

Programación Avanzada de la Shell

Javier Gómez Santos

Table of Contents

1. Contenidos	1
2. Objetivos	1
3. 1. Introducción al Shell Scripting	1
4. ¿Que es un Shell?	2
5. Distintos Shells	2
6. Historia del bash	2
7. ¿Cuándo usar un shell?	2
8. ¿Cuándo no usar un shell?	3
9. Estructura de directorios en Unix/Linux	3
10. El editor vi	5
11. Pongamos en práctica lo aprendido	6
12. Primeros Pasos	6
13. Estructura de un Shell y nociones básicas	6
14. Nuestro primer Shell	9
15. Nuestro primer Shell	10
16. Referencias a variables en Unix	11
17. Condiciones	14
18. Estructuras de control	19
19. Condiciones	22
20. Arrays	24
21. Arrays	24
22. Prácticas	25
23. Prácticas	25
24. Funciones	31
25. Declaración de funciones	31
26. Paso de parámetros	32
27. Re-declaración de funciones (método readonly)	32
28. Hacer una librería de funciones	34
29. Prácticas	37
30. Gestión de Entrada y Salida	40
31. Entrada y Salida por teclado	40
32. Entrada y Salida por fichero	40
33. Entrada estándar, salida estándar y salida de errores	41
34. Redirigir la salida estándar o la salida de errores	41
35. Prácticas	42
36. Algunos comandos útiles	43
37. Los básicos	44
38. El uso del Pipe	45

39. grep	45
40. find	45
41. sed	46
42. sort	47
43. head y tail	47
44. uniq	47
45. rev	48
46. cut	48
47. nohup	50
48. expr	50
49. Prácticas	53
50. Señales y Traps	56
51. ¿Qué es una señal en Unix?	56
52. Tratamiento de señales en Unix	57
53. Practicas	61
54. Comando getopts	62
55. ¿Para qué sirve el comando getopts?	63
56. Uso de getopts	64
57. Prácticas	67
58. Expresiones regulares	68
59. Expresiones Regulares	68
60. Patrones de expresiones regulares	69
61. Ejemplos de expresiones regulares	70
62. Prácticas con egrep y sed	73
63. awk	80
64. ¿Qué es el AWK y qué es awk?	80
65. Estructura de un programa en AWK	81
66. Operadores en AWK	82
67. Bucles en AWK	84
68. El comando print	87
69. Un poco de práctica	87
70. Variables pre-construidas en AWK	88
71. Un poco de práctica 2	89
72. Arrays asociativos	90
73. Prácticas	90
74. Monitorización y optimización	93
75. Monitorización y ajuste del sistema I/O	93
76. Monitorización y ajuste CPU	109
77. Monitorización y ajuste Memoria Virtual	123
78. Monitorización y ajuste de red	125
79. Gestión e instalación de paquetes	129

80. YUM	129
81. RPM.....	131
82. Instalación mediante compilación	143
83. Planificación	143
84. Introducción	143
85. nohup}	144
86. at	145
87. crontab.....	147
88. dialog	149
89. Gestión de usuarios.....	157
90. Comando Unix para la gestión de usuarios	157
91. Módulos PAM	159
92. LDAP y NIS	171
93. Práctica Final	174
94. Práctica final 1	174
95. Práctica final 2	175
96. Fin	175



1. Contenidos

1. Introducción al Shell Scripting
2. Primeros Pasos
3. Arrays
4. Funciones
5. Gestión de Entrada y Salida
6. Algunos comandos útiles
7. Señales y Traps
8. Comando getopts
9. Expresiones regulares
10. awk
11. Monitorización y optimización
12. Gestión e instalación de paquetes
13. Planificación
14. Gestión de usuarios
15. Práctica Final

2. Objetivos

- El objetivo del curso es el de dotar al alumno de las **competencias necesarias** para la gestión de tareas a alto nivel a través de procesos shell.
- Descargar al administrador de tareas repetitivas o manuales mediante la creación de procesos batch.
- Manipulación de ficheros de datos, entradas y salidas.
- Conocer el conjunto de herramientas que proporciona el SO para su uso.
- Agrupar distintos comandos para obtener un resultado conjunto.

3. 1. Introducción al Shell Scripting

4. ¿Que es un Shell?

- La finalidad de un Shell es añadir una capa de abstracción para aislarnos del sistema operativo.
- Mediante comandos, podemos operar con nuestro SO.
- Este código es interpretado, por lo que el propio script es "legible".
- Estos scripts (o procesos por lotes) nos permiten interactuar con otra serie de programas del SO denominados **comandos**.

5. Distintos Shells

- Existen diversos shells o interpretes en los SO
 - Bourne Shell (sh)
 - Alquist Shell (ash)
 - Bourne-Again shell (bash)
 - Korn shell (ksh)
 - C shell (csh)
 - ...

6. Historia del bash

- Hacia 1978 el shell Bourne era el shell distribuido con el Unix versión 7. Stephen Bourne, por entonces investigador de los Laboratorios Bell, escribió el shell Bourne original.
- Brian Fox escribió el shell bash en 1987.
- En 1990, Chet Ramey se convirtió en su principal desarrollador.

7. ¿Cuándo usar un shell?

- Como ya hemos dicho antes, un shell es un conjunto de instrucciones **que deben ser interpretadas** por el SO.
- Los shells han tenido principalmente 2 objetivos:
- Servir de estructura de control de procesos finales
 - Preparar la ejecución de un proceso
 - Controlar las salidas y responder ante problemas
 - Gestionar un workflow de varios procesos dependientes
- Realizar programas sencillos
 - Son rápidos de crear, aunque al ser código interpretado pueden consumir más cpu
 - Preprocesar ficheros

- Mover ficheros

8. ¿Cuándo no usar un shell?

- Al ser código interpretado **no** debe usarse para realizar procesos muy pesados.
- Cuando se quiera acceder a direcciones físicas de memoria (drivers).
- Cuando no queramos entregar nuestro código.

9. Estructura de directorios en Unix/Linux

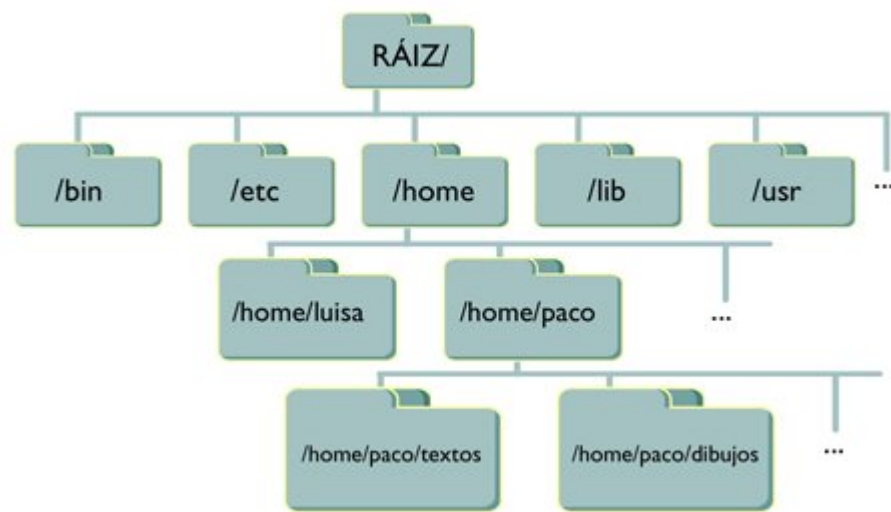


Figure 1. Estructura de directorios.

- En Unix, la raíz del árbol de directorios es la carpeta /.
- Cualquier otro directorio, representa una rama del árbol que empieza en la raíz /.
- Otros discos duros, CD-Roms, Discos USB... se representan como una rama del mismo árbol.
- Los subdirectorios más importantes son:
 - /boot: Contiene información indispensable para el arranque del sistema y el núcleo del Unix.
 - /bin: Contiene ciertos programas que estarán disponibles para su ejecución (por ejemplo comandos como cat, ls, ps ...).
 - /sbin: Programas disponibles sólo para el administrador.
 - /usr: Programas disponibles para usuarios finales (Y sus datos con acceso sólo lectura)

- Los subdirectorios más importantes son: (continuación)
 - /lib: Librerías y módulos.
 - /etc: Archivos de configuración y servicios.
 - /dev: Dispositivos conectados (o no) al sistema.
 - /home: Mantiene información de los usuarios del sistema.
 - /root: Mantiene información del administrador del sistema.
 - /tmp: Archivos temporales creados por algunos programas (serán borrados por el sistema operativo durante el arranque).
 - /var: En este directorio los programas que lo requieran pueden mantener archivos que deban modificarse frecuentemente.
 - /proc: Este directorio es virtual, no está presente en el disco, porque es creado por el sistema para intercambiar información con más facilidad.

