.::shell scripts::.

d u n e t n a . k e r n e l p a n i c

Tabla de contenidos

0. intro

Este manual, tutorial o como le queráis decir, es el resultado del curso realizado por kernelpanic (hacklab de Barcelona) de programación de shell scripts.

No pretende ser un gran manual sino una ayuda a aquellas personas que, ya sabiendo programar y tienen unos conocimientos genéricos de UNIX, quieren usar el lenguaje que ofrece la shell para hacer scripts que las faciliten la vida ;-)

Dedicamos estos apuntes al CSO Les Naus, desalojado el 9 de diciembre del 2003. Nos quitarán los espacios pero seguiremos luchando, construyendo y compartiendo conocimientos e ilusiones.

NOTA: Podeis encontrar los ejemplos de scripts que tienen su nombre entre paréntesis en la web http://kernelpanic.hacklabs.org. Para cualquier duda o sugerencia podeis enviar un mail a info@kernelpanic.hacklabs.org.

1. conceptos básicos

1.1. variables

La shell nos permite definir variables donde almacenar datos. Tenemos dos zonas de memoria donde podemos definir nuestras variables: el área local y el entorno.

Variables locales: sólo son visibles por la shell donde estamos trabajando, no son visibles por ninguna subshell, es decir, no son visibles por ningún subproceso de la shell.

Variables de entorno: son visibles tanto por la shell donde estamos como por cualquier subshell que abrimos (o por cualquier subproceso de la shell)

NOTA: una variable declarada en un proceso hijo no será visible por su proceso padre (aunque sea de entorno)

COMANDOS	
set	Ver todas las variables definidas
env	Ver todas las variables de entorno definidas
nombre_var=valor_var	Definir variable y asignarle valor
export ***nombre_var=valor_var***	Definir variable entorno y asignarle valor
export ***nombre_var***	Convertir variable local a variable de entorno
unset ***nombre_var***	Liberar una variable

::ejemplo::

```
$ varmeva=3              declaro una variable local
$ echo $varmeva          miro su valor
3
$ bash                   entro en una subshell
$ echo $varmeva          no tiene valor porque es local
$ exit                   volvemos a la shell principal
$ export varmeva         convierto la var. local a var. de entorno
$ echo $varmeva          en la shell principal sigo viendo su valor
3                        3
$ bash                   entro en una subshell
$ echo $varmeva          ahora sí que vemos su valor!!!
3                        (es de entorno)
$ exit
```

Variables predefinidas: Disponemos también de una serie de variables ya definidas que nos pueden ser de gran ayuda para obtener o dejar información genérica.

VARIABLES PREDEFINIDAS	
HOME	Directorio de trabajo actual
PATH	Lugares donde podemos acceder directamente sin escribir el camino
PS1	Prompt primario
PS2	Prompt secundario
BASH	Camino del programa bash
BASH_VERSION	Versión de la bash actual
COLUMNS	Número de columnas en la pantalla
GROUPS	Identificador del grupo principal del usuari@
HISTCMD	Índice en el histórico del comando actual
HISTFILE	Fichero donde se guarda el histórico
HISTFILESIZE	Medida del fichero histórico
HOSTNAME	Nombre de la máquina
LANG	Idioma por defecto, si no se ha especificado en ningún LC_
LC_ALL	Idioma
PID	Identificador del proceso actual
PWD	Camino donde estamos situados
RANDOM	Número aleatorio
SECONDS	Segundos que lleva encendida la máquina
UID	Identificador del usuari@ actual

::ejemplo::

```
$ echo "Mi directorio de trabajo es $HOME y mi id de grupo es $GROUPS"
    Mi directorio de trabajo es /home/kp y mi id de grupo es 1000
```

1.2. sustitución de variables

Sustitución de variables: Es la técnica que utilizaremos para hacer referencia al valor contenido en una variable.

SUSTITUCIÓN DE VARIABLES	
$nombre_var	Hacer referencia al valor de la variable *nombre_var*
${nombre_var}	Hace referencia al valor de la variable *nombre_var*. Las llaves nos delimitan el nombre de la variable.

::ejemplos::

```
1::: $ saluda="hola"            defino variable saluda con valor "hola"
     $ nombre="anna"            defino variable nombre con valor "anna"
     $ echo saluda nombre       muestro por pantalla los valores de las
                                variables...
     saluda nombre              está mal: he puesto los nombre de las
                                vars, no su valor
     $ echo $saluda $nombre     ahora está bien!
     hola anna
```

2::: $ masc=”gat” defino variable masc con valor ”gat”

```
$ echo masculino:$masc femenino:$masca
```
muestro el $masc y $masc seguido de una a

```
masculino:gat femenino:
```
me interpreta masca como una var (vacía)

```
$ masculino:$masc femenino:${masc}a
```
pongo {} para diferenciar el nombre de la variable

```
masculino:gat femenino:gata
```
eso es lo que quería!!!

1.3. sustitución de comandos

Sustitución de comandos: Es la técnica que utilizaremos para sustituir en nombre de un comando por la salida de ésta.

SUSTITUCIÓN DE COMANDOS	
`*comando*` **$(*comando*)**	Sustituir el nombre del comando por su salida

::ejemplos::

1:::
```
# whoami
```
ejecutamos un comando

```
root
```
que nos dice a quien estás conectado

```
# echo Soy el/la whoami
```
si queremos mostrar por pantalla y ponemos el comando...

```
Soy el/la whoami
```
...nos sale literalmente lo que le ponemos

```
# echo Soy el/la `whoami`
```
sustituimos whoami por el valor que devuelve y...

```
Soy el/la root
```
...obtenemos el que queríamos!!!

```
# echo Soy el/la $(whoami)
```
otra manera de hacerlo

```
Soy el/la root
```

2:::
```
$ date +%F; date +%D
```
ejecutamos date con formatos %F y %D

```
2004-10-21
10/21/04
$ aaaammdd=”%F”
```
damos como valores estos formatos a a dos variables

```
$ mmddaa=”%D”
$ echo “Data (aaaa-mm-dd):`date +$aaaammdd`”
```
hacemos sustitución de comandos y de variables...

```
Data (aaaa-mm-dd):2004-10-21
$ echo “Data (mm/dd/aa):`date +$mmddaa`”
Data (mm/dd/aa):10/21/04
```

3::: pequeño script que nos crea una copia de seguridad de nuestro directorio de trabajo:

```
CS=/var/backup-`date +%Y%m%d`.tgz
```
variable con el nombre de la copia de seguridad (ej: /var/backup-20040601)

```
tar -czf $CS /home/nomusu
```
comprimimos y empaquetamos el directorio de trabajo

1.4. caracteres especiales

Caracteres especiales: Hay caracteres que para la shell tienen un significado especial. Existen diferentes técnicas para que la shell ignore este significado o para que lo tenga en cuenta.

