {
local origarray
local newarray
local elements
local i
origarray=(`echo "$@"`)
newarray=(`echo "$@"`)
elements=$(($# - 1))
for (( i = 0; i <= $elements; i++ ))
do
{
        [$i]=$((${origarray[$i]} * 2))
}
done
echo ${newarray[*]}
}
myarray=(1 2 3 4 5 6)
echo "El array original es: ${myarray[*]}"
arg1=`echo ${myarray[*]}`
result=`arraydblr $arg1`
echo "El nuevo array es: ${result[*]}"

$ ./prueba.sh
El array original es: 1 2 3 4 5 6
El nuevo array es: 2 4 6 8 10 12
```

- **Chamadas recursivas a funcións.**

Esta acción trata de que unha función se chama a si mesma mentres que se cumpre algunha condición.
O exemplo máis típico dun algoritmo recursivo é o cálculo do factorial dun número. Así, o factorial de 5 = 5! = 5 * 4 * 3 * 2 * 1 = 120
Empregando a recursividade a ecuación redúcese a:

$$x! = x * (x -1)!$$

```
$ cat prueba.sh
#!/bin/bash
#
function factorial {
if [ $1 -eq 1 ] || [ $1 -eq 0 ]
then
echo 1
else
local temp=$(($1 -1))
local result=`factorial $temp`
echo $(($result * $1))
fi
}

read -p "Introduce un número: " num
result=`factorial $num`
echo "El factorial de $num es: $result"

$ ./prueba.sh
Introduce un número: 5
El factorial de 5 es: 120
```

- **Crear unha librería de funcións**

Se queremos empregar as funcións creadas en varios scripts o mellor é crear unha librería de funcións.
Trátase dun arquivo onde se gardan todas as funcións que nos interesan.
Un exemplo de arquivo de funcións é a seguinte:

```
$ cat myfuncs
#####################
#Librería de Funcións
#####################
#
#Función sumar
function sumar {
echo $(($1 + $2))
}
####################
#Función multiplicar
function multi {
echo $(($1 * $2))
}
###################
#Función dividir
function div {
if [ $2 -ne 0 ]
then
echo $(($1 / $2))
else
echo -1
fi
}
```

Para chamar a este arquivo dende calquera script só temos que empregar o comando **source** ou empregar o seu alias **dot operator**. Vexamos un exemplo que chama o anterior arquivo de funcións:

```
$ cat prueba.sh
#!/bin/bash
#Cargamos o arquivo de funcións
. ./myfuncs

valor1=10
valor2=15

resultado1=`sumar $valor1 $valor2`
resultado2=`multi $valor1 $valor2`
resultado3=`div $valor1 $valor2`

echo "La suma de $valor1 y $valor2 es $resultado1"
echo "La multiplicación de $valor1 y $valor2 es $resultado2"
echo "La división de $valor1 entre $valor2 es $resultado3"

$ ./prueba.sh
La suma de 10 y 15 es 25
La multiplicación de 10 y 15 es 150
La división de 10 entre 15 es 0
```

Captura de sinais: Comando trap

Os sinais son mensaxes que un proceso envía a outro. Posúen nomes e números (dende 1 ata o número que soporte o sistema operativo). Para saber os sinais que soporta un sistema GNU/Linux pódense empregar o comando: **kill -l** ou **trap -l**, os cales darán unha saída na shell similar á seguinte:

```
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

NOTA: *Pode obter máis información sobre os sinais mediante o comando:* ***man 7 signal***

Existen combinacións de teclas que executan sinais, por exemplo:

Combinación de teclas	Sinais	Explicación
Ctrl+C	SIGINT	Interrupción dende o teclado. O proceso debe rematar
Ctrl+Z	SIGTSTP	Proceso parado dende o terminal. O proceso debe deterse.
Ctrl+\	SIGQUIT	Pechado dende o teclado. Soamente debería empregarse cando SIGINT non funciona, posto que, pode deixar ficheiros sen pechar.