Los comandos necesarios para navegar e interactuar sobre el sistema de ficheros son:

- cd -Change Directory-
- pwd -Print Working Directory-
- mkdir -MaKe Directory-
- rm -ReMove a file-
- rmdir -ReMove a Directory-
- ls -LiSt files-
- cp -CoPy a file-
- mv -MoVe a file or directory-

10. El editor vi

Vim Reference Card

a insert text after the cursor position	A insert text at the end of the line (same as \$a)	CTRL-A increment the number under the cursor
b go back a word	B go back a WORD	CTRL-B Move one screen backward
c {motion} change text till motion	C change till end of the line	cc change the whole line
d {motion} delete text till motion	D delete till end of the line	dd delete the whole line
e move to end of the word	E move to end of the WORD	:e filename open filename
f find a char in the same line	F find a char in the same line, backwards	CTRL-F Move one screen forward
gg go to first line	G go to last line	qu {motion} change to lowercase qU {motion} uppercase
qD jump to declaration in the current function	qD jump to global declaration	qf open the file under cursor
h move cursor to the left	H go to the top of the screen	:h online help
i insert text in the cursor position	I insert text at the start of the line (same as ^i)	CTRL-I go to next cursor position
j move down the cursor a line	J join two lines	:%j join all lines into single line
k move up the cursor a line	K jump to hyperlink	CTRL-X CTRL-K complete from dictionary
l move cursor to the right	L go to bottom of the screen	CTRL-X CTRL-L complete whole lines
ma create a mark a	M go to middle of the screen	:m {address} move the lines to specific {address}
n find the next match (used in search)	N find the previous match (used in search)	CTRL-N complete the next word
o create a new line below the cursor	O create a new line above the cursor	CTRL-O go to the previous cursor position
p put the yanked text/copied text	P put the yanked text before cursor	CTRL-P complete the previous word
q create a macro named a, press q again to stop recording	Q {motion} format the text till motion	QQ format the current line
r replace a char	R replace several chars; ends with ESC	:r filename read the file content to this file
s delete the char under cursor and jump to insert mode	S delete the whole line and jump to insert mode (cc)	:%s/foo/bar/g replace all foo with bar
t go till the char (similar to f)	T go till the char, backwards	:t same as :m
u undo	U undo all the changes in the line	CTRL-R redo
v go to visual mode (char-wise)	V go to visual mode (line-wise)	:v/foo/d execute d(delete) on lines that not contain foo
w go to the next word	W go to the next WORD	:w write to file
x delete a char under the cursor	X delete a char before the cursor	CTRL-X CTRL-F complete the file name
y {motion} yank text(copy) till motion	Y yank the whole line	yy yank the whole line (same as Y)
z {motion} create a fold till motion	zf {motion} create a fold till motion	ZZ (uppercase) save and exit

Special Characters

~ change case	xp exchanges two chars	[f go to previous function	[m go to start of block
^^ jump to previous location	dwp exchange two words]] go to next function]m go to end of block
! {motion} redirect the output	!! redirects output current line	{ goto previous paragraph	} go to next paragraph
@a play the macro a	@@ play the previous macro	[i find the previous occurrence of word under cursor]i find the next occurrence of word
# search backwards		: start an ex command	
\$ jump to end of line	`` jumps to last cursor position	'a go to mark a at the start of line	`a go to mark a
% jump to the matching (, [, {	:q! really quit, no questions	; find next f or t search (see f or t)	, find previous f or t
^ jump to start of line	:wq write and quit	. repeat the previous change	
& perform the previous substitute	:ls lists buffers	< {motion} shift left till motion	> {motion} shift right till motion
* search the char under cursor	:e# edit alternate file	/ start a search	? start a search backwards
(jump to start of sentence		 goto first non blank char in line	0 (zero) goto first char in the line
) jump to end of sentence		CTRL-P complete previous word in the file(insert mode)	CTRL-N complete the next word in the file
- go to previous line		CTRL-A increase the number under cursor	CTRL-X decrease the number under cursor
+ go to next line		CTRL-O jump to previous location (in history)	CTRL-I jump to next location (in history)
= {motion} indent till motion	== indent the current line	CTRL-X CTRL-L complete the next line(insert mode)	

Figure 2. Diferencia entre el la pila y el montón.

El editor vi es un editor de texto incluido en cualquier sistema Unix/Linux.

11. Pongamos en práctica lo aprendido

Desde el home del usuario hay que crear el siguiente árbol de directorios

- Practica_00
 - Parte_A
 - Parte_B
 - Subparte_1
 - Subparte_2
 - Parte_C

Dentro de cada directorio, crear usando vi un fichero con un texto.

Borrar el directorio Parte_B con un sólo comandos.

mostrar todos los directorios y subdirectorios con un sólo comandos.

- El comando "man" en Unix nos puede ayudar a conocer mejor un comando del sistema.

12. Primeros Pasos

1. Estructura de un Shell y nociones básicas
2. Nuestro primer Shell
3. Permisos en Unix
4. Referencias a variables en Unix
5. Condiciones
6. Estructuras de control
7. Prácticas

13. Estructura de un Shell y nociones básicas

Un Shell es un conjunto de instrucciones preparadas para ser procesadas/interpretadas. Si escribiéramos directamente sobre la línea de comandos las instrucciones surtirían el mismo efecto.

```
#!/bin/bash
#Comentarios

#Variables (se pueden definir en cualquier punto)
var1=3;
var2="hola";

#Funciones
function f1 {
    #codigo
}

#codigo ejecutable
f1
exit
```

Las variables se pueden definir en cualquier momento del código, y a partir de entonces serán accesibles para cualquier función que las use

```
#!/bin/bash
#Comentarios

#Funciones
function f1 {
    #codigo
}

f1
#Variables
var1=3;
var2="hola";

#codigo ejecutable
exit
```

```
#!/bin/bash
#Comentarios

#Funciones
function f1 {
    #codigo
}

#Variables
var1=3;
var2="hola";

#codigo ejecutable
f1
exit
```

Para acceder al contenido de una variable debemos anteceder el carácter **\$** al nombre de la misma. El comando **echo** nos permite mostrar información en pantalla.

```
#!/bin/bash

var2="hola"
echo var2
exit
```

Mostrará en pantalla "var2"

```
#!/bin/bash

var2="hola"
echo $var2
exit
```

Mostrará en pantalla "hola"

14. Nuestro primer Shell

Vamos a crear un shell sencillo, que muestre un "hola mundo". Situémonos en el directorio `Practica_00` y creemos un shell con el `vi`.

```
#!/bin/bash

echo "Hola Mundo"

exit
```

Para ejecutarlo invocaremos al shell mediante **./nombre_del_shell.sh**. Existen otras maneras de invocar los shell-scripts pero hablaremos de ellas más adelante.

- `./script.sh`
- `. script.sh`
- `bash script.sh`

Pero podemos encontrarnos el siguiente problema al ejecutarlo:

```
[usuario@localhost curso_shell]$ ./mi_script.sh
-bash: ./mi_script.sh: Permission denied
[usuario@localhost curso_shell]$
```

Figure 3. Esto ocurre porque el script no tiene permisos.

15. Nuestro primer Shell

Los permisos en Unix se gestionan en 3 niveles

- Usuario
- Grupo
- Otros

Y para cada nivel se definen permisos de

- Lectura
- Escritura
- Ejecución

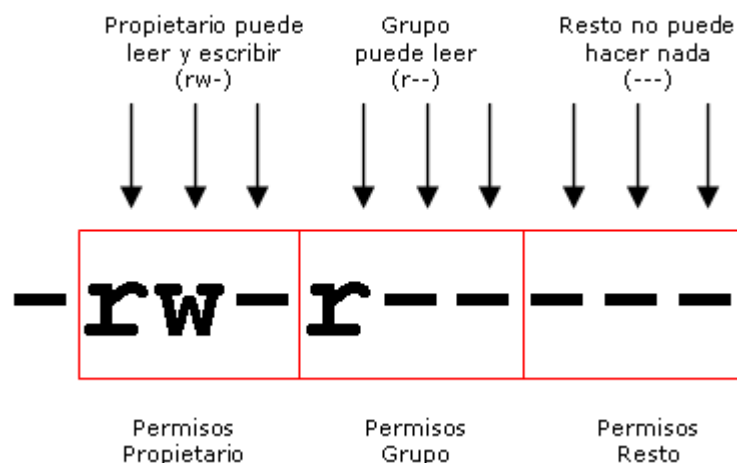


Figure 4. Permisos de un fichero.

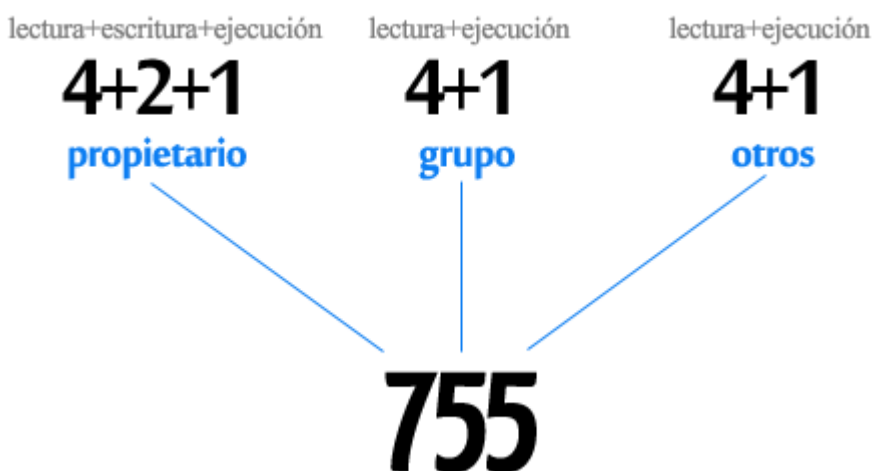


Figure 5. Representación interna.

Podemos cambiar los permisos de un fichero mediante el comando **chmod**

```
chmod 751 fichero.sh
```

16. Referencias a variables en Unix

Podemos acceder a las variables de entorno como si fueran variables propias del sistema. Si quisiéramos por ejemplo acceder a la variable de entorno **UID** que contiene el ID de nuestro usuario, podríamos consultarla mediante la siguiente instrucción:

```
echo $UID
```

Así mismo, los parámetros que pasamos en la llamada de un script se almacenan en unas variables especiales

- \$0 - Contiene el nombre del script ejecutándose
- \$1 - Contiene el primer parámetro

- \$2 - Contiene el segundo parámetro ...

Podemos usar estos parámetros como una variable mas

```
echo "Ha escrito como parametro 1 <$1>"
```


Las variables que creamos en nuestros scripts son tratadas como variables locales, por lo que desaparecen al finalizar el script y no son utilizables por otros procesos.