COMANDOS	
\	anula el significado especial del carácter que va detrás
' '	anula el significado especial de todos los caracteres que estén dentro de las comillas
" "	anula el significado especial de todos los caracteres excepto: $ \ `` ""

::ejemplos::

1:::
```
$ echo "El "silencio""        queremos mostrar silencio entre ""...
El silencio                   nos interpreta las " de silencio como
                              un carácter especial!
$ echo El \"silencio\"        las hemos de "escapar"
El "silencio"
```

2:::
```
# echo 'Soy el/la $LOGNAME y estoy en $PWD'   con '' ...
Soy el/la $LOGNAME y estoy en $PWD            ...no interpreta los $
# echo "Soy el/la $LOGNAME y estoy en $PWD"   con "" ...
Soy el/la root y estoy en /root               ...sí que los interpreta!
# echo "Soy el/la $LOGNAME y \$PWD: $PWD"     mezclando"" y \ ...
Soy el/la root y $PWD: /root                  ...podemos interpretar o no según nos convenga
```

1.5. redireccionamento e/s

***stdin*, *stdout* y *stderr*:** Hay comandos que aceptan los datos de entrada por lo denominamos entrada estándar (stdin, con descriptor de fichero 0), que es el teclado. También hay comandos que nos dan su salida por la que denominamos salida estándar (stdout, con descriptor de fichero 1) que es la pantalla. Por último, todos los errores que pueda producir un comando se dirigen a la salida de errores (stderr, con descriptor de fichero 2)

Redireccionamiento de E/S: El redireccionamiento de E/S nos permitirá que cogiendo los datos de entrada, salida y error y las redireccionemos hacia otro fichero diferente al donado por defecto (*stdin*, *stdout*, *stderr*)

REDIRECCIONAMENTO E/S	
<	redireccionamento de *stdin*
>	redireccionamento de *stdout* si el fichero no existe lo crea si el fichero existe se carga el contenido
>>	redireccionamento de *stdout* si el fichero no existe lo crea si el fichero existe añade una continuación
2>	redireccionamento de *stderr* si el fichero no existe lo crea si el fichero existe se carga el contenido
2>>	redireccionament d'*stderr* si el fichero no existe lo crea si el fichero existe añade una continuación
1>&2	redireccionar *stdout* a *stderr*
2>&1	redireccionar *stderr* a *stdout*
>&	redireccionar *stdout* y *stderr* a un fichero

::ejemplos::

1::: `$ mail root -s "hack your mind!" < mailpr` mailpr sería un fichero con el mensaje

2::: `$ cat fit1 fit2` fit1 existe, fit2 no existe

```
esto es el contenido del fichero 1
cat: fit2: No existe el fichero o el directorio
```
stdout / *stderr*

`$ cat fit1 fit2 >> copiafit 2>> error.log` añadimos fit1 al fichero copiafit y el error se irá a error.log

3::: `$ cat fit1 fit2 2> fit3 1>&2` (fit1 existe, fit2 no existe) todas las salidas están en fit3 porque los errores van a fit3 y redirigimos *stdout* a *stderr*

`$ cat fit1 fit2 > fit3 2>&1` (fit1 existe, fit2 no existe) todas las salidas están en fit3 porque *stdout* va a fit3 y redirigimos *stderr* a *stdout*

4::: `$ cat fit1 fit2 >& fit3` (fit1 existe, fit2 no existe) todas las salidas están en fit3 porque redirigimos *stdout* y *stderr* a fit3

1.6. filtros

Filtro: Es un programa que recibe datos por *stdin* y saca datos por *stdout*, sin modificar los datos introducidos por *stdin*.

FILTROS	
cat [*nombre_fichero*]	muestra stdin/*nombre_fichero*

FILTROS	
cut -*clista* [*nombre_fichero*] **cut -*flista* -d*sep* [*nombre_fichero*]**	extrae las columnas citadas en *lista* extrae los campos citados en *lista* según el separador *sep* formato de lista: *A*,*B* seleccionar columnas/campos *A* y *B* *A*-*B* seleccionar columnas/campos desde *A* hasta *B* *A*- desde la columna/campo *A* hasta el final -*B* desde el principio hasta la columna/campo *B*
grep [-cinv] *patrón* [*nombre_fichero*]	Búsqueda de las líneas de los ficheros/stdin del patrón determinado -c sólo muestra el número de línea -y ignora mayúsculas/minúsculas -n añade el número de línea -v muestra las líneas que no contienen el *patrón* formato de *patrón* (expresiones regulares básicas): . cualquier carácter simple [] conjunto de caracteres [^] cualquier carácter no incluido en los corchetes [-] rangos * 0 o más ocurrencias de la expresión precedente + 1 o más ocurrencias de la expresión precedente ^*exp* cualquier cadena que comience con *exp* *exp*$ cualquier cadena que acabe con *exp*
head [-*num*] [*nombre_fichero*]	Muestra las primeras num líneas del fichero/stdin (10 por defecto)
sed 's/*expr1*/*expr2*/[g]' **sed -r 's/*expr1*/*expr2*/[g]'**	Sustituye expr1 por expr2 Sustituye expr1 por expr2 con expresiones regulares complejas g substituye todas las ocurrencias
sort [-ndut*sep*] [-k *num*] [*nombre_fichero*]	Ordenar las líneas de *nombre_fichero*/stdin -n ordenación numérica -d no tiene en cuenta caracteres que no sean letras, números o blancos -u no tiene en cuenta las líneas duplicadas -t*sep* especifica un delimitador de campo -k*num* especifica que ordenaremos por el campo *num*
tail [-*num*] [*nombre_fichero*]	Muestra las últimas *num* líneas del fichero/stdin (10 por defecto)
tee [-a] *nombre_fichero1* [*nombre_fichero2*]	Muestra por pantalla nombre_fichero2/stdin y lo escribe en nombre_fichero1 -a en lugar de sobreescribir nombre_fichero1 añade a continuación

FILTROS	
tr *c1 c2* [*nombre_fichero*] **tr -s *c1* [*nombre_fichero*]**	traduce *c1* por *c2* de nombre_fichero/stdin convierte c1 consecutivos en uno solo
uniq [*nombre_fichero*]	convierte diversas líneas iguales consecutivas de *nombre_fichero*/stdin en una de sola
wc [-lwc] [nombre_fichero]	cuenta líneas, palabras y caracteres de *nombre_fichero*/stdin. -l sólo el nombre de líneas -w sólo el nombre de palabras -c sólo el nombre de caracteres

1.7. tuberías (pipelines)

Tubería: Nos permiten redireccionar la salida de un comando como una entrada a otro comando.

COMANDOS	
comando1 | comando2	Redireccionamos *stdout* de *comando1* a *stdin* de *comando2*

::ejemplos::

1::: Mostrar los nombres de l@s usuari@s dados de alta en el sistema y su shell de inicio, ordenados alfabéticamente y separando los dos campos por un tabulador.
$ cat /etc/passwd | cut -f1,7 -d: | sort | tr ":" "\t"

2::: Visualizar un número que sea la suma de las líneas de los ficheros /etc/profile y /etc/hosts
$ wc -l /etc/profile /etc/hosts | tail -1 | cut -f2 -d" "

3::: Contar el número de ficheros (excluyendo directorios) del directorio actual modificados en enero, a la vez que se almacena el nombre de los ficheros obtenidos en otro fichero denominado "enero".
$ ls -l | tr -s " " | cut -f6,8 -d" " | grep ^....-01- | cut -f2 -d" " |tee enero | wc -l

4::: A partir del fichero /etc/passwd crear un fichero denominado migrupo que esté formado por el nombre de usuari@, shell inicial y directorio inicial de l@s usuari@s de tu mismo grupo.
$ cat /etc/passwd | grep ^.*:.*:.*:`id -g`: | cut -f1,7,6 -d: > migrupo

5::: Obtener un fichero denominado procsroot que contenga el identificador (PID) y el nombre (CMD) de todos los procesos que pertenecen a el/la usuari@ root.
$ ps -ef | grep "^root " | tr -s " " | cut -f2,8 -d" " > procsroot

6::: Obtener un fichero denominado propsetc que contenga una lista (sin repeticiones) con los nombres de l@s propietari@s y grupo a los que pertenecen los ficheros y directorios del directorio /etc.

```
$ ls -l /etc | tr -s " " | cut -f3,4 -d" " | sort | uniq > propsetc
```

2. programación de shell scripts

2.1. ejecución de shell scripts

Shell script: No es más que un programa que usa el lenguaje propio de la shell. Hay diferentes maneras de ejecutarlo:

1. En una subshell de la shell activa
   a) $ bash *nombre_script*
   b) $ *./nombre_script* (necesitamos permisos de ejecución)
   c) $ *nombre_script* (necesitamos permisos de ejecución y que el directorio donde está *nombre_script* esté al PATH)

2. Como un programa ejecutable que se ejecuta como un proceso hijo de la shell
   Como primera línea del script pondremos con qué shell queremos que nos lo interprete:
   ```
   #!/bin/bash
   ```
   y ejecutamos el script como en el caso 1

3. En la misma shell activa
   $ source *nombre_script*
   $. *nombre_script*

::ejemplo::

Crea un script de nombre dorm.sh con la siguiente línea:

```
sleep 222
```

Ejecución tipo 1:

```
$ chmod u+x dorm.sh
$ ./dorm.sh

en otra consola veríamos:
$ pstree -p
   ...
   -bash(905)---bash(1542)---sleep(1543)
   ...
```

Ejecución tipo 2:

```
Ahora el script tendrá el siguiente contenido:
#!/bin/bash
sleep 222

$ ./dorm.sh
en otra consola veríamos:
$ pstree -p
   ...
   -bash(905)---dorm.sh(1564)---sleep(1565)
   ...
```

Ejecución tipo 3:

```
$ . dorm.sh
en otra consola veríamos:
$ pstree -p
   ...
   -bash(905)---sleep(1576)
   ...
```

2.2. depuración de shell scripts

Depuración: Cuando hacemos un programa y los resultados que éste produce no son los esperados, es muy útil tener una herramienta que nos permita depurarlo. Es decir, una herramienta que nos permita hacer un seguimiento paso a paso de por donde va el flujo del programa.

DEPURACIÓN	
set -x	activa la depuración
set +x	desactiva la depuración
set -e	Sale inmediatamente si una orden simple acaba sin éxito (con un código de retorno diferente de cero)
set +e	desactiva el salir si una orden simple acaba sin éxito

::ejemplo::

Si tenemos el siguiente script:

```
SALUDA=”hola”
DESPIDE=”adiós”
set -x
echo “$SALUDA $LOGNAME”
echo “Usuari@s Conectad@s”
who
set +x
echo “$DESPIDE $LOGNAME”
```

cuando lo ejecutamos nos depurará las líneas de código que están entre “set -x” y “set +x” y tendremos el siguiente resultado por pantalla:

```
++ echo hola hm
hola hm
++ echo Usuari@s Conectad@s
Usuari@s Conectad@s
++ who
hm   tty1         Jun  3 19:39
++ set +x
adiós hm
```

2.3. comentarios

Comentarios: Cuando programamos hemos de tener presente que no estamos solos en el mundo o, que aunque lo estemos, la memoria nos puede fallar y no entender el programa que nosotr@s mism@s

hemos escrito... En resumen, que hemos de comentar nuestros scripts para hacernos la vida más fácil :)

COMENTARIOS	
# *comentario*	comentar partes de código

2.4. parámetros y variables especiales

Parámetros: Un parámetro de un script serán todos aquellos valores que adjuntamos cuando ejecutamos éste y que queremos poder ver dentro del programa.

$ *./nombre_script parámetro1 parámetro2 ... parámetron*

Podemos hacer referencia a los parámetros pasados según la siguiente tabla:

PARÁMETROS	
$0	nombre del shell-script que se está ejecutando
$*n*	Parámetro pasado al shell-script en la posición *n*
"$*"	expande los parámetros en una cadena: "par1 par2 ..."
"$@"	expande los parámetros en cadenas diferenciadas: "par1" "par2" ...
shift *n*	Desplazar los parámetros en *n* posiciones. Sin parámetro n desplaza 1 posición.

Variables especiales: Tenemos diferentes definidas en cualquier script que podemos usar según nuestras necesidades.

VARIABLES ESPECIALES	
$#	número de parámetros
$$	PID de la shell proceso que se está ejecutando
$!	PID del último proceso ejecutado
$?	Código de retorno del último proceso ejecutado

::ejemplo::

(sistema.sh) script que nos informa de diversos datos del sistema

2.5. instrucciones e/s

Instrucciones de entrada y salida: Son aquellas instrucciones que nos permiten leer datos de *stdin* (entrada) y mostrar datos por *stdout* (salida)

COMANDOS	
read *nombre_var*	entrada por *stdin*
echo [-ne] cadena/ $nombre_var	salida por *stdout* con salto de línea final -e interpreta caracteres con \ \n : salto de línea \t : tabulador ... -n suprime el salto de línea final

::ejemplo::

```
$ read varmia                          leemos una variable de
3                                      teclado
$ echo -e "Mi var es:\t\t$varmia"      miro su valor con un
Mi var es:          3                  formato
```

2.6. operadores

Operadores numéricos: son aquellos que nos permiten operar con números y variables que contienen números.

Operadores lógicos: son aquellos que nos permiten especificar condiciones compuestas

OPERADORES NUMÉRICOS		OPERADORES LÓGICOS	
+	suma	!	no
-	resta	&&	y
*	producto	||	o
/	división		
%	módulo		
\(\)	paréntesis		

2.7. evaluación de expresiones numéricas

En muchas ocasiones tendremos que hacer cálculos numéricos ya sea para mostrar el resultado o bien por almacenarlo en una variable.

EXPRESIONES NUMÉRICAS	
expr *expr_num*	evalúa *expr_num* sacando el resultado por *stdout* (a *expr_num* hemos de separar operadores de operandos con un espacio)
let expr_num ((expr_num))	evalúa *expr_num* (a *expr_num* no hemos de separar operadores de operandos con un espacio, nos sirve para asignar)

::ejemplos::

```
1::: $ expr 3 + 5        calculamos 3+5
     8                   8!
```

```
2::: $ ((a=3+5))          calculamos 3+5 y ponemos el resultado a la
     $ echo $a            variable a
     8                    mostramos el valor de la variable

3::: $ a=1
     $ ((a=$a+1))         incrementamos el valor de a
     $ echo $a            mostramos el valor de la variable
     2
```

2.8. especificación de condiciones

Condición: Para romper el flujo de un programa necesitamos especificar condiciones que nos lo bifurquen hacia un lado o hacia otro. Para especificar condiciones se usa:

test *expr*

o bien

[expr]

A continuación se exponen las condiciones que podemos especificar:

CONDICIONES FICHEROS	
[-e *nombre_fichero*]	true si el fichero existe
[-d *nombre_fichero*]	true si el fichero existe y es un directorio
[-f *nombre_fichero*]	true si el fichero existe y es regular
[-L *nombre_fichero*]	true si el fichero existe y es un enlace simbólico
[-r *nombre_fichero*]	true si el fichero existe y tiene permiso de lectura
[-w *nombre_fichero*]	true si el fichero existe y tiene permiso de escritura
[-x *nombre_fichero*]	true si el fichero existe y tiene permiso de ejecución
[*nombre_fichero1* -nt *nombre_fichero2*]	true si fichero1 es más nuevo que fichero2
[*nombre_fichero1* -ot *nombre_fichero2*]	true si fichero1 es más antiguo que fichero2
CONDICIONES CADENAS	
[*cadena*]	true si no es la cadena vacía
[-n *cadena*]	true si la longitud de *cadena* es diferente de 0
[-z *cadena*]	true si la longitud de *cadena* es 0
[*cadena1* = *cadena2*]	true si *cadena1* y *cadena2* son iguales
[*cadena1* != *cadena2*]	true si *cadena1* y *cadena2* son diferentes
CONDICIONES ENTEROS	
[*num1* -eq *num2*]	true si *num1* y *num2* son iguales
[*num1* -ne *num2*]	true si *num1* y *num2* son diferentes
[*num1* -gt *num2*]	true si *num1* es más grande que *num2*
[*num1* -ge *num2*]	true si *num1* es más grande o igual que *num2*
[*num1* -lt *num2*]	true si *num1* es más pequeño que *num2*
[*num1* -le *num2*]	true si *num1* es más pequeño o igual que *num2*

2.9. estructuras alternativas

Estructuras alternativas: son aquellas que nos permiten ejecutar un trozo de código según si se cumple o no una condición.