NOTA: *Pode obter máis información sobre as combinacións de teclas que executan sinais mediante o comando:* ***stty -a***

En ocasións, interésanos que certos sinais, como **<Ctrl>+C**, non impidan ou interrumpan a execución dos scripts. Neste caso deberemos empregar o comando **trap**, o cal permite capturar sinais e impedir a súa execución polas ordes que consideremos. A súa sintaxe é a seguinte:

```
trap comando sinal1 sinal2 ...
```

onde, *comando* pode ser un comando, un script ou unha función e *sinal1*, *sinal2*, ... poden ser dadas por nome(tamén omitindo os caracteres *SIG* co que comezan tódolos sinais) ou número.

Un exemplo será máis clarificador:

EXEMPLO A: Emprego do comando trap

```
#!/bin/bash
trap impide_corte_script 2
impide_corte_script() {
    echo "A combinación de teclas ^C non funciona"
}
for i in `seq 1 60`
do
    echo $i
    sleep 1
done
```

Captura de teclas: Comando bind

¿Cómo capturar teclas en scripts e darlles un comando? Mediante o comando **bind** podemos capturar combinacións de teclas. Previamente á captura de combinacións de teclas deberiamos comprobar cales existen na shell Bash mediante o comando:

```
bind -p
```

Os arquivos **/etc/inputrc** e **$HOME/.inputrc** conteñen definicións das combinacións de teclas, como por exemplo: que é o que fan as frechas, avpág, repág... e son lidos pola shell Bash cando arranca. A sintaxe do comando **bind** para poder executar, mediante combinación de teclas, comandos é a seguinte:

```
bind '"\eCódigo_teclas":"comandos"'
```

onde, *Código_teclas* son os códigos das teclas e *comandos* son os comandos a executar cando se emprege a tecla ou combinación de teclas. Para saber o código dunha tecla, facer o seguinte:

1. Executar o comando **read**
2. Premer tecla
3. Veremos o código da tecla pulsada, do cal interésanos todos os caracteres agás os 2 primeiros: **^[**

Por exemplo:

1. Executar o comando **read**
2. Premer F8
3. Veremos o código **^[[19~**, do cal interésanos **[19~**

O seguinte exemplo intentarán clarificar o exposto:

EXEMPLO A: Script comandos-bind.sh

```
#!/bin/bash
#set -o emacs #Para activar a edición de liña
bind '"\e[19~":"/sbin/ifconfig eth0\n"' #F8
bind '"\e[21~":"exit\n"' #F10
bind '"\ep":"tail -f /var/log/dmesg\n"' #ESCp
bind '\"\C-h\":"history | grep ssh\n"' #Ctrl+h
```

A execución do script debe facerse mediante a seguinte orde, para que esté activa na shell da execución:

```
source comandos-bind.sh
```

NOTA: *Ter en conta que a execución do comando* ***bind*** *para a combinación de teclas coa tecla* ***Ctrl*** *cambia.*

Se o que se quere é sempre ter estas combinacións de teclas, incorporalas no arquivo **$HOME/.bashrc**

Para máis información ver ligazón Libraría readline

Tarefas programadas

Ver ligazón Planificador de tarefas: cron

Scripts remotos mediante chaves ssh sen emprego de contrasinais

NOTA: Para entender este apartado é necesario previamente ler a ligazón Conexión SSH sen contrasinal

Os scripts pódense executar en local ou en remoto, sendo en remoto maioritariamente onde teñen máis razón de ser. Pero na execución en remoto imos atopar unha serie de problemas a maiores dos que podemos atopar en local.

Problemas na execución dun script en remoto

1. En local na execución xa estamos dentro do sistema pero en remoto temos que autenticarnos.
2. A autenticación en remoto podería levarse a cabo mediante a parella usuario/contrasinal mais na execución dun script non nos interesa parar a execución do mesmo e ter que esperar a por o contrasinal para que continue, co cal é moi interesante poder autenticarse sen contrasinal e executar o script.
3. Autenticarse sen contrasinal: si pero non calquera, ademais interesaranos envíar os datos e comandos do script de forma cifrada por motivos de seguridade.