Si quisiéramos que nuestra variable persistiera, deberíamos usar el comando export

```
export mi_variable_persistente="Mi_Dato"
```

Existe un conjunto de variables de sistema especiales que son de gran utilidad para el administrador o programador. Algunas de ellas son:

- \$\$ → Devuelve el PID de la Shell
- \$? → Devuelve el último código de retorno recibido
- \$# → Devuelve el número de parámetros recibidos por script
- \$* → Devuelve todos los parámetros recibidos
- \$1,\$2... → Devuelve el parámetro indicado
- \$! → Devuelve el PID del último proceso lanzado en background

[Más información](#)

17. Condiciones

Las condiciones que se pueden usar dentro de los bucles de control pueden dividirse en tres tipos

- Chequeos con cadenas
- Chequeos con enteros
- Chequeos con ficheros

Este último punto resulta realmente útil, ya que muchos de nuestros procesos van a desarrollar un comportamiento distinto a si el fichero no está, a si existe pero está vacío o si este contiene datos.

Chequeos con cadenas:

- string → devuelve **true** si la cadena es no nula
- -z string → devuelve **true** si la longitud del string es 0
- -n string → devuelve **true** si la longitud del string **no** es 0
- string1 = string2 → devuelve **true** si los strings son iguales
- string1 != string2 → devuelve **true** si los strings son distintos

```
[ -z $MiVariable ]
```

Chequeos con enteros:

- `int1 -eq int2` → los enteros son iguales
- `int1 -ne int2` → los enteros son distintos
- `int1 -gt int2` → `int1` mayor que `int2`
- `int1 -ge int2` → `int1` mayor o igual que `int2`
- `int1 -lt int2` → `int1` menor que `int2`
- `int1 -le int2` → `int1` menor o igual que `int2`

```
[ $Num1 -gt 0 ]
```

Chequeos con ficheros:

- -e file → existe
- -r file → existe y es legible
- -w file → existe y se puede escribir
- -x file → existe y es ejecutable
- -f file → existe y es de tipo regular
- -d file → existe y es un directorio
- -c file → existe y es un dispositivo de caracteres
- -b file → existe y es un dispositivo de bloques
- -p file → existe y es un pipe
- -S file → existe y es un socket
- -L file → existe y es un enlace simbólico
- -u file → existe y es setuid
- -g file → existe y es setgid
- -k file → existe y tiene activo el sticky bit
- -s file → existe y tiene tamaño mayor que 0

```
[ -r ./mi_fichero.txt ]
```

Por último, comentar que existen otros operadores lógicos propios del algebra de boole

- `!` → Invierte el resultado
- `-a` → operador AND
- `-o` → operador OR
- `(expr)` → Agrupa operaciones, pero debemos cerrar estas paréntesis (puesto que tienen significado especial en la shell)
- `=~` → Analiza expresiones regulares (Lo veremos más adelante)

```
[ ! \ ( $var1 -ge 0 -o -w "$RUTA/mifichero.txt" \ ) ]
```

También podemos evaluar expresiones aritméticas mediante doble corchete, como en el ejemplo

```
if [[ $Var < 5 ]]
```

También aplicable a las expresiones regulares

```
if [[ $Texto =~ ^a*b ]]
```

18. Estructuras de control

Bucle if/else

```
if "condicion"
then
    "comandos"
elif "condicion"
then
    "comandos"
else
    "comandos"
fi
```

```
if [ $1 -gt 0 ]
then
    echo "Mayor que 0"
elif [ $1 -lt 0 ]
then
    echo "Menor que 0"
else
    echo "Es 0"
fi
```

Bucle for

```
for nombre [lista de elementos]
do
    acciones sobre $nombre
done
```

```
#!/bin/bash

for i in `cat lista.txt`
do
    echo $i
done
```


Bucle while

```
while condicion
do
    comandos
done
```

```
#!/bin/bash

NUM=0

while [ $NUM -le 10 ]; do
    echo "\$NUM: $NUM"
    let NUM=NUM+1
done
```

Bucle case

```
case expresion in
    caso_1 )
        comandos;;
    caso_2 )
        comandos;;
    .....
esac
```

```
case $1 in
    0)
        echo "Parametro 1 es igual a cero";;
    1)
        echo "Parametro 1 es igual a uno";;
    2)
        echo "Parametro 1 es igual a dos";;
    3)
        echo "Parametro 1 es igual a tres";;
    *)
        echo "Es otro numero";;
esac
```

19. Condiciones

Practica 1:

Crearemos un script que cumpla la siguiente funcionalidad

- Si el número de parámetros pasados como argumentos es distinto de 1, salga del script devolviendo un 1 y mostrando un error en pantalla
- Si el tamaño del parámetro es igual o inferior a 2 caracteres (entendemos que es numérico), mostramos en pantalla el mensaje "OK" tantas veces como indique el parámetro
- Si no cumple estas condiciones, miramos si existe el fichero con el nombre dado por parámetro
 - Si no existe, lo creamos. Mostramos por pantalla "Fichero <\$NombreFichero> creado"
 - Si existe y está vacío mostramos "Fichero <\$NombreFichero> vacío"
 - Si existe y no está vacío, mostramos "Fichero <\$NombreFichero> contiene datos"

Consejos

- \$# devuelve el número de parámetros pasados al Shell
- \$1 es el contenido del primer parámetro
- \${#VAR} devuelve el tamaño de la variable

- touch es un comando que permite alterar la fecha de creacion de un fichero, o crear uno vacío

```
#!/bin/bash

if [ $# -ne 1 ]
then
    echo "Se necesita un parámetro"
    exit 1
fi

Param1=$1

if [ ${#Param1} -le 2 ]
then
    Bucle=0
    while [ $Bucle -lt $Param1 ]
    do
        echo "OK - $Bucle"
        let Bucle=Bucle+1
    done
else
    if [ ! -e $Param1 ]
    then
        touch $Param1
        echo "Fichero <$Param1> creado"
    elif [ -s $Param1 ]
    then
        echo "Fichero <$Param1> contiene datos"
    else
        echo "Fichero <$Param1> vacío"
    fi
fi
```

20. Arrays

1. Arrays
2. Prácticas

21. Arrays

Los Arrays son un añadido *reciente* al interprete Shell, y su uso y gestión es análoga al resto de variables. Aunque un Array puede inicializarse de manera explícita mediante un *declare -a*, el igual que el resto de variables no es necesaria su declaración para su uso

Los arrays han de ser unidimensionales.

```
mi_array[0]=2
mi_array[1]=4
mi_array[2]=8

echo ${mi_array[1]}
```

Podemos mostrar todos los elementos del array mediante

```
${temp[*]}
```

Podemos conocer el total de elementos del array mediante

```
${#temp[*]}
```

22. Prácticas

La colección de elementos de un array no tienen por que ser del mismo tipo. Instrucciones como la mostrada a continuación son válidas

```
mi_array[0]=2
mi_array[1]="hola"
mi_array[2]=8

echo ${mi_array[1]}
```

23. Prácticas

Practica 1

Generar un script que acepte un número por parámetro y genere un array de tantos elementos como indique el parámetro. Después, listar los elementos de dicho Array, uno por línea.

- Pistas:
 - Usar \$# para conocer el número de parámetros
 - Usar \${#array[*]} para conocer el número de elementos del array
 - Usar \${array[*]} si se quiere obtener todos los elementos
 - El comando let permite operar con variables (sumas, restas...)
 - El comando seq permite obtener la colección de números existentes entre 2 números dados

Opción A: hacer un bucle while y mostrar los elementos

```
#!/bin/bash

if [ ! $# -gt 0 ]; then
    echo "Se debe indicar el número de elementos del array"
    exit -1
fi

NumElementos=$1
ElementosCreados=0

while [ $ElementosCreados -lt $NumElementos ]
do
    mi_array[$ElementosCreados]=$ElementosCreados
    let ElementosCreados=ElementosCreados+1
done

while [ $ElementosCreados -gt 0 ]
do
    let ElementosCreados=ElementosCreados-1
    echo " -> ${mi_array[$ElementosCreados]}"
done

exit 0
```

Opcion B: generar un bucle for por cada elemento del bucle

```
#!/bin/bash

if [ ! $# -gt 0 ]; then
    echo "Se debe indicar el número de elementos del array"
    exit -1
fi

NumElementos=$1
ElementosCreados=0

while [ $ElementosCreados -lt $NumElementos ]
do
    mi_array[$ElementosCreados]=$ElementosCreados
    let ElementosCreados=ElementosCreados+1
done

for i in ${mi_array[*]}
do
    echo "-> $i"
done

exit 0
```

Practica 2

Diseñar un programa que pida por pantalla a un usuario posición y valor de una variable para almacenar en un array. El script debe decir si esa posición ha sido reescrita o no existía valor para ella. Cuando el usuario escriba "FIN" como posición de la variable, se para la lectura de datos por parte del usuario y se muestren todas las posiciones ocupadas con sus respectivos valores.

Consejos

- `read` → Este comando permite leer una entrada del usuario
- `${!array[*]}` → Devuelve una colección de claves del array
- `-z` nos permitía saber si una variable está definida o no

Una posible solución

```
#!/bin/bash

FinBucle=0

while [ $FinBucle -eq 0 ]
do
    echo "Por favor, introduzca posicion (FIN para finalizar lectura)"
    read Leido
    if [ $Leido = "FIN" ]
    then
        echo "FIN leído"
        FinBucle=1
    else
        echo "Por favor, introduzca el valor"
        read Valor
        if [ ! -z ${mi_array[$Leido]} ]
        then
            echo "La posicion <$Leido> se sobrescribe, su valor anterior
era <${mi_array[$Leido]}>"
            fi
            mi_array[$Leido]=$Valor
        fi
    fi
done

for Claves in ${!mi_array[*]}
do
    echo "En la posicion <$Claves> se almacena el valor <${mi_array[$Claves]}>"
done
exit 0
```

Practica 3

Generaremos un script que pida por pantalla al usuario nombres de ficheros y los vaya almacenando en un array en **posiciones consecutivas**.