ESTRUCTURA if	ESTRUCTURA case
if *condicion1* then *instrucciones* elif *condicion2* then *instrucciones* else *instrucciones* fi	case *nombre_var* in *patron1*) *instrucciones*;; *patron2*) *instrucciones*;; ... *patronn*) *instrucciones*;; *) *instrucciones*;; esac

::ejemplos::

1::: (existrc.sh) script que comprueba la existencia del fichero .bashrc

2::: (propsfit.sh) script que comprueba si existe el fichero pasado por parámetro y, si existe, nos da sus propiedades

3::: (backup.sh) script que crea una copia de seguridad del directorio que le pasamos como parámetro y la guarda en /var/backups con nombre de formato backup-aaaammdd.tgz

4::: (soinifi.sh) script pensado para poner en /etc/init.d, que nos reproducirá un sonido al entrar al sistema y un sonido al salir.

2.10. estructuras iterativas

Estructuras iterativas: son aquellas que nos permiten ejecutar diversas veces un trozo de código.

ESTRUCTURA while	ESTRUCTURA until	ESTRUCTURA for
while *condicion* do *instrucciones* done	until *condicion* do *instrucciones* done	for *nombre_var* in *lista_valores* do *instrucciones* done

::ejemplos::

1::: (rename.sh) script que renombra diversos ficheros a la vez en el directorio actual.

2::: (llistafit.sh) script que acepta como argumentos nombres de ficheros y muestra el contenido de cada uno de ellos precedido de una cabecera con el nombre del fichero.

3::: (estatdporm.sh) script que realiza un estudio de todo el árbol de directorios y ficheros a partir del directorio pasado como parámetro de forma que obtenemos la siguiente info:

Número de ficheros que disponemos con permiso de lectura
Número de ficheros que disponemos con permiso de escritura
Número de ficheros que disponemos con permiso de ejecución
Número de ficheros que no disponemos con permiso de lectura
Número de ficheros que no disponemos con permiso de escritura
Número de ficheros que no disponemos con permiso de ejecución

4::: (grafica.sh) script que recibe como argumentos números comprendidos entre 1 y 75. Dará error en caso que algún argumento no esté dentro del rango y acabará sin hacer nada. En caso contrario generará una línea para cada argumento con tantos asteriscos como indique su argumento.

5::: (onesta.sh) script que busca la presencia del comando pasado como argumento en alguno de los directorios referenciados a la variable PATH, señalando su localización y una breve descripción del comando.

6::: (fitpar.sh) script que pone cada una de las palabras pasadas por parámetro en un fichero. Estos ficheros se denominarán palabra0, palabra1,... respectivamente.

7::: (sequencia.sh) script que cuenta por ti :-)

8::: (binbash.sh) script que añade la línea #!/bin/bash al principio de los ficheros que le pases como parámetro, si es que ya no lo tiene (para gente con poca memoria...)

3. una poco más...

3.1. tablas

Tabla: es una estructura de datos compuesta.

va10	va11	va12	va13	va14	va15	va16	va17	va18	va19
0	1	2	3	4	5	6	7	8	9

arr

A continuación podemos ver como definirla y como hacerle referencia:

TAULES	
***nombre_arr*=(*val1 val2 ... valn*)**	declaración y asignación iniciales
declare -a *nombre_arr*	declaración de una tabla. Útil para hacer asignaciones dinámicas.
nombre_arr*[*index*]=*val	asignación de *val* al elemento *index* de *nombre_arr*
${*nombre_arr*[*index*]}	Hace referencia al elemento situado en *index* de la tabla *nombre_arr*

Podemos hacer referencia a elementos y características de la tabla con las siguientes expansiones:

EXPANSIONES	
${*nombre_arr*[#]}	número de elementos de la tabla
${*nombre_arr*[@]}	lista los elementos de la tabla tratados cada uno de ellos como una cadena
${*nombre_arr*[*]}	lista los elementos de la tabla tratados como una única cadena
${#*nombre_arr*[*index*]}	longitud de ${*nombre_arr*[*index*]}

::ejemplo::

(gen_menu.sh) script que admite como argumentos parejas formadas por 'descripción' y 'comando' y que construye un menú de opciones donde cualquiera de los comandos puede ser ejecutada seleccionando la descripción correspondiente.

3.2. funciones

Definición de una función: Para definir una función tenemos dos posibilidades:

```
nombre_funcion(){
   ...
   instrucciones
   ...
}
```

```
function nombre_funcion{
   ...
   instrucciones
   ...
}
```

Parámetros: Para hacer referencia a los parámetros que nos puedan llegar a la función lo haremos como hemos explicado en el apartado 2.4. ($1, ..., $n,...). Todos los parámetros se pasan por valor, es decir, cuando volvemos de la función el valor de los parámetros no habrá cambiado.

Retorno: Podemos hacer que las funciones retornen un valor.

RETORNO	
return *valor*	interrumpe la función asignando un valor al código de retorno de la función

::ejemplos::

1::: (menu.sh) menú con diferentes opciones

2::: (gen_aniv.sh) Suponemos que tenemos un fichero denominado "cumpleaños.txt", y cuyas las líneas tienen el siguiente formato:

nombre:fechaaniversario

Realizamos un script denominado "generacumple" que te genere el fichero que tenemos que pasar al comando crontab para que el día antes de cada fecha de aniversario recibas un mail que tenga la siguiente información:

...
Subject: Recordatorio Cumpleaños
...
Mañana día [data] es el cumpleaños de [nombre]. No olvides felicitarl@.

3.3. expresiones regulares

Expresiones regulares: son una herramienta para buscar coincidencias entre un texto y un patrón. La explicación dada aquí esta basada en las expresiones regulares estilo Perl.

COMODÍN	
.	cualquier carácter
CLASES DE CARACTERES	
[*llista*]	alguno de los caracteres de la lista
[*min-max*]	caracteres comprendidos entre *min* y *max*
[^*llista*]	Cualquier carácter que no esté en la lista
\w	palabra
\W	contrario de \w
\s	espacios, tabuladores,... (espacio, \t, \n, \r)
\S	contrario de \s
\d	dígito
\D	contrario de \d
\A	comenzar a mirar por el principio de la cadena
\Z	comenzar a mirar por el final de la cadena

\b	concordar con los límites de la palabra
\B	contrario de \b
[[:alnum:]]	alfanuméricos
[[:alpha:]]	alfabéticos
[[:cntrl:]]	caracteres de control
[[:digit:]]	dígitos
CLASES DE CARACTERES	
[[:graph:]]	gráficos
[[:lower:]]	minúsculas
[[:print:]]	caracteres imprimibles
[[:punct:]]	caracteres de puntuación
[[:space:]]	espacios
[[:uppor:]]	mayúsculas
[[:xdigit:]]	dígitos hexadecimales
ALTERNATIVAS	
alter1*|*alter2	Puede aparecer *alter1* o *alter2*
CUANTIFICADORES	
?	0 o 1 ocurrencia de la expresión precedente
*	0 o más ocurrencias de la expresión precedente
+	1 o más ocurrencias de la expresión precedente
{m}	*m* ocurrencias de la expresión precedente
{m,}	*m* ocurrencias o más de la expresión precedente
{m,n}	de *m* a *n* ocurrencias de la expresión precedente
??, *?, +?, {}?	el mismo pero no intenta coger el máximo de caracteres (no 'greedy')
ANCLAS	
^*expr*	cualquier cadena que comience con *exp*
***expr*$**	cualquier cadena que acabe con *exp*
GRUPOS Y REFERENCIAS	
(*expr*)	hacer un grupo para poder referenciarlo después
n	referencia al grupo n-ésimo
(?:*expr*)	grupo sin referencia