Solucións:

1. Entón deberiamos pensar en remoto e cifrado: servidor ssh
2. Deberiamos pensar en autenticación sen deter a execución do script á espera dun contrasinal: chaves ssh.

Novo problema:
A solución está clara: servidor ssh mediante chaves pública e privada (cifrado asimétrico), pero esta solución tamén acarrexa un problema : cada vez que nos conectamos a un servidor ssh xenérase e engádese unha entrada ao ficheiro **$HOME/.ssh/known_hosts** , sendo:
$HOME-->A variable do sistema que contén como valor o directorio de traballo do usuario, isto é, contén a casa do usuario.
.ssh--> Cartafol onde se gardan por defecto as chaves e algúns ficheiros empregados na conexión cliente/servidor ssh.
known_hosts-->Arquivo que contén os hosts servidores ssh coñecidos nos que algunha vez estableceuse a conexión ssh.

E cal é o problema, pois o problema radica en que cada vez que conectemos a un novo servidor ssh xerase unha entrada no arquivo known_hosts, pero non se fai de forma automática, senón que o sistema espera unha resposta afirmativa ou negativa á identidade do servidor ssh detendo a execución do script. Se a resposta é afirmativa continúase coa execución do script, mais se é negativa finaliza a execución do script. Entón deberemos parametrizar o **cliente ssh** para que nunca pregunte ante unha nova conexión ata un servidor ssh, isto poderémolo facer de dúas formas:

1. Creando un arquivo **$HOME/.ssh/.config** co seguinte contido:

```
Host *
        StrictHostKeyChecking no
```

   Este arquivo pertencente a un usuario aplícase soamente a ese usuario.
2. Modificando o arquivo **/etc/ssh/ssh_config** pondo o a acción de StrictHostKeyChecking a non, así: **StrictHostKeyChecking no**.
   Este arquivo pertencente ao sistema aplícase a todos os usuarios.

Control dos scripts mediante envíos de correo electrónico: programa email

É moi útil a revisión da execución do script na entrega dun correo electrónico. Para tal finalidade empregarase o paquete email que permitirá empregar o servidor de correo de google para enviar o que desexemos á/s conta/s de correo que determinemos. O único requisito, claro está, é que para empregar o servidor de correo de google debemos posuir unha conta de correo do mesmo.

Instalación

1. Prerequisitos: Instalación dos paquetes: gcc, make, libopenssl-devel e as súas dependencias
2. Descargar o paquete **email-version.tar.bz2** de http://www.cleancode.org/downloads/email/
3. Non é necesario ser usuario root. Proceder:

```
$ tar xvjf email-version.tar.bz2
$ cd email-version
$ ./configure --with-ssl
$ make
```

4. Sendo usuario root:

```
# make install
```

Configuración

Por defecto na instalación xerouse un executable e un ficheiro de configuración, respectivamente:
/usr/local/bin/email e /usr/local/etc/email/email.conf

Exemplo tipo arquivo de configuración /**usr/local/etc/email/email.conf**:

-***NOTA:*** *Pode ser necesario crear o arquivo inexistente: ~/dean.ldif*

```
############################################################
# This is the DEFAULT configuration file for email.
#
# Please CHANGE THE VALUES below to suit your environment.
# Don't forget to set the shell environment variable EDITOR
############################################################

############################################################
# SMTP Server and Port number you use
############################################################
SMTP_SERVER = 'smtp.gmail.com'
SMTP_PORT = '587'

############################################################
# If you'd rather use sendmail binary, specify it and the
# command line switches to use, here.  If you have both
# this option and SMTP_SERVER set, SMTP_SERVER will be of
# higher priority than SENDMAIL.
############################################################
# SENDMAIL_BIN = '/usr/lib/sendmail -t -i'

############################################################
# Your email address: If you'd like To have your name to
```

```
# show in the from field instead of just your email address,
# then keep the format below and edit it to your email
# and name.
############################################################
MY_NAME  = 'Nome do usuario do envío'
MY_EMAIL = 'usuario_do_envio@gmail.com'
#REPLY_TO = ''

############################################################
# If your mail server uses SSL or TLS in order to create
# a secure communication, set this to "true". Only very
# simple TLS is implemented at this time. In other words,
# there is no way to load your own certs just yet.
############################################################
 USE_TLS = 'true'

############################################################
# Signature file and settings: Where is the signature file
# You can comment this out if you do not like signatures
############################################################
SIGNATURE_FILE    = '&/email.sig'

############################################################
# This is where you store your address book... If you don't
# want address book functionality, just comment it out and
# use regular email addresses.
############################################################
ADDRESS_BOOK = '&/email.address.template'

############################################################
# If you would like to have a copy Of your final email saved
# after it is sent, please specify a directory to save
# it in below... If not, just comment this field out and it
# won't save a sent file other wise, it will save it as
# email.sent in the directory below
############################################################
# SAVE_SENT_MAIL = '~'

############################################################
# With email, temporary files are stored with a random name
# Starting with .EM.  You must specify which directory you
# would like these temporary files stored while email is in
# Operation...  After email is done executing, it will remove
# Every temporary file it used.  The most common storage area
# for temporary files system wide is the /tmp directory
# if TEMP_DIR is not set, you can set the environment variable
# TMPDIR. If that is not set, email will just use /tmp
############################################################
 TEMP_DIR = '/tmp'

############################################################
# You should put the absolute path of your GPG binary
# In this variable.  If you do not know the absolute path
# just putting "gpg" will suffice if GPG is in your PATH
# environment variable path.
############################################################
# GPG_BIN = '/usr/bin/gpg'

############################################################
# You can bypass email asking you for a password for gpg
# when you are encrypting or signing emails if you store
# it here.  Keep this safe! Make sure that people can't
# read your personal ~/.email.conf file if you're storing
# your password here!
############################################################
# GPG_PASS =

############################################################
# You can use SMTP_AUTH with email.  You just need to
# specify which type of SMTP AUTH you will use. Email
# support two types of SMTP AUTH: LOGIN and PLAIN.
# Typically, LOGIN is the most used amongst SMTP servers.
# If you are unsure, ask your ISP.
```

```
############################################################
 SMTP_AUTH = 'LOGIN'

############################################################
# Setting your SMTP username is MANDITORY.  Please
# uncomment the option below and set your username in order
# to use SMTP AUTH
############################################################
 SMTP_AUTH_USER = 'usuario_do_envio@gmail.com'

############################################################
# Setting your password is not manditory.  If you do not
# set your password, then email will prompt you for one
# if it sees that you are trying to use SMTP_AUTH.
############################################################
# SMTP_AUTH_PASS =

############################################################
# If you have a v-card, you can have it attached to each
# message by specifying it's location here.
############################################################
## VCARD = "~/dean.ldif"
```

Control na execución dun script bash

Co comando bash podemos empregar parámetros que nos permiten facer un seguimento de control e/ou comprobación do script antes ou durante a execución:
bash -n nome_script.sh: Permite comprobar se existe un erro na execución do script facendo unha simulación da execución do mesmo.
bash -x nome_script.sh: Permite saber que é o que pasa durante a execución do script: que liñas se van executando, que valor toman as variables...