La lectura finalizará cuando el usuario escriba **FIN**, y se recorrerá el array para indicar elemento a elemento qué ficheros existen y qué ficheros NO existen.

```
#!/bin/bash

FinBucle=0
PosActual=0
while [ $FinBucle -eq 0 ]
do
    echo "Introduzca un nombre de fichero (introduzca valor FIN para finalizar lectura)"
    read Leido
    if [ $Leido = "FIN" ]
    then
        echo "FIN leído"
        FinBucle=1
    else
        mi_array[$PosActual]=$Leido
        let PosActual=PosActual+1
    fi
done

for Ficheros in ${mi_array[*]}
do
    if [ ! -e $Ficheros ]
    then
        echo "El fichero <$Ficheros> NO existe"
    else
        echo "El fichero <$Ficheros> EXISTE"
    fi
done
exit 0
```

24. Funciones

1. Declaración de funciones
2. Paso de parámetros
3. Re-declaración de funciones (método readonly)
4. Hacer una librería de funciones
5. Prácticas

25. Declaración de funciones

Como ya vimos anteriormente, una función se debe definir al principio de un shell (para que esté visible para el resto de elementos) Internamente, el shell carga las funciones en memoria (a medida que las lee) y estarán disponibles durante la ejecución del script (es decir, tienen scope local)

```
function mi_funcion()
{
    echo "hago algo"
}
```

No es necesario usar la palabra reservada **function** para declarar la función

```
mi_funcion()
{
    echo "hago algo"
}
```

26. Paso de parámetros

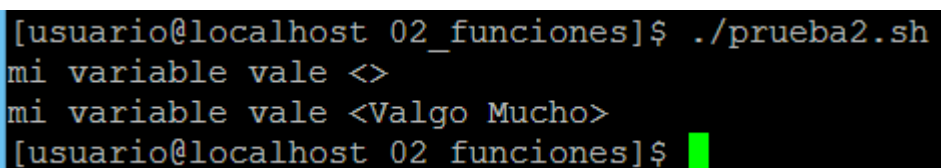
En la declaración de la función nunca se especifica cuántos parámetros se va a recibir. Esto no quiere decir que no podamos recibir parámetros, los recibimos tal como los recibimos al invocar un shell, mediante las variables especiales \$1, \$2...

```
funcion "Param1" "Param2"
```

Además, cualquier variable que haya sido definida **previo a la invocación de la función** estará disponible en la misma. El siguiente código ilustra este comportamiento

```
funcion()
{
    echo "mi variable vale <$MiVar>"
}

funcion
MiVar="Valgo Mucho"
funcion
```



```
[usuario@localhost 02_funciones]$ ./prueba2.sh
mi variable vale <>
mi variable vale <Valgo Mucho>
[usuario@localhost 02_funciones]$
```

Figure 6. Resultado de la ejecución del script.

27. Re-declaración de funciones (método readonly)

Una función se puede redeclarar todas las veces que se quiera, aunque esto no es una práctica muy

habitual.

Tan sólo habría que volver a definir la función

```
function ()  
{  
    echo "Funcion 1"  
}  
  
function  
  
function ()  
{  
    echo "Funcion redefinida"  
}  
  
function
```

```
[usuario@localhost 02_funciones]$ ./prueba3.sh  
Funcion 1  
Funcion redefinida
```

Figure 7. Resultado de la ejecución del script.

Podemos evitar la redefinición de funciones mediante la instrucción **readonly -f nombre_funcion**

```
function ()
{
    echo "Funcion 1"
}

function

readonly -f function

function ()
{
    echo "Funcion redefinida"
}

function
```

```
[usuario@localhost 02_funciones]$ ./prueba4.sh
Funcion 1
./prueba4.sh: line 15: function: readonly function
Funcion 1
```

Figure 8. Resultado de la ejecución del script.

28. Hacer una librería de funciones

- Como hemos comentado anteriormente, una función tiene una vida ligada a la ejecución de un script, por lo que parece difícil tener una librería de funciones disponibles para ser usadas por otros scripts
- Algunos shells implementan comandos como **autoload** en KSH o **export -f** que permiten conservar las funciones ya declaradas, pero debido a su falta de estandarización no vamos a profundizar en ellas
- Sin embargo, si podemos cargar una librería de funciones en nuestros scripts, pero para eso tenemos que entender lo que vimos en el punto **Nuestro primer Shell**

- Cuando ejecutamos un script, al igual que cuando llamamos a una función en cualquier otro lenguaje, creamos una nueva pila con nuestras variables locales, inaccesible desde el padre.
- Así cuando un script invoca a otro script que usa una variable del mismo nombre, esta es de ámbito local y no afecta a la variable del script que realiza la invocación

```
Variable1="hijo"
```

```
echo $Variable1
```

```
Variable1="padre"
```

```
echo $Variable1
```

```
./script_hijo.sh
```

```
echo $Variable1
```

```
[usuario@localhost 02_funciones]$ ./script_padre.sh
padre
hijo
padre
```

Figure 9. Resultado de la ejecución del script Padre.

- Sin embargo, existe otra manera de invocar los scripts, pidiendo que no se ejecuten en una pila nueva, si no que funcionen como si estuvieran definidos como parte del propio script.
- Para hacer esto, en lugar de invocar al script mediante **./script** debemos invocarlo **. script**

```
Variable1="padre"

echo $Variable1

. script_hijo.sh

echo $Variable1
```

```
[usuario@localhost 02_funciones]$ ./script_padre2.sh
padre
hijo
hijo
```

Figure 10. Resultado de la ejecución del script Padre.

De esta manera, podemos hacer un script en el que sólo existan declaración de funciones, y que sea cargado por todos los scripts que vayan a hacer uso de dichas funciones

```
#script: librerias_funciones.sh
f1()
{
    echo "cosas 1"
}
...
fn()
{
    echo "cosas n"
}
```

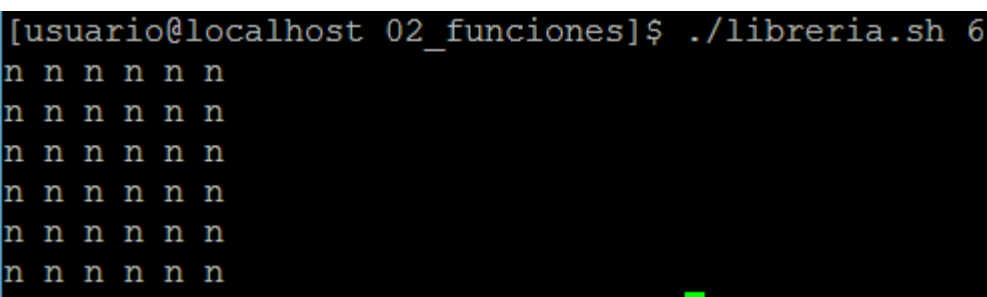
```
#script: mi_script.sh
. librerias_funciones.sh

f1
f2
...
exit 0
```

29. Prácticas

Vamos a crear nuestra propia librería de funciones, que posteriormente iremos enriqueciendo y usando en futuros proyectos. De momento necesitamos 2 funciones

- recuadro: obtiene un número como parámetro y muestra por pantalla un recuadro de NxN elementos



```
[usuario@localhost 02_funciones]$ ./libreria.sh 6
n n n n n n
n n n n n n
n n n n n n
n n n n n n
n n n n n n
n n n n n n
```

Figure 11. Ejemplo de recuadro.

- hora_minuto_segundo: Una función que guarda en una variable llamada **Hora** la hora, minuto y segundo del sistema
 - El comando para obtener fecha del sistema es **date**
 - No es lo mismo en Unix la doble comilla "algo", la comilla simple 'algo' y el apóstrofe `algo`
 - "" es texto que hace sustitución de variables. "hola \$Amigo" sustituirá la variable **\$Amigo** por su valor

- " es texto fijo, no hace sustitución de variables. 'hola \$Amigo' mostrará el texto tal cual
 - `` ejecuta el comando que aparezca y sustituirá variables. `hola \$Amigo` llamará al comando **hola** con el valor de **\$Amigo** como parámetro
-
- Por último, debemos crear un script que cargue esta librería y use sus funciones para comprobar que todo está correcto.

Una posible implementación de la librería:

```
#!/bin/bash

recuadro ()
{
if [ $# -ne 1 ]
then
    return 1
fi
#creamos linea base
Tam_matriz=$1
Iterador=0
while [ $Iterador -lt $Tam_matriz ]
do
    Linea="$Linea n"
    let Iterador=Iterador+1
done
Iterador=0
while [ $Iterador -lt $Tam_matriz ]
do
    echo $Linea
    let Iterador=Iterador+1
done
}

hora_minuto_segundo()
{
Hora=`date +%H%M%S`
}
```

El script para probar dicha librería podría ser como se muestra:

```
#!/bin/bash

. libreria.sh

echo "Pinto un recuadro de 4"
recuadro 4

echo "Pido la hora"
hora_minuto_segundo

echo "Me ha dado la hora: <$Hora>"
```

30. Gestión de Entrada y Salida

1. Entrada y Salida por teclado
2. Entrada y Salida por fichero
3. Entrada estándar, salida estándar y salida de errores
4. Redirigir la salida estándar o la salida de errores
5. Prácticas

31. Entrada y Salida por teclado

Aunque ya lo hemos visto durante el curso, existen dos maneras básicas de comunicación de la shell con el usuario

- **read** → Lee un valor escrito por el usuario directamente sobre la terminal
- **echo** → Muestra por pantalla el mensaje que se envíe como parámetro del **echo**

```
echo "Por favor, escriba algo por pantalla"
read Leido
echo "Ha introducido <$Leido>"
```

32. Entrada y Salida por fichero

Sin embargo, una de las grandes ventajas del Shell Scripting, es poder trabajar de manera casi transparente con los ficheros. Podemos trabajar directamente con los ficheros mediante los operadores de indirección

- **>>** → este operador redirecciona la salida al fichero destino especificado, concatenando los datos si ya existiera

- > → análogo al anterior, pero sin hacer append. Machaca lo que hubiera
- < → Redirecciona la entrada con la información del fichero origen

Ejemplos:

```
#guardamos Hola en un fichero destino, machacando lo que hubiera
echo "Hola" > destino.txt

#Concatenamos al fichero destino.txt
echo "Adios" >> destino.txt

#usamos el comando wc con los datos almacenados en el fichero destino.txt
wc -l < destino.txt
```

33. Entrada estándar, salida estándar y salida de errores

En Unix existe tres descriptores de fichero definidos por el SO que están siempre disponibles

- 1 → Corresponde a la salida estándar
- 2 → Corresponde a la salida de error
- 3 → Corresponde a la entrada estándar

Siempre que, por ejemplo, hacemos uso del comando **echo**, estamos pasando esa información al fichero 1. Si un script da un error, vuelca estos mensajes en el fichero 2.