:::ejemplos:::

1::: [a-z] letras minúsculas
[A-Z] letras mayúsculas
[0-9] números
[,'¿!¡;:\.\?] caracteres de puntuación
[A-Za-z] letras del alfabeto
[a-Z] el mismo que el anterior
[A-Za-z0-9] caracteres alfanuméricos
[^a-z] todo menos las letras minúsculas
[^0-9] todo menos números

2::: a.b axb aab abb aSb a#b ...

```
a..b          axxb aaab abbb a4$b ...
[abc]         a b c
[aA]          a A
[aA][bB]      ab Ab aB AB
[0-9][0-9][0-9]     000 001 009 010 019 100 999
[0-9]*        cadena_vacía 0 1 9 00 99 123 456 999 9999
[0-9][0-9]*         0 1 9 00 99 123 456 999 9999 99999 99999999
^.*$          cualquier línea completa
[0-9]+        0 1 9 00 99 123 456 999 9999 99999 99999999 ...
[0-9]?        cadena_vacía 0 1 2 3 ...
^a|b          a b
(ab)*         cadena_vacía ab abab ababab ...
^[0-9]?b      b 0b 1b 2b 9b
([0-9]+ab)*         cadena_vacía   1234ab   9ab9ab9ab   9876543210ab
99ab99ab
```

3::: `([0-9]) \1 ([0-9])` 3 3 5

```
el cargol|gat       el cargol el gat
ho{1,4}la           hola hoola hooola hoooola
(slashdot|barrapunto)       slashdot barrapunto
s[aeou]c            sac, sec, soc, suc
-?[0-9]+            64, -2, 65536, -512
0x[0-9A-F]{2}               0xF0, 0x3B, 0xAA
go{2,5}ggle                 google, gooogle, goooogle, gooooogle
\bkernel\b          kernel, pero no kernelpanic o microkernel
\d+\.\d+\.\d+\.\d+ 192.168.0.1 (dirección ip)
(?:\d+\.){3}\d      192.168.0.1 (dirección ip)
(?:[\w-]+\.)+(?:net|com|org)        ii.jocs.fractals.net
s/<y>(.*?)<\/y>/<b>\1<\/b>/g        itálicas   =>   negritas   (en
documento html)
s/(\d+)\.(\d+)\.(\d+)\.(\d)+/\4.\3.\2.\1/  192.168.0.1       =>
1.0.168.192
(\d+\.\d+\.\d+\.\d+)(?:\n)*\1(?:\n)*\1     3 veces la misma ip
(\d+\.\d+\.\d+\.\d+\n*){3}                 el   mismo   que   el
anterior
```

3.4. comandos muuuuy útiles

3.4.1. sed

sed: es un editor no interactivo, recibe la entrada por *stdin* o por un fichero, hace las operaciones necesarias y saca el resultado por *stdout*. De todas sus funcionalidades detallaremos aquí tres de las más potentes: escribir, borrar y sustituir. La sintaxis es:

sed [*opciones*] {*OPERADOR* | *script fichero*}

OPCIONES	
-r	para poder usar expresiones regulares complejas. Por ejemplo, para poder hacer referencia a grupos: podemos utilizar \n para hacer referencia al *n*-ésimo grupo, donde cada grupo se delimita por (expr).
-f	si queremos usar un script en vez de un OPERADOR
-n	no escribe por pantalla la línea que está tratando

OPERADORES BÁSICOS	
***rang*p**	imprime las líneas de *rang*
***rang*d**	borra las líneas de *rang*
s/*patron1*/*patron2*/ [*modificador*]	sustituye la primera ocurrencia de *patron1* por *patron2*
***rang*/s/*patron1*/*patron2*/ [*modificador*]**	sustituye la primera ocurrencia de *patron1* por *patron2* de las líneas de *rang*

MODIFICADORES	
g	sustituye a todas las ocurrencias de las líneas, no sólo a la primera
y	no distingue entre minúsculas y mayúsculas

:::ejemplos:::

8d	Borra la 8a línea
/^$/d	Borra todas las líneas en blanco
/kernel/p (con -n)	Imprime las líneas con la palabra kernel
s/Windows/Linux/ de cada línea	Sustituye "Linux" por la primera instancia de "Windows" en
s/Windows/Linux/g "Windows"	Sustituye "Linux" por cada instancia de
s/ *$// línea	Borra todos los espacios del final de cada
s/00*/0/g único 0	Reduce todas las secuencias de 0's a un
/Windows/d Windows	Borra todas las líneas donde aparece
s/Windows//g borrar la	Borra todas las palabras Windows, sin línea

:::ejemplos:::

```
sed -r 's/(La|El) (.*) es.*/\2/'
```

Extrae el sujeto (sin artículo) de frases del tipo *"La pelota pequeña es de un color muy extraño"*

```
sed -r 's/(La|El) ([^ ]*) (.*) (es.*)/\1 \3 \2 \4/'
```

Pone la primera palabra del sujeto al final de este

3.4.2. awk

awk: es un lenguaje para procesar texto con muchas utilidades. Aquí veremos una pequeña introducción con aquellas que nos puedan ser más útiles para hacer shell scripts.

awk [-F *separador*] [-v *var=valor*] [-f *prog_awk*|'*prog_awk*'] [*ficheros*]

Un programa awk puede tener una o diversas líneas. Cada una de ellas tendrá la siguiente estructura:

expresión { *acción1*; *acción2*; ... }

Acciones: Si la evaluación de la expresión es positiva se ejecuta/n la/las acción/es indicada/s

ACCIONS	
print	Imprime lo que le piden, si ponemos una , imprime un espacio en blanco

:::ejemplo:::

```
$ echo -e "\nun\ndos\ntres" | awk '{ print "Yo uso"; print "Linux" }'
Yo uso
Linux
Yo uso
Linux
Yo uso
Linux
```

Por cada línea, como que la expresión es cumple, nos hace las acciones: imprimir "Yo uso" y imprimir "Linux"

Campos: Cada línea del fichero que le pasemos a awk está formada por campos (si no le pasemos un delimitador, este será el espacio en blanco).

CAMPOS	
$*n*	n-ésimo campo

:::ejemplo:::

```
$ echo "un dos tres" | awk '{ print $1,$2 }'
un dos
```

Imprimimos el primer y el segundo campo, separados por un espacio

Variables: Con awk podemos definir variables como cualquier otro lenguaje de programación.

Tenemos además, un conjunto de variables predefinidas por el propio awk:

VARIABLES PREDEFINIDAS	
FS	contiene carácter delimitador de campos (por defecto, espacio en blanco)
NF	contiene el número total de campos del registro que se está procesando
RS	contiene el carácter que indica donde se acaba cada registro (por defecto, \n)
NR	contiene el número de orden del registro que se está procesando
OFS	contiene el separador de campos para la salida generada
ORS	contiene el carácter de final de registro para la salida generada

:::ejemplos:::

1::: `$ echo –e "Uso el sistema\noperativo denominado GNU/Linux" | awk –v OFS="\t" –v ORS="\t" '{print $1, $2, $3}'`

```
Uso el sistema operativo denominado GNU/Linux
```

Como salida, la separación de campos y la de registros es un tabulador

2::: awk.txt:

```
{ print "Procesando la línea", NR }
```

imprimimos el número de línea que estamos procesando

```
NF > 1 { print "La línea tiene más de una palabra" }
```

miramos si tiene más de un campo

```
$1 == "GNU/Linux" {print "La primera palabra es GNU/Linux"}
```

miramos si el primer campo es GNU/Linux

```
$ echo -e "GNU/Linux es un SO\nMe gusta
  GNU/Linux\nGNU/Linux es mi SO" | awk -f awk.txt
Procesando la línea 1
La línea tiene más de una palabra
La primera palabra es GNU/Linux
Procesando la línea 2
La línea tiene más de una palabra
```