Co comando **set** tamén podemos controlar a execución dos scripts, ben dentro dos scripts ou ben fora deles na shell. Por exemplo, na shell Bash as anteriores opcións mediante **set** serían respectivamente:

```
set -o noexec
set -o xtrace
```

NOTA: *Coa parámetro **-o** actívase a opción e co parámetro **+o** desactívase.*

Dentro dos scripts podemos depurar seccións mediante o comando **set -x** para activar e **set +x** para desactivar:

```
#Inicio da sección a depurar
set -x
comandos a depurar
set +x
# Fin da sección a depurar
```

Por defecto, as opcións **-x** ou **xtrace**, empregan o símbolo **+** ao principio de cada liña, e por cada **+** anidado terase un nivel máis de expansión, e dicir, os comandos executados coas súas opcións, á súa vez son depurados. A variable prompt **PS4** é a que garda o valor do símbolo **xtrace**, por defecto: **+**, e polo tanto pode ser modificado. Un valor moi útil sería que amosase a liña depurada, isto é, a liña do script que fai que cousa. Este valor sería:

```
PS4='$0:$LINENO:'
```

o cal sería convinte exportar:

```
export PS4='$0:$LINENO:'
```

Hai que ter en conta, que antes, cada símbolo **+** era unha expansión, agora soamente a última expansión anidada conterá o nome do script (**$0**, ver Parámetros $) e o número de liña (**$LINENO**), xa que as anteriores conterán o primeiro carácter do nome do script.

Igual, o anterior, verase mellor mediante os seguintes exemplos:

EXEMPLO A: Depuración de todo o script de nome ExemploA.sh co valor PS4='+'

```
#!/bin/bash
conf_eth0() {
/sbin/ifconfig eth0
}
```

```
conf_eth0
```

Resultado da execución:

```
$ bash -x ExemploA.sh
+ conf_eth0
+ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  direcciónHW 00:01:02:03:04:05
          ACTIVO DIFUSIÓN MULTICAST  MTU:1500  Métrica:1
          Paquetes RX:0 errores:0 perdidos:0 overruns:0 frame:0
          Paquetes TX:0 errores:0 perdidos:0 overruns:0 carrier:0
          colisiones:0 long.colaTX:1000
          Bytes RX:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupción:42 Dirección base: 0x2000
```

EXEMPLO B: Depuración de parte do script de nome ExemploB.sh co valor PS4='+'

```
#!/bin/bash
conf_eth0() {
set -x
/sbin/ifconfig eth0
set +x
}
conf_eth0
```

Resultado da execución:

```
$ bash ExemploB.sh
+ /sbin/ifconfig eth0
eth0      Link encap:Ethernet  direcciónHW 00:01:02:03:04:05
          ACTIVO DIFUSIÓN MULTICAST  MTU:1500  Métrica:1
          Paquetes RX:0 errores:0 perdidos:0 overruns:0 frame:0
          Paquetes TX:0 errores:0 perdidos:0 overruns:0 carrier:0
          colisiones:0 long.colaTX:1000
          Bytes RX:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupción:42 Dirección base: 0x2000

+ set +x
```

EXEMPLO C: Depuración de todo o script de nome ExemploC.sh co valor PS4='$0:$LINENO:'

```
#!/bin/bash
conf_eth0() {
/sbin/ifconfig eth0
}
conf_eth0
```

Resultado da execución:

```
$ export PS4='$0:$LINENO:'
$ bash -x ExemploC.sh
ExemploC.sh:5:conf_eth0
ExemploC.sh:3:/sbin/ifconfig eth0
eth0      Link encap:Ethernet  direcciónHW 00:01:02:03:04:05
          ACTIVO DIFUSIÓN MULTICAST  MTU:1500  Métrica:1
          Paquetes RX:0 errores:0 perdidos:0 overruns:0 frame:0
          Paquetes TX:0 errores:0 perdidos:0 overruns:0 carrier:0
          colisiones:0 long.colaTX:1000
          Bytes RX:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupción:42 Dirección base: 0x2000
```

EXEMPLO D: Depuración de parte do script de nome ExemploD.sh co valor PS4='$0:$LINENO:'

```
#!/bin/bash
conf_eth0() {
set -x
/sbin/ifconfig eth0
set +x
}
conf_eth0
```