34. Redirigir la salida estándar o la salida de errores

Cuando invocamos a un script podemos redirigir estas salidas con facilidad, como se muestra en el ejemplo a continuación

```
./mi_script.sh Param1 Param2 1>fichero_salida.txt 2>fichero_error.txt
```

Incluso podemos redirigir ambas salidas a un mismo fichero, como se muestra

```
./mi_script.sh Param1 Param2 1>fichero_salida.txt 2>&1
```

Esta redirección de las salidas y entradas puede ser redefinida también en el propio Shell, usando un pequeño truco, el comando **exec**.

De esta manera, el script se comporta como si se hubiera hecho la redirección al invocarse, pero se usará la salida estándar de cualquier elemento previo a la invocación de este comando.

```
exec 1>salida_redirigida.txt

echo "Prueba primera"

cat fichero_entrada.txt
```

Este script no generará salida estándar por pantalla, en su lugar guardará todo en el fichero **salida_redirigida.txt**.

35. Prácticas

Vamos a hacer un programa que vaya pidiendo palabras a un usuario y rellene un array con ellas, hasta que el usuario escriba **FIN**.

Una vez finalizado este punto, escribiremos las variables en tres ficheros distintos:

- `peques.txt` → Para las palabras que ocupen 3 letras o menos
- `mediantas.txt` → Para las palabras que ocupen entre 4 y 7 letras
- `grandes.txt` → Para las palabras que ocupen 8 letras o más

Una posible solución:

```
#!/bin/bash

FinBucle=0
Posicion=0

#limpiamos los ficheros
rm -fr peques.txt
rm -fr medianas.txt
rm -fr grandes.txt

while [ $FinBucle -eq 0 ]
do
    echo "Por favor, introduzca un valor (FIN para finalizar lectura)"
    read Leido
    if [ $Leido = "FIN" ]
    then
        echo "FIN leído"
        FinBucle=1
    else
        mi_array[$Posicion]=$Leido
        let Posicion=Posicion+1
    fi
done

for Valores in ${mi_array[*]}
do
    Tam_Valor=${#Valores}

    if [ $Tam_Valor -le 3 ]
    then
        echo $Valores >> peques.txt
    elif [ $Tam_Valor -le 7 ]
    then
        echo $Valores >> medianas.txt
    else
        echo $Valores >> grandes.txt
    fi
done
exit 0
```

36. Algunos comandos útiles

1. Los básicos
2. El uso del Pipe
3. grep

4. find
5. sed
6. sort
7. head y tail
8. uniq
9. rev
10. cut
11. nohup
12. expr
13. Prácticas

37. Los básicos

- **echo** → Muestra por la salida estándar el mensaje pasado como parámetro
- **ls** → Listado de ficheros y directorios
- **history** → Muestra el histórico de comandos ejecutados
- **mkdir** → Crea un directorio
- **touch** → Cambia la fecha del fichero por la fecha actual del sistema
- **chmod** → Cambia los permisos de acceso de un fichero/directorio
- **chown** → Cambia el usuario y grupo dueños de los ficheros
- **date** → Pide la fecha del sistema
- **cp** → Copia un fichero
- **cat** → Muestra el contenido de los ficheros pasados por parámetro
- **mv** → Mueve un fichero
- **wc** → Cuenta el total de palabras o líneas que tenga un fichero

- **pwd** → Muestra el directorio actual
- **cd** → Cambia el directorio
- **rm** → Borra un fichero
- **ps** → Muestra los procesos del sistema
- **top** → Muestra los procesos que más están consumiendo del sistema
- **df** → Muestra el espacio de disco usado y disponible
- **whereis** → Muestra la ubicación del ejecutable localizable a través del path
- **at** → Permite planificar procesos para ser ejecutados en un momento concreto
- **shift** → Desplaza todos los parámetros una posición a la izquierda, eliminando el primero

38. El uso del Pipe

En Unix se pueden concatenar las funcionalidades de varios comandos mediante el uso de pipes |. Esto nos permite trabajar con el resultado de un comando anterior como entrada del siguiente.

El siguiente fragmento de código cuenta cuántos ficheros hay en un directorio. Para ello mostramos en líneas separadas todos los elementos del mismo, y luego contamos las líneas mediante un comando que conoceremos más adelante

```
ls -lrt|wc -l
```

39. grep

El **grep** es un comando que permite realizar búsquedas de patrones en un conjunto de ficheros pasados por parámetro. Es uno de los comandos más potentes y útiles. Algunas de sus opciones son:

- **-i** → Permite realizar búsquedas ignorando mayúsculas y minúsculas
- **-l** → Devuelve el nombre del fichero sobre el que ha habido coincidencias
- **-c** → Cuenta el total de coincidencias
- **-n** → Nos devuelve los números de líneas en los que se han encontrado coincidencias
- **-v** → Nos devuelve las líneas **que no** tienen ese patrón buscado

Además, grep puede trabajar con **expresiones regulares**, y existe una implementación del **grep** focalizada en las expresiones regulares llamada **egrep**

40. find

El comando **find** permite buscar ficheros en el sistema de ficheros del SO. Es un comando muy potente, puesto que permite buscar por fechas, realizar tratamiento con los positivos obtenidos, buscar por conjuntos de permisos...

Algunos ejemplos de opciones para invocar al find son:

- -mtime, -cmin → Permite buscar ficheros que hayan sido modificados en los rangos pasados como parámetros al mtime
- -print → Muestra los resultados por la salida estándar
- -maxdepth → Define a qué profundidad de subdirectorios realizará la búsquedas
- -name → El nombre de los ficheros que ha de buscar

El siguiente fragmento de código buscaría todos los ficheros *sh cuya fecha de modificación sea 1 día o menos y que estén en nuestro directorio

```
find . -maxdepth 1 -name "*sh" -mtime -1 -print
```

41. sed

El comando **sed** nos permite buscar y remplazar textos. Este comando puede trabajar con expresiones regulares, mas tarde volveremos sobre él, pero ahora vamos a conocer su comportamiento básicos

El comando **sed** recibe como parámetro una cadena de texto con la cadena de texto a buscar y la cadena con la que se quiere reemplazar, y unas opciones.

Las opciones más comunes del sed son /g (global), /I (Ignorar mayúsculaes) y /p (imprimir las líneas modificadas, se usa con -n). Existen algunas otras opciones no estándares como /e (ejecutar el resultado)

Vamos a transformar el siguiente fichero

```
linea primera  
otra linea  
linea 3
```

Usamos el comando **sed** indicando que queremos cambiar **linea** por **CAMBIO**

```
sed "s/linea/CAMBIADO/g" fichero_entrada.txt
```

El resultado es el siguiente

```
CAMBIADO primera  
otra CAMBIADO  
CAMBIADO 3
```

42. sort

El comando **sort** ordena los elementos recibidos. Puede ordenar tanto alfabeticamente como numéricamente, y tanto ascendente como descendente.

- -n → Ordena numéricamente
- -r → Ordena descendentemente

El siguiente script leerá el fichero con números y los ordenará de mayor a menor por pantalla

```
sort -nr fichero_con_numeros.txt
```

43. head y tail

Los comandos **head** y **tail** sirven para mostrar los n primeros o últimos caracteres de un fichero pasado como parámetro.

```
head -3 fichero.txt
```

```
tail -3 fichero.txt
```

44. uniq

El comando **uniq** filtra las líneas de un fichero agrupando elementos contiguos iguales. Esto no agrupa elementos iguales que estén separados.

Suele usarse junto al **sort** para asegurarse de que los elementos iguales están adyacentes

- -c Devuelve el número de ocurrencias de un elemento

Si tenemos este fichero

```
uno
cuatro
cinco
seis
uno
uno
```

Y ejecutamos un **uniq -c fichero.txt** obtendremos

```
1 uno
1 cuatro
1 cinco
1 seis
2 uno
```

Si repetimos esto mismo con **sort fichero.txt | uniq -c**

```
1 cinco
1 cuatro
1 seis
3 uno
```

45. rev

El comando **rev** invierte el texto de cada línea. Por ejemplo, para un fichero como el mostrado

```
linea primera
otra linea
linea 3
la cuarta y penultima
quinta y ultima
```

Ejecutar un **rev fichero.txt** devolvería el siguiente resultado

```
aremirp aenil
aenil arto
3 aenil
amitlunep y atrauc al
amitlu y atniug
```

46. cut

El comando **cut** nos permite quedarnos con unas posiciones concretas de cada línea. Se especifica como parámetro **-c** indicando de qué posición a qué posición queremos quedarnos

Sobre el fichero siguiente

```
linea primera
otra linea
linea 3
```

Ejecutamos **cut -c3-7** para quedarnos con los caracteres que van de la posición 3 a la 7

nea p
ra li
nea 3

Pero existen otros comandos para trabajar por campos

- -c → Trabaja con caracteres
- -f → Trabaja con campos
- -d → Permite cambiar el delimitador de campos (por defecto el tabulador)

Así, para el siguiente fichero

```
linea primera  
otra linea  
linea 3
```

Si ejecutamos el siguiente comando **cut -d ' ' -f2 fichero.txt**, obtendremos

```
primera  
linea  
3
```

47. nohup

El comando **nohup** nos permite evitar que un proceso pare su ejecución al cerrarse nuestro shell. Redirigirá la salida a un fichero llamado **nohup.out** y seguirá ejecutándose en el sistema aunque nuestra conexión se cierre.