Operadores: Para poder operar tenemos los siguientes operadores:

OPERADORES	
+	suma
-	resta
*	multiplicación
/	división
%	residuo
^	exponenciación
espacio	concatenación
!	negación
var++	incrementar var en 1
var--	decrementar var en 1
>	mayor que
>=	mayor o igual que
<	menor que
>=	menor o igual que
==	igual
!=	diferente
?:	estructura if-else

Plantillas: En vez de la expresión podemos poner unas plantillas que nos sirven para que se ejecute alguna/s acción/es antes y/o al finalizar el programa:

PLANTILLAS DE PRÓLOGO Y EPÍLOGO	
BEGIN	La acción se ejecutará al principio
END	La acción se ejecutará al final

:::ejemplo:::

sumaawk.txt:

```
BEGIN { total=0 }          inicializamos la variable total a 0
{ total=total+$1 }         aumentamos total con el primer campo
END {print "El total es",total}  imprimimos el resultado final
```

```
$ echo -e "12\n15\n10" | sumaawk -f awk.txt
El total es 37
```

Estructuras de control: Disponemos también de las diferentes estructuras de control de cualquier lenguaje de programación. En el caso de awk se han heredado la sintaxi del lenguaje de programación C.

ESTRUCTURA if	ESTRUCTURA for	ESTRUCTURA while	ESTRUCTURA do/while

if(*cond*) {
instrucciones
}
[else {
instrucciones
}]
	for (inic;cond;instr){
	instrucciones
	}
		while(cond){
		instrucciones
		}
			do{
			instrucciones
			} *while(cond)*

:::ejemplos:::

1::: posnegawk.txt:

```
{
   if( $1 > 0 ){                    miramos si el primer campo es
      print $1,"es positivo";       positivo...
   }
   else{                            ...o negativo
      print $1,"es negativo";
   }
}
```

```
$ echo -e "12\n-15\n10" | awk -f posnegawk.txt
12 es positivo
-15 es negativo
10 es positivo
```

2::: tablaawk.txt:

```
{
   print "\nTabla del",$1
   print "-------------"
```

```
   for( y = 1; y < 11; y ++ ){
      j = $1 * y;
      print y,"*",$1,"=",j;
   }
}

$ echo -e "2\n3" | awk -f awk.txt

tabla del 2
-------------
1 * 2 = 2
2 * 2 = 4
3 * 2 = 6
4 * 2 = 8
5 * 2 = 10
6 * 2 = 12
7 * 2 = 14
8 * 2 = 16
9 * 2 = 18
10 * 2 = 20

tabla del 3
-------------
1 * 3 = 3
2 * 3 = 6
3 * 3 = 9
4 * 3 = 12
5 * 3 = 15
6 * 3 = 18
7 * 3 = 21
8 * 3 = 24
9 * 3 = 27
10 * 3 = 30
```

Funciones: Tenemos además diversas funciones predefinidas:

FUNCIONES	
length(*s*)	Devuelve la longitud de *s* en bytes
rand()	Devuelve un número al azar entre 0 y 1
srand()	Inicia la semilla de generación al azar
int(*var*)	Devuelve *var* convertido en un entero
substr(*s*,*m*,*n*)	Devuelve la subcadena de *s* comenzando por la posición *m* con una longitud de *n*
index(*s*,*t*)	Posición de *s* donde aparece *t* o 0 si no está
match(*s*,*r*)	Posición de *s* donde se cumple la expresión *r*
split(*s*,*a*,*fs*)	Devuelve *s* en elementos separados por *fs* a la tabla *a*
sub(*r*,*t*,*s*)	Cambia en s la cadena t por r
gsub(*r*,*t*,*s*)	Cambia en s la cadena t por r en todas las ocurrencias
sprintf(*f*,*e*,*e*,...)	Imprime con formato
system(*cmd*)	Ejecuta *cmd* y devuelve el código de retorno
tolower(*s*)	Devuelve *s* en minúsculas
touppor(*s*)	Devuelve *s* en mayúsculas
getline	Fuerza una lectura de fichero

:::ejemplo:::

```
echo -e "hola maria \nhola joan" | awk
'{sub("hola","adeu");print}'
adeu maria
adeu joan
```

3.5. un poco de color y movimiento...

La *American National Standards Institute* (*ANSI*) proporciona una serie de secuencias de caracteres para poder realizar determinadas tareas bajo el S.O.. Aquí veremos las secuencias que nos permiten formatear los caracteres de la pantalla (negrita, color,...), mover el cursor, entre otros. Para conseguirlo tendremos que poner:

SECUENCIA_ESCAPE+SECUENCIA_CONTROL

SECUENCIAS DE ESCAPE	
^[	Si hacemos un texto directo y queremos verlo con cat Para obtener este carácter con el editor VI: ctrl+x ESC
\E	Si hacemos un echo -e “”

SECUENCIAS DE CONTROL	
ATRIBUTOS DEL TEXTO	
[0m	Texto normal (reset)
[1m	Negrita
[3m	Cursiva
[4m	Subrayado
[5m	Intermitente
[7m	Inverso
[22m	No negrita
[23m	No cursiva
[24m	No subrayado
[25m	No intermitente
[27m	No inverso
COLORS	
[XX;YYm	XX: color de letra YY: color de fondo **COLORES DE LETRA** / **COLORES DE FONDOS** **30** negro / **40** negro **31** rojo / **41** rojo **32** verde / **42** verde **33** amarillo / **43** amarillo **34** azul / **44** azul **35** magenta / **45** magenta **36** cian / **46** cian **37** blanco / **47** blanco Si combinamos estos valores con 0 (texto normal) y 1 (negrita) obtenemos colores fuertes o suaves.
MOVIMIENTO DEL CURSOR	
[*x*A	Subir *x* líneas
[*x*B	Bajar *x* líneas
[*y*C	Ir a la derecha y espacios
[*y*D	Ir a la izquierda y espacios
[*y*;*x*H **[*y*;*x*f**	Posicionar el cursor a *y*,*x*
[?6h	Posicionar en la parte superior izquierda
[s	Grabar cursor y atributos
[u	Restaurar atributos grabados
CONTROL DE PANTALLA	
[2J	Limpiar pantalla
#8	Llenar con E's
[K	Borrar hasta final de línea
[?5h	Enciende inverso
[?5l	Apaga inverso

::ejemplos::

1::: Texto rojo claro sobre verde subrayado y volver a dejar los colores normales

```
echo -e "\E[1;4;31;42mHOLA\E[0m"
```

2::: Posicionar el cursor en la fila 15 columna 40 con fondo amarillo, texto azul (colores suaves) e intermitente. Poner nuevamente el cursor 5 líneas más abajo
echo -e "\E[2J\E[15;40H\E[1;5;34;43mHOLA\E[0m\E[5B"

3::: (testsuma.sh) Script que propone diez sumas y te da la nota que sacas. Se usan diferentes colores y posicionamientos del cursor.