Resultado da execución:

```
$ export PS4='$0:$LINENO:'
$ bash ExemploD.sh
ExemploD.sh:4:/sbin/ifconfig eth0
eth0      Link encap:Ethernet  direcciónHW 00:1e:33:27:9d:9d
          ACTIVO DIFUSIÓN MULTICAST  MTU:1500  Métrica:1
          Paquetes RX:0 errores:0 perdidos:0 overruns:0 frame:0
          Paquetes TX:0 errores:0 perdidos:0 overruns:0 carrier:0
          colisiones:0 long.colaTX:1000
          Bytes RX:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupción:42 Dirección base: 0x2000

ExemploD.sh:5:set +x
```

Programación modular

Poder reutilizar código é moi interesante e imprescindible, non imos refacer código xa feito, polo tanto pensar en estruturar código que poida ser doado reutlizalo, amplialo, modificalo é moi importante. Un bo plantexamento do código permitirá actuar sobre o mesmo de forma sinxela, polo tanto imos estruturar os scripts como mínimo en 3 ficheiros:

1. **variables.txt**: onde definiremos as variables a empregar na execución do script
2. **funcions.txt**: onde definiremos as funcións a empregare na execución do script
3. **script.sh**: o script a executar, no cal cargaremos as variables de **variables.txt** e as funcións de **funcions.txt**

Exemplos:

Exemplo 1: Execución en local. Crear as contas de sistema de múltiples usuarios

- **NOTA: E moi común en determinadas aplicacións empregar ficheiros para a creación de usuarios, así imos partir dun ficheiro en formato csv, moi empregado para tal finalidade, co cal, partimos do ficheiro de texto de tipo csv de nome usuarios.csv**
- **ESTRUTURA DO SCRIPT**: Como se comentou antes imos dispor de 3 arquivos **na mesma ruta**: script.sh, variables.txt e funcions.txt.

script.sh

```
#!/bin/bash

# IMPORTANTE: Considérase que todos os ficheiros están na mesma ruta.

# Recollemos as variables definidas en variables.txt para poder traballar con elas.
source $PWD/variables.txt

# Recollemos as funcións definidas en funcions.txt para poder traballar con elas.
source $PWD/funcions.txt

#Chamada á función crearusuarios() para crear os usuarios pedidos
crearusuarios
```

variables.txt

```
#Creamos e exportamos a variable NUMT  cuxo valor é o número total de liñas do ficheiro usuarios.csv (11)
export NUMT=`cat usuarios.csv | wc -l`
```

funcions.txt

```
crearusuarios() {
#Creamos a variable L nun bucle contador de 2 ao número total de liñas do ficheiro usuarios.csv
# Comezamos na liña 2 porque a 1 soamente identifica as columnas

for L in $(seq 2 $NUMT)
do
{
   #Creamos a variable user que collerá os valores da primeira columna
    user=`cat usuarios.csv | head -$L | tail -1 | sed -e "s/,/ /g" -e "s/\"//g" | awk '{print $1}'`

   #Creamos a variable contrasinal que collerá os valores da segunda columna
```

```
    contrasinal=`cat usuarios.csv | head -$L | tail -1 | sed -e "s/,/ /g" -e "s/\"//g" | awk '{print $2}'`

   #Creamos a variable consola que collerá os valores da terceira columna
    consola=`cat usuarios.csv | head -$L | tail -1 | sed -e "s/,/ /g" -e "s/\"//g" | awk '{print $3}'`

   #Creamos a variable dir_home que collerá os valores da cuarta columna
    dir_home=`cat usuarios.csv | head -$L | tail -1 | sed -e "s/,/ /g" -e "s/\"//g" | awk '{print $4}'`

   #Empregamos o comando useradd para a creación de usuarios e o comando mkpasswd para a creación de contrasinais
    useradd -m -d $dir_home -s $consola -p `mkpasswd $contrasinal` $user
}
done

}
```

Exemplo 2: Execución en remoto. Crear as contas de sistema de múltiples usuarios

Enlaces interesantes

- Advanced Bash-Scripting Guide - Mendel Cooper en The Linux Document Project
- Advanced Scripting Guide - PDF

-- Felix Díaz, ricardofc [19/03/12]