Es muy útil si estamos ejecutando cosas en remoto y queremos evitar que un script se detenga a medias por un problema de conexión

48. expr

El comando **expr** nos permite operar con números y variables, y es mucho más potente que el comando **let**. Su sintaxis es:

```
expr operando1 operador operando2
```

Para poder trabajar con él en un shell debemos capturar el valor devuelto

```
MiVar=`expr 2 + 3`
```

Permite operandos aritméticos, lógicos y de cadena

Aritméticos

- $+$ → Suma
- $-$ → Resta
- $\backslash*$ → Multiplica
- $/$ → Divide
- $\%$ → Módulo

Lógicos

- $\<$ → Menor
- \leq → Menor o igual
- $\>$ → Mayor
- \geq → Mayor o igual
- $=$ → Igual
- \neq → Distinto
- \mid → OR
- $\&$ → AND

Cadenas de texto

- `length` → Devuelve el tamaño de la cadena (`expr length Texto`)
- `substr` → Devuelve la sub cadena (`expr substr Texto pos_inicial longitud`)
- `index` → Devuelve la posición del texto en la cadena (`expr index Texto buscado`)
- `match` → Análogo a `index` pero trabajando con expresiones regulares

49. Prácticas

- Listar en el fichero de municipios-calles todos los municipios existentes
- Ordenar los municipios en función del que más calles tenga al que menos, y mostrar tan sólo los 30 primeros
- Generar un array con tantas posiciones como parámetros, y que guarde cada parámetro en la posición que corresponda. Luego lo muestre por pantalla (usar el comando `shift`)

Una posible solución:

```
cat municipios-calles.txt |cut -d ';' -f2|sort|uniq
```

```
cat municipios-calles.txt |cut -d ';' -f2|sort|uniq -c |sort -nr|head -30
```

- Repetir la operación, pero asegurándonos previamente que "Madrid" "Asturias" y "Barcelona" no salgan en el resultado y sustituyendo la palabra "de" por "-"

Una posible solución:

```
grep -v "Asturias" municipios-calle.txt|grep -v "Madrid"|grep -v "Barcelona"|cut -d ';' -f2|sort|uniq -c |sort -nr|sed "s/ de / - /g"|head -30
```

```
egrep -v "Asturias|Madrid|Barcelona" municipios-calle.txt|cut -d ';' -f2|sort|uniq -c |sort -nr|sed "s/ de / - /g"|head -30
```

50. Señales y Traps

1. ¿Qué es una señal en Unix?
2. Tratamiento de señales en Unix
3. Practicas

51. ¿Qué es una señal en Unix?

Las señales son un mecanismo básico de comunicación del SO con los procesos. las señales existentes se pueden mostrar mediante el commando **kill -l**

```
[usuario@localhost Datos]$ kill -l
 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
```

Figure 12. Listado de señales disponibles en CentOS.

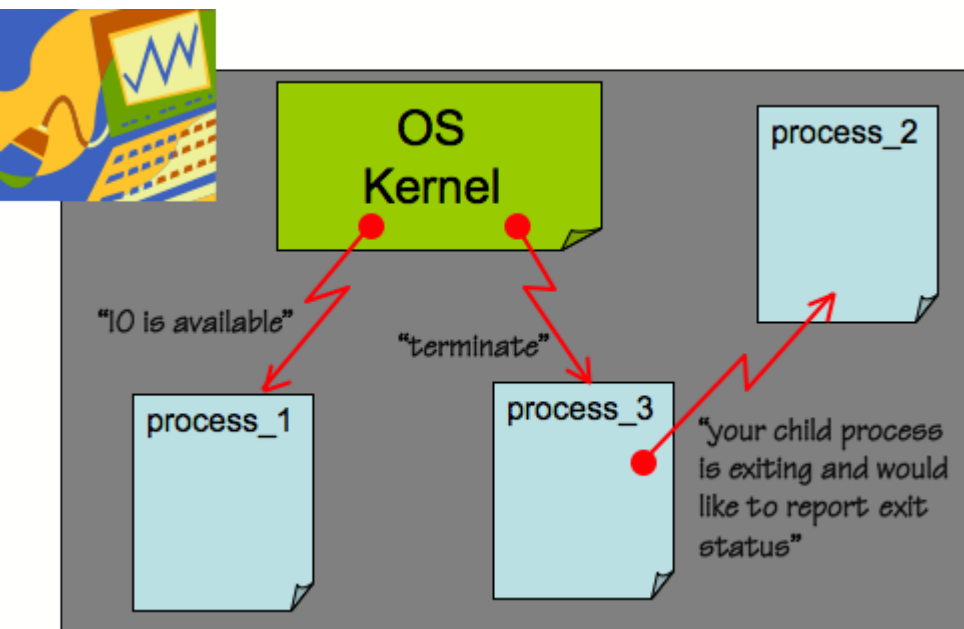


Figure 13. Esquema de uso de señales.

52. Tratamiento de señales en Unix

Vamos a hacer una pequeña práctica, vamos a crear un script con un bucle while infinito, que cada 2 segundos muestra un mensaje en pantalla

```
#!/bin/bash

while [ 0 -ne 1 ]
do
    sleep 2
    echo "sigo durmiendo"
done
echo "sali del while"
exit 0
```

Si ejecutamos este script va a estar corriendo de manera continua hasta que lo abortemos (CTRL+C)

Vamos a intentar mandarle unas señales a nuestro proceso shell. Debemos arrancar una nueva consola para seguir trabajando.

En primer lugar necesitamos obtener su PID, ¿Cómo lo haríais?

Una opción para obtener el PID es usar el comando **ps**

```
ps -af  
ps -edaf|grep nombre_de_mi_script
```

Ahora, mediante el comando **kill** podemos mandar señales a este proceso (usando su PID). Probemos a mandar varios tipos de señales a nuestro proceso. ¿Qué sucede?

```
kill -9 PID
```

Esto funciona de esta manera debido a que cuando un proceso recibe una señal busca las instrucciones de cómo tratarla. Si no existen estas instrucciones o **traps** el proceso no sabe qué hacer con esta señal, y para su ejecución.

El comando **trap** recibe *al menos* dos parámetros, las instrucciones a realizar y las señales que debe "atrapar"

```
trap 'instrucciones' ID_SEÑAL
```

Vamos a repetir el ejercicio anterior, pero con ciertas modificaciones sobre el script original

```
#!/bin/bash

trap 'echo "Signal atrapada"' 1
trap 'echo "Hemos parado un tipo 2"' 2
trap 'echo "tipos 3,4 o 5 paradas"' 3 4 5

while [ 0 -ne 1 ]
do
    sleep 2
    echo "sigo durmiendo"
done
echo "sali del while"
exit 0
```

Probemos ahora a lanzar nuestro script y atacarlo con distintas señales. ¿Qué sucede ahora?

Repitamos el ejercicio anterior pero añadiendo más señales al tratamiento, asegurémonos de incluir la señal 9.

¿Qué sucede ahora?

53. Practicas

Vamos a hacer una practica para asimilar los conceptos aprendidos. Queremos hacer un proceso que, cada segundo gener un número aleatorio, y lo escupa en 3 ficheros distintos.

- Nuestro proceso al levantarse comprueba si existe o no un fichero llamado "semaforo.txt". Si este fichero existe dará un error y no se ejecutará.
- El fichero "semaforo.txt" debe contener el PID de nuestro proceso.
- Tendremos 3 salidas distintas, peques.txt para números menores de 10.000, medianos para números ≥ 10.000 y < 20.000 y grandes, para ≥ 20.000 .
- Estos ficheros de salida, en caso de existir al comienzo de la ejecución, deben ser borrados.
- Entre la generación de un número y el siguiente, debe pasar 1 segundo. Debemos realizar esta tarea 50 veces.
- Al finalizar el proceso, debemos borrar el fichero "semaforo.txt" para dejar libre el "slot" de ejecución del proceso.
- Añadir tratamiento de señales del tipo 1-8 para borrar el fichero pero salir del proceso cuando se reciben
- la variable de entorno especial **\$RANDOM** nos proporciona un número aleatorio entre 0 y 32767

```
#!/bin/bash

#atrapo signals
trap 'echo "Signal atrapada"; rm -fr semaforo.txt; echo "Semaforo Borrado"; exit' 1 2
3 4 5 6 7 8

#Compruebo que no hay un proceso en ejecucion
if [ -s semaforo.txt ]
then
    echo "Existe un proceso en ejecucion"
    exit
fi

#genero semaforo con el PID
echo $$ > semaforo.txt

#Declaro variables y borro ficheros de ejecuciones anteriores
rm -fr peques.txt medianos.txt grandes.txt
Vueltas=50

while [ $Vueltas -gt 0 ]
do
    NumRandom=$RANDOM
    if [ $NumRandom -lt 10000 ]
    then
        echo $NumRandom >> peques.txt
    elif [ $NumRandom -lt 20000 ]
    then
        echo $NumRandom >> medianos.txt
    else
        echo $NumRandom >> grandes.txt
    fi
    let Vueltas=Vueltas-1
    echo "Numero escupido $NumRandom"
    sleep 1
done

#borro semaforo
rm -fr semaforo.txt
```

54. Comando getopt

1. ¿Para qué sirve el comando getopt?
2. Uso de getopt
3. Prácticas

55. ¿Para qué sirve el comando **getopts**?

Como hemos visto en temas anteriores, en los shells es importante tratar los parámetros con los que se invoca el proceso.

Para script sencillos cogíamos directamente los parámetros mediante el tratamiento de las variables especiales \$1, \$2...

El comando **getopts** trata de abstraer esta lógica para hacerla transparente al programador. Su finalidad es tratar las distintas opciones enviadas como parámetros.