4. anexos

4.1. ejercicios

:::VARIABLES:::

1. Usando variables:
   a) Asignar el valor “Lunes” a la variable DIA1, el valor “Martes” a la variable DIA2 y así consecutivamente hasta la variable DIA7.
   b) Mostrar el valor de todas las variables por verificar.
   c) Usando estas variables obtener la salida:
      Semana: Lunes Martes Miércoles Jueves Viernes Sábado Domingo
   d) Usando sólo las variables definidas, cargar en la variable SEMANA la lista de días separados por espacios.
2. Usando variables de entorno, mostrar:
   a) camino del directorio de trabajo del usuari@
   b) nombre del login del usuari@
   c) nombre de la terminal en uso del usuari@
   d) nombre del intérprete de comandos actual
   e) camino o caminos de búsqueda de ejecutables
   f) todas las variables de entorno
3. Usando variables:
   a) Declara la variable VIAS como el valor “/usr/doc:/var/lib/dpkg”. Muestra su contenido.
   b) Agregar a la variable VIAS el directorio /usr/doc/HOWTO al final y /usr/DOC/FAQ al principio (separados todos los directorios con :)
4. Cuál es la salida de los siguientes comandos?
   ```
   echo $LOGNAME
   echo “$LOGNAME”
   echo '$LOGNAME'
   echo \”$LOGNAME\”
   echo “Mi login es $LOGNAME”
   echo 'Mi login es $LOGNAME'
   ```
5. Usando variables:
   a) Visualiza el contenido de la variable donde se guarda el prompt. Guarda su valor en una variable local que se denomine SAVE.
   b) Cambiar el prompt para que se muestre así:
      UNIX llest$
   c) Haz que el prompt vuelva a tener su valor inicial usando la variable SAVE.
6. Usando variables:
   a) Inicializar la variable VAR1 con la cadena “shell bash 1”. Mostrar su contenido. Invocar la shell csh. Qué valor tiene ahora la variable VAR1? Por qué?
   b) Sal de csh. Qué valor tiene ahora VAR1? Por qué?
   c) Haz que la variable VAR1 sea una variable de entorno y repite los pasos a) y b) Qué ha pasado?
   d) Si en lugar de invocar csh invocamos bash. Pasa lo mismo?
7. Añade el directorio /usr/prog/bin a la variable PATH. Cómo

harías para comprobar que funciona bien el nuevo PATH?
8. Crea la variable MID que contenga los valores de las variables HOME y LOGNAME separados por dos puntos.

:::REDIRECCIONAMIENTO:::

9. Crea el archivo líneas con el texto “archivo líneas” como contenido. Agregar al archivo creado una línea de texto, por ejemplo “Esta es la línea 1” sin usar ningún editor de texto.
   a) Con el comando cat, muestra por pantalla el contenido del archivo /etc/services
   b) Escribe el comando cat redireccionando la entrada estándar desde el archivo /etc/services y la salida estándar hacia el archivo servicios.txt. Visualiza servicios.txt
   c) Usando echo, crea el archivo errores.txt con el contenido “archivo d Errores”
   d) Con el comando cat intenta mostrar el archivo noexiste.xxx sin redireccionar entrada estándar, pero redireccionando la salida estándar hacia noexiste.txt y el error estándar hacia agregar al archivo errores.txt. Visualiza noexiste.txt y errores.txt.
10. Utiliza la siguiente sentencia " $ cat f1 f2 ". El fichero f1 ha de existir y el f2 no ha de existir. Observa la salida por el monitor.
   a) Consigue que la salida de errores vaya a un fichero denominado errores.
   b) Consigue que el contenido del fichero que existe se copie a un fichero denominado f3 (además del que has conseguido en el apartado a)).
11. Usando redireccionamento, crea un fichero que contenga la siguiente información:
   - Una línea con el contenido “Informe del sistema”
   - Dos líneas en blanco
   - El calendario del mes actual
   - La fecha actual
   - El tipo de ordenadores que estamos usando
   - Los/las usuari@s que están conectad@s

:::FILTROS I PIPELINES:::

12. Dado el siguiente fichero denominado text:

a
aa
ab
aba
aaa
abab
abba
La cadena a es capicua
También es capicua la cadena aa
La cadena ab no es capicua
En una cadena capicua su final refleja su principio
Si concatenas una cadena y su reflejo el resultado es capicua
A
AA

ABA

a) Listar las líneas que contengan una letra ‘a’
b) Listar las líneas que contengan dos letras ‘a’ consecutivas
c) Listar las líneas que no contengan la letra ‘a’
d) Listar las líneas que no contengan letras mayúsculas
e) Listar las líneas que comiencen por la letra ‘a’
f) Listar las líneas que comiencen por una letra minúscula
g) Listar las líneas que acaben con la cadena ‘capicua’

13. Listar todos los archivos del directorio actual sin que aparezcan los directorios.
14. Mostrar todos los procesos del sistema con UID root.
15. Mostrar todos los procesos del sistema con UID diferente de root.
16. Extraer los nombres de l@s usuari@s del resultado de la orden “who”.
17. Extraer los campos 1 y 3 del resultado de la orden “who”.
18. Extraer los permisos de todos los ficheros del directorio $HOME.
19. Listar el/la propietari@ y medida de todos los ficheros del directorio $HOME.
20. El fichero /etc/passwd del servidor Linux contiene información de tod@s las/los usuari@s que tienen una cuenta en la máquina. Cada línea corresponde a un/una usuari@ diferente y en ella aparecen los siguientes campos delimitados por “:”

nombre de usuari@
Password código introducido en forma de x
Identificador del/ la usuari@
Identificador del grupo al cual pertenece el/la usuari@
Información respecte al/ la usuari@ (nombre, apellidos, etc.)
Directorio de trabajo
Intérprete de comandos utilizado

Un ejemplo de la línea correspondiente a un/a usuari@ en este fichero sería:

maria:x:210:204:Maria Sanchez:/home/maria:/bin/bash

a) Comprueba si existe o no una entrada de tu usuari@ este fichero.
b) Lista tod@s l@s usuari@s del mismo grupo que tú que existen en este fichero.
c) Lista los identificadores de usuari@.
d) Lista los shells usados por l@s usuari@s que tengan el mismo grupo que tú.
e) Lista el campo 1 y los campos 3 a 5.

21. Mostrar el contenido del fichero /etc/passwd ordenado alfabéticamente.
22. Ordenar alfabéticamente el resultado de la orden “who”
23. Extraer los nombres de l@s usuari@s del resultado de la orden “who” y ordenarlos alfabéticamente.
24. Extraer los nombres de l@s usuari@s del resultado de la orden “who”, ordenarlos alfabéticamente y eliminar los nombres repetidos.
25. Listar el tamaño y los nombres de los ficheros (excluyendo los directorios) del directorio actual ordenados por tamaño.

26. En el fichero /etc/group se especifican los grupos existentes en el sistema con su identificador y sus componentes. El formato de cada línea es el siguiente:

nombre_grupo:id_grupo:comp1,comp2,...compn

a) Muestra por pantalla las líneas de los grupos donde está tu usuari@.
b) Muestra por pantalla en cuántos grupos estás.
c) Muestra por pantalla las líneas de los grupos donde estás tú y algún/a usuari@ más.
d) Ahora, haz que sólo salgan los nombres de los grupos donde está tu usuari@.
e) Almacena estos nombres de grupos en una variable que se llame MISGRUPOS
f) Muestra por pantalla el primer y tercer campo (nombre y identificador) de la línea correspondiente a tu usuari@ en el archivo /etc/passwd. El nombre e identificador estarán separados por ":".
g) Muestra el mismo resultado que en el apartado f) pero ahora el nombre y el identificador estarán separados por un espacio en blanco.
h) Asigna la salida anterior a una variable denominada IDENTIDAD.
i) Crea una variable que se llame YO con el contenido de la variable IDENTIDAD y de la variable MISGRUPOS, separados con un $. Muestra el resultado por pantalla.