Los parámetros que no vengan como opciones **NO** serán procesados por **getopts**

getopts procesará los parámetros enviados como opciones, pero tendremos que especificar previamente qué hacer con cada opción.

Internamente dispone de 3 variables con las que podemos trabajar

- OPTIND → Posición del argumento que vamos a tratar (Ojo, el argumento, no la opción)
- OPTARG → Argumento que vamos a tratar (en caso de que la opción venga seguida de un argumento)
- OPTERR → Indica al shell que muestre los errores del **getopts**. por defecto a 1, poner a 0 si no queremos mostrarlos (recomendado)

56. Uso de getopts

El comando **getopts** es fácilmente trabajable desde un bucle **while**, ya que cada vez que se llama devuelve una nueva opción, y devuelve falso si no hay más elementos.

Su sintaxis es la siguiente:

```
getopts :lista_de_posibles_parametros[:] variable_destino
```

Es decir, tenemos que poner la lista de posibles opciones, y detrás de cada opción debemos incluir el carácter : si esta opción ha de venir acompañada de un argumento.

Esto se ve mucho mas claro con un ejemplo. Imaginemos que queremos crear un script que puede recibir como parámetro las opciones -c, -p y -r y que todas ellas van acompañadas de un valor

```
#!/bin/bash

while getopts :c:p:r: opt
do
    echo "var <$opt>"
    echo "OPTIND <$OPTIND>"
    echo "OPTARG <$OPTARG>"
    echo "OPTERR <$OPTERR>"
done
```

Generemos este script y probemos a invocarlo

```
mi_script.sh -c 3 -r HOLA -p 89
```

Si quisieramos que sólo la opción p viajara con valores, haríamos lo siguiente

```
#!/bin/bash

while getopts :cp:r opt
do
    echo "var <$opt>"
    echo "OPTIND <$OPTIND>"
    echo "OPTARG <$OPTARG>"
    echo "OPTERR <$OPTERR>"
done
```

Generemos este script y probemos a invocarlo

```
mi_script.sh -c -r -p 89
```

El **getopts** suele usarse junto a la clausula **case**, ya que así podemos definir el comportamiento de cada opción de manera cómoda

```
#!/bin/bash

while getopts :c:p:r: opt
do
    case $opt in
        c)
            #tratamiento opcion c
            echo "Valor opcion c $OPTARG"
            ;;
        p)
            #tratamiento opcion p
            echo "Valor opcion p $OPTARG"
            ;;
        r)
            #tratamiento opcion r
            echo "Valor opcion r $OPTARG"
            ;;
        \?)
            echo "Invalid option: -$OPTARG" >&2
            ;;
    esac
done
```

57. Prácticas

Generemos un script, que cargue nuestra librería de funciones, y realice los siguientes pasos

- Muestre siempre por pantalla un "Bienvenido Usuario"
- Admita las opción **-m valor** que imprimirá por pantalla una matriz de NxN elementos (siendo N el valor pasado como parámetro a la opción)
- Admintna la opción **-h** para mostrar la hora del sistema

```
#!/bin/bash

. ../02_funciones/libreria.sh

echo "Bienvenido Usuario"

while getopts :m:h opcion
do
    case $opcion in
        m)
            recuadro $OPTARG
            ;;
        h)
            hora_minuto_segundo
            echo "Hora: <$Hora>"
            ;;
        \?)
            echo "Opcion no valida"
            ;;
        esac
    done
```

58. Expresiones regulares

1. Expresiones Regulares
2. Patrones de expresiones regulares
3. Ejemplos de expresiones regulares
4. Prácticas con egrep y sed

59. Expresiones Regulares

Las expresiones regulares son secuencias de caracteres que identifican un patrón de búsqueda. Son de suma importancia a la hora de trabajar con cadenas de caracteres.

Por ejemplo, validar que la cadena que estamos leyendo es un número, validar un e-mail introducido por una web o verificar que el código postal de una dirección de UK corresponde con la notación esperada.

Aunque matematicamente las expresiones regulares están definidas de manera única, su implementación en distintos lenguajes o procesos que trabajan con ellas pueden variar.

Las expresiones regulares que vamos a ver tienen uso tanto en el comando **grep**, como en el **sed** como en el **awk**.

60. Patrones de expresiones regulares

- `char` → Un carácter o conjunto de caracteres busca coincidencias consigo mismo
- `*` → Busca 0 o más ocurrencias de un patrón definido previamente
- `\+` → Busca 1 o más ocurrencias de un patrón definido previamente
- `\?` → Busca 0 o 1 ocurrencias de un patrón definido previamente
- `{ i }` → Busca exactamente *i* ocurrencias
- `{ i,j }` → Busca entre *i* y *j* ocurrencias
- `{ i, }` → Busca *i* o más ocurrencias
- `(regexp)` → agrupa una expresión regular y permite usar back reference

- `.` → Cualquier carácter incluido New Line
- `^` → Comienzo de línea
- `$` → Final de línea
- `[lista]` → Busca cualquier carácter de la lista
- `[^lista]` → Busca cualquier carácter NO presente en la lista
- `regexp1|regexp2` → Busca coincidencias con `regexp1` o con `regexp2`
- `\digit` → Busca coincidencias con el resultado de la agrupación parentizada especificada
- `\n` → Busca el carácter de nueva línea
- `\char` → Escapa el carácter para buscarlo como literal. Usable con `$ * . [\ ^`
- `() {}` → en `sed` se deben escapar para entenderlos como carácter de control, en `egrep` no

61. Ejemplos de expresiones regulares

- `'abcdef'`
- `'a*b'`
- `'a?b'`
- `'a\+b\+'`
- `'.'`
- `'^main.*\(.*)'`
- `'^#'`
- `'\\$'`

- 'abcdef' → Busca la cadena "abcdef"
- 'a*b' → Busca la cadena que contenga una b, precedida de 0 o más a ("aaaab", "ab", "b"...)
 - 'a?b' → Busca la cadena "ab" o "b"
- 'a+b\+' → Busca la cadena formada por 1 o más a seguidas de 1 o más b ("aaaaab", "ab", "abbbb", "aaabbb"...)
 - '.' → Busca la cadena formada por todos los caracteres de la línea
 - '^main.*\(.*\)' → Busca la cadena que empieza por "main" y después contiene unos paréntesis ("main()", "main123(234)"...)
 - '^#' → Busca la cadena que empieza por #
 - '\\\$' → Busca la cadena que acaba por \

- '\$'
- '[a-zA-Z0-9]'
- '^(\.*)\n1\$'
- '.{9}A\$'
- '^.{15}A'
- '(\.1(\.2(\.3'
- '^(\.)(\.)\321\$'

- '\$' → Busca la cadena que comienza por \$
- '[a-zA-Z0-9]' → Busca cualquier letra o dígito
- '^.*\n1\$' → busca dos líneas iguales consecutivas
- '.{9}A\$' → Busca líneas que acaben en A y tengan 9 caracteres delante
- '^.{15}A' → Busca líneas que empiecen por 16 caracteres de los cuales el último sea una A
- '(.1(.2(.3)' → Busca líneas que tengan al menos 6 letras iguales 2 a 2 (la 1-2, la 3-4 y la 5-6).
(bookkeeper)
- '^.(.)(.321\$' → Busca palíndromos de 6 letras

62. Prácticas con egrep y sed

Sobre el fichero de datos (municipios-calles.txt) queremos realizar las siguientes búsquedas con **egrep**

- Buscar todas las líneas que contengan la cadena AYUD
- Buscar todas las líneas que contengan la cadena ayud
- Buscar todas las líneas que comiencen con 32
- Buscar todas las ocurrencias que empiecen con 18, 16 o un número entre 30 y 39
- Buscar todas las ocurrencias que posean en el segundo campo un ciudad de entre 3 y 5 caracteres, y que finalicen la línea con D
- Procesar un fichero que contenga IPs y eliminar las IPs que NO tengan una notación correcta

```
egrep "AYUD" municipios-calles_convertido.txt
```

```
egrep "ayud" municipios-calles_convertido.txt
```

```
egrep "^32" municipios-calles_convertido.txt
```

```
egrep "^(^18|^16|^3[0-9]{1})" municipios-calles_convertido.txt
```

```
egrep "^[0-9]*;[a-zA-Z]{3,5};.*D$" municipios-calles_convertido.txt
```

```
egrep "^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9]?[0-9])$"
```

Ahora vamos a trabajar con algunos de los resultados anteriores, transformándolos con sed

- Buscar todas las líneas que contengan la palabra AYUD y cambiarlo por ayud
- Buscar todas las líneas que comiencen con 32 y cambiarlo por XX
- Buscar todas las ocurrencias que empiecen con 18, 16 o un número entre 30 y 39 y cambiarlos
18 → XX, 16 → YY 30-39 → ZZ
- Buscar todas las ocurrencias que posean en el segundo campo un ciudad de entre 3 y 5 caracteres, y que finalicen la línea con D y cambiar la ciudad por CIUDAD

```
cat municipios-calles_convertido.txt|egrep "AYUD"|sed 's\AYUD\ayud\g'
```

```
egrep "^32" municipios-calles_convertido.txt |sed 's/^32/XX/g'
```

```
egrep "^(^18|^16|^3[0-9]{1})" municipios-calles_convertido.txt |sed 's/^18/XX/g'|sed  
's/^16/YY/g'|sed 's/^3[0-9]\{1\}/ZZ/g'
```

```
egrep "^[0-9]*[a-zA-Z]{3,5};.*D$" municipios-calles_convertido.txt|sed 's/^([0-  
9]*;)[a-zA-Z]{3,5}\(\(;.*D\)/\1CIUDAD\2/g'
```


Hacer una pequeña función para validar parámetros en nuestros scripts, que diga si un parámetro es numérico o no.

Añadirla a nuestra librería de funciones y probarla.