27. Queremos imprimir por pantalla "Hola <num_id>" donde num_id es tu identificador. Hazlo de tres maneras diferentes:
a) Usando el comando id con opciones
b) Usando el comando id sin opciones
c) Usando el archivo /etc/passwd

28. Usando el comando date sin ninguna opción haz que:
a) Salga el día (núm) por pantalla. Almacenar el mes en la variable DIA.
b) Salga el mes por pantalla. Almacenar el mes en la variable MES.
c) Salga el año por pantalla. Almacenar el mes en la variable AÑO.
d) Salga por pantalla "Hoy es día <num_día> del mes <mes>del año <año>", usando las variables creadas.
e) Guardar la línea anterior en el fichero hoy.dat sin que la salida se vea por pantalla.
f) Guardar la línea anterior en el fichero hoy.dat y que se vea la salida por pantalla.

:::SHELL-SCRIPTS:::

29. Diseñar un script que pasándole dos argumentos los sume si el primero es menor que el segundo y los reste en caso contrario.
30. Diseñar un script que pida un carácter y nos diga si es un número, una letra u otra cosa.
31. Diseñar un script al cual se le pasa un argumento y si éste es un directorio lista su contenido.
32. Diseñar un script que sume todos los números que se le pasan por parámetro.

33. Diseñar un script que cuente el número de caracteres que tienen los nombres de los ficheros del directorio actual.
34. Diseñar un script que determine si los números que se le pasan como parámetros son pares o impares.
35. Diseñar un script que pida un nombre d'usuari@ y nos diga si éste existe y, si existe, nos diga si está conectado.
36. Diseñar un script denominado “lse” que muestre por pantalla los ficheros y directorios que hay en el directorio actual de la siguiente manera:
Si con el comando ls obtenemos la siguiente salida

```
fit1   prog1   prog2   joc
```

Con nuestro script lse nos lo mostrarà así:

```
|_fit1
.|_prog1
..|_prog2
...|_joc
```

37. Diseñar un script denominado “sumatam” que sume el tamaño de todos los ficheros que se le pasen como argumentos dando un error para todos aquellos argumentos que no existen o que sean directorios.
38. Diseñar un script denominado “sumadir” que sume el tamaño de todos los ficheros del directorio pasado com argumento, dando un mensaje de error si el argumento pasado no es un directorio o no existe.
39. Diseñar un script denominado “opf” que permita copiar (-c), mover (-m) y borrar (-d) ficheros. El script ha de comprobar que la sintaxis utilizada es la correcta.
40. Diseñar un script denominado “usua” que nos vaya pidiendo nombres de usuari@ y si éstos existen y están conectad@s nos dirá la fecha y hora de su conexión. El script se acabará cuando el/la usuari@ teclee FI.
41. Diseñar un script denominado “lro” que muestre todos los ficheros con permiso de lectura de un directorio que se proporciona como parámetro.
42. Diseñar un script denominado “lus” que, dado un identificador de grupo, liste, para cada usuari@ que pertenezca al grupo, su id, su nombre y su directorio de trabajo. La salida sería:

```
USUARI@S DEL GRUPO 52 (5)
NUM. ID.      NOMBRE          DIR. TREBALL
-------------------------------------------------------------
----
235           maria           /home/maria
246           jordi           /home/jordi
...
```

43. Edita un fichero denominado “cumpleaños.txt” las líneas del cual tengan el siguiente formato

```
nombre:fechacumpleaños
```

Realiza un script denominado “generacumple” que te genere el fichero que deberíamos de pasar en el comando crontab para que el día antes de cada fecha de cumpleaños recibas un mail que tenga la siguiente información:

```
...
Subject: Recordatorio Cumpleaños
```

...
Mañana día [data] es el cumpleaños de [nombre]. No olvides felicitárl@.

44. Diseña crontabs para que se realicen las siguientes tareas:
    - Cada día a las 7 de la mañana mira que tamaño tiene el directorio /tmp y si es más grande que 10Mb lo borre y envíe un mail al administrador informándolo.
    - Cada dos días a las 12 de la noche, comprima y empaquete el directorio /home y guarde este archivo en el directorio /usr/src/backups. Si el directorio no existe, se creará.
    - Cada día, de lunes a viernes, cree un informe a las 19:00, a las 20:00 y a las 21:00 con l@s usuari@s que están conectad@s y le envíe al administrador. Este informe tendrá el siguiente aspecto:

```
FECHA: [fecha]
HORA: [hora]

USUARI@S CONECTAD@S ([numusuari@s])
NUM. ID.      NOMBRE  GRUPO
-------------------------------------------------
[id1] [nomusu1]       [grupousu1]
[id2] [nomusu2]       [grupousu2]
[id3] [nomusu3]       [grupousu3]
...
```

:::EXPRESIONES REGULARES:::

45. Crear las expresiones regulares para obtener cadenas que:
  a) contengan la cadena "aba"
  b) contengan tres "b" seguidas
  c) comiencen por dos "a"
  d) acaben por "ba"
  e) comiencen por "a" y acaben por "b" (en el medio puede haber cualquier cosa)
  f) sólo contenga "a" (la cantidad no importa)
  g) primero una "b" y después varias "a"
  h) tenga tanto "a" como "b" (el orden o la cantidad no importa)
  i) no tenga más de tres "a" o tres "b" seguidas
  j) vaya alternando las "a" y las "b" sin repetirse
  k) sólo tenga parejas de "a" y de "b"
  l) tengan una única "a" o una única "b"
  m) haya unas cuantas "a" y después unas cuantas "b" o al revés
  n) contenga la cadena "aba" o la cadena "bab"
  o) contenga la cadena "ba" dos veces

46. Crear las expresiones regulares para obtener cadenas que contengan:
  a) números decimales (con una coma como separador decimal)
  b) números decimales (con una coma o un punto como separador decimal)
  c) números de teléfonos (de nueve cifras, que comiencen por 9 o 6)
  d) códigos postales (de cinco cifras, que comiencen como

mucho por 5)
e) DNI (siete u ocho cifras que pueden ir seguidas de una letra)
f) palabras en minúsculas sin números
g) palabras en las cuales sólo la primera letra sea mayúscula
h) tres o cuatro palabras (sin números)

47. Usando el fichero /etc/passwd dar las expresiones regulares para obtener:
a) usuari@s con la palabra "Unix" al principio del campo de comentario.
b) usuari@s del grupo 101.
c) usuari@s de los grupos 100, 101 o 105.
d) listar usuari@s con UID de un dígito.
f) listar usuari@s con UID de 1 o 2 dígitos.
g) usuari@s con nombre de exactamente 4 caracteres.
h) usuari@ con nombre de 4 caracteres comenzando por r.

48. Hacer expresiones regulares por a:
a) Fecha (DD/MM/AAAA)
b) Hora
c) Correo electrónico
d) Números de teléfono: (93) 841.61.00
e) url's de tipo .org

4.2. links recomendados

:::shell scripts:::

http://www.tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf

Curso introductorio de shell-scripts, muy completo. Completaría opciones y algunos aspectos no explicados en este mini-curso.

http://www.ciberdroide.com/misc/novato/curso/

Curso de GNU/Linux por consola. Tiene una parte de shell-scripting.

:::expresiones regulares:::

http://bulma.net/body.phtml?nIdNoticia=770

Tutorial de expresiones regulares.

http://iie.fing.edu.uy/~vagonbar/unixbas/expreg.htm

Tutorial de expresiones regulares.

http://qmckinney.info/resources/regex.pdf

Referencia rápida de expresiones regulares

:::awk:::

http://www.wikilearning.com/awk_paso_a_paso-wkc-31.htm

Pequeño tutorial de awk

http://www.inicia.es/de/chube/Manual_Awk/Manual_Awk_castellano.pdf

Tutorial completo de awk en castellano

4.3. off-topic

En cap moment no es pot dir *per sempre*. En canvi, sempre hi ha un moment en què cal dir *mai més*. Però: no és el mai més un per sempre? Tanmateix, el per sempre se situa en el temps, mentre el mai més és intemporal.

(Manuel de Pedrolo)