La función guardará en PARAMETRO el valor numérico del parámetro recibido, y la dejará vacía en caso de que no sea numérico (Así podremos comprobar en nuestros scripts si ha habido error al evaluar el parámetro)

```
#!/bin/bash
```

```
Comprueba_Numero()
```

```
{  
    PARAMETRO=$1  
    if [ `echo $PARAMETRO|egrep "^[0-9]*$" ` ]  
    then  
        echo "Es numero <$PARAMETRO>"  
    else  
        echo "NO es numero <$PARAMETRO>"  
        PARAMETRO=""  
    fi  
}
```

```
Comprueba_Numero 334
```

```
Comprueba_Numero 3h1
```

```
Comprueba_Numero hola
```

```
Comprueba_Numero 123
```

```
exit 0
```

Sobre el fichero de datos (municipios-calles.txt) queremos realizar las siguientes búsquedas con **egrep**

- Eliminar todos los registros constituído exclusivamente por líneas vacías o que sólo contengan blancos
- Buscar todos las líneas que tengan 4 letras seguidas, siendo la primer y la tercera iguales, y la segunda y la cuarta iguales
- Buscar todos las líneas que tengan 4 letras seguidas, siendo la primer y la segunda iguales, y la tercera y la cuarta también, y que además después tengan un abrir y cerrar paréntesis (no necesariamente adyacente)
- Buscar todos las líneas que tengan 4 letras seguidas, siendo la primer y la tercera iguales, y la segunda y la cuarta iguales, exclusivamente en el campo quinto
- Contar los positivos de cada ejercicio

```
egrep -v "^[ ]*$"
```

```
egrep "([a-zA-Z])([a-zA-Z])\1\2" municipios-calles_convertido.txt
```

```
egrep "([a-zA-Z])\1([a-zA-Z])\2.*\(.*\)" municipios-calles_convertido.txt
```

```
egrep "^[\0-9]+;[\^;]*;[\^;]*;[\^;]*;[\^;]*([a-zA-Z])\1([a-zA-Z])\2" municipios-calles_convertido.txt
```

```
comando | wc -l
```

63. awk

1. ¿Qué es el AWK y qué es awk?
2. Estructura de un programa en AWK
3. Operadores en AWK
4. Bucles en AWK
5. El comando print
6. Un poco de práctica
7. Variables pre-construidas en AWK
8. Un poco de práctica 2
9. Arrays asociativos
10. Prácticas

64. ¿Qué es el AWK y qué es awk?

- **AWK** es un lenguaje de programación diseñado para procesar datos basados en texto, ya sean ficheros o flujos de datos
 - Esta orientado a realiza un tratamiento línea a línea para un flujo de datos
 - Es capaz de **trocear** los campos de cada registro de manera similar al comando **cut**
 - Hace uso de un tipo especial de arrays llamado **arrays asociativos**
 - También hace uso de **expresiones regulares**
 - **OJO**, en AWK debemos acabar las líneas con un ;
- **awk** es el comando que interpreta dicho lenguaje

65. Estructura de un programa en AWK

```
BEGIN { acción }  
/patrón/ { acción }  
END { acción }
```

- El bloque **BEGIN** ejecuta las acciones descritas **antes** de que los datos comiencen a ser procesados
- El bloque principal ejecuta las acciones para todas aquellas líneas **filtradas** por el patrón
- El bloque **END** ejecuta acciones después de que todos los datos sean procesados
- Todas las partes del programa pueden aparecer más de una vez, por ejemplo puede haber dos bloques principales cada uno con su patrón

Un pequeño ejemplo de AWK

```
#!/bin/awk -f  
BEGIN {  
    print "hola mundo";  
    exit;  
}
```

```
#!/bin/bash  
awk 'BEGIN {  
    print "hola mundo";  
    exit;  
}'
```

En AWK no se puede trabajar directamente con las variables de entorno del sistema, sin embargo podemos crear y definir cualquier variable al llamar al proceso **awk**. Esto se hace mediante la opción **-v nombre_var=\$env_var**

```
#!/bin/bash

awk -v mi_var=$Var_DelSistema 'BEGIN {
    print "Mi " mi_var;
    exit;
}'
```

66. Operadores en AWK

Operadores aritméticos:

- + → Suma (2+3 → 5)
- - → Resta (5-3 → 2)
- * → Multiplicación (2*1 → 2)
- / → División (6/2 → 3)
- % → Módulo (7%1 → 1)
- espacio → Concatenación (7 3 → 73)
- = → Asignación (Al igual que en C pueden usarse combinaciones con los operadores aritméticos +=, /= ...)

Operadores lógicos:

- `==` → Igual
- `!=` → Distinto
- `>=` → Mayor o igual
- `>` → Mayor
- `<=` → Menor o igual
- `<` → Menor
- `!` → Negación
- `&&` → Y lógico
- `||` → O lógico

Operadores para expresiones regulares:

- `~` → El patrón encaja
- `!~` → El patrón no encaja

67. Bucles en AWK

Bucle **if**

```
if (condición)
{ instrucciones}
else if (condición)
{ instrucciones}
else { instrucciones}
```


Bucle **while**

```
while (condicion)
{ instrucciones}
```

Bucle **for**

```
for ( expresión; condición; expresión )  
{ instrucciones}
```

```
for ( variable in array )  
{ instrucciones}
```

- La palabra reservada **break** fuerza la salida del bucle en el que estamos trabajando
- La palabra reservada **continue** sale de la iteración actual del bucle y nos deja al comienzo de la siguiente
- La palabra reservada **next** cambia el flujo del programa para pasar a procesar la siguiente línea
- La palabra reservada **exit** finaliza un script AWK

68. El comando print

El comando **print** muestra por pantalla la salida que especifiquemos

El comando **printf** muestra por pantalla la salida que especifiquemos con el formato especificado. Su sintaxis es similar al C

```
BEGIN {  
    i=1;  
    print "estoy imprimiendo i " i;  
    printf "estoy imprimiendo i " i "\n";  
    print "estoy imprimiendo i %d",i;  
    printf "estoy imprimiendo i %d\n",i;  
}
```

69. Un poco de práctica

- Realizar un programa en AWK que muestre por pantalla los cuadrados de los 15 primeros números enteros
- Realizar un programa en AWK que diga cuántas líneas ha procesado en total

Una posible solución:

```
BEGIN {
    i=1;
    while (i <= 15) {
        printf "El cuadrado de " i " es " i*i "\n";
        i = i+1;
    }
}
```

```
BEGIN {lineas=0;}
{lineas++;}
END {print "Total de lineas: " lineas;}
```

70. Variables pre-construidas en AWK

Como ya hemos mencionado anteriormente, en **AWK** se puede trabajar con los distintos campos de una línea, que automáticamente están delimitados por un *tabulado*.

Esta configuración por defecto se puede cambiar con la opción **-F 'Delimitador'** al llamar al comando `awk`

```
awk -F ':' '{print $2;}'
```

`awk` asigna las variables `$1`, `$2` ... a cada campo, siendo `$0` la línea entera

Además de las indicadas, las siguientes variables vienen definidas en AWK:

- FS → Field Separator, contiene el caracter delimitador que se está usando
- OFS → Output Field Separator, es el separador que pone por defecto AWK al imprimir varios campos
- NF → Number of Fields, es el total de campos que hay en una línea
- NR → Number of Records, te dice el número de registro
- FNR → File Number of Records, te dice el número de registro del fichero procesado
- RS → Record Separator, el separador de líneas
- ORS → Output Record Separator, el separador de líneas para salidas

71. Un poco de práctica 2

- Generar un programa en AWK que muestre los campos 2 y 4 de un fichero de pagos
- Generar un programa en AWK que muestre sólo los registros pares

Una posible solución:

```
awk -F ';' '{print $2 $4;}'
```

```
'{  
    if (NR%2==0)  
    {print $0;}  
}'
```

72. Arrays asociativos

Los arrays asociativos funcionan como arrays normales con una diferencia fundamental, sus índices en lugar de números pueden ser cadenas de texto. Internamente funcionan como una tabla hash, cada elemento del array tiene una clave (índice) y un valor.

```
mi_array[0]=0;  
mi_array["uno"]=1;  
mi_array["dos"]="dos";  
...
```

Para obtener sus claves, podemos iterar directamente sobre el array

```
for (clave in mi_array)  
{  
    instrucciones;  
}
```

73. Prácticas

- Realizar un programa que agrupe y acumule los pagos del fichero de pagos en categorías, ignorando la cabecera del fichero, y mostrando al final los acumulados por cada grupo, el total de pagos realizados y la cantidad total pagada

Una posible solución:

```
#!/bin/bash

cat pagos.txt|awk -F ';' '
BEGIN {
    total_pagos=0;
    total_acumulado=0;
}
{
    if (NR>1)
    {
        mi_array[$2]+=$3;
        total_pagos++;
        total_acumulado+=$3;
        # printf "valor %s es %.2f\n",$2,mi_array[$2];
    }
}
END {
    for (clave in mi_array)
    {
        printf "valor %s es %.2f\n",clave,mi_array[clave];
    }
    printf "El total de pagos ha sido <%d> por un total de
    <%.2f>\n",total_pagos,total_acumulado;
}
'
```

Existe un proceso bancario que cada noche genera un fichero con las cuentas que hay en el sistema. Cada día, se guarda el listado de cuentas nuevas en el fichero FICHERO_NEW.txt, y el fichero del día anterior en FICHERO_OLD.txt

Sólo queremos incluir, los nuevos registros, y en ningún momento incluir registros que haya desaparecido.

Fichero NEW

```
0317770950000009352  
0317770950000014312
```

Fichero OLD

```
0317770950000089072  
0317770950000009352
```

Fichero RESULTADO

```
0317770950000014312
```


Una posible solución:

```
awk 'BEGIN{}  
FNR==NR {  
    a[$0];  
    next;  
}  
!($0 in a) {  
    print $0;  
}  
' FICHERO_OLD.txt FICHERO_NEW.txt > NUEVAS.txt
```

74. Monitorización y optimización

1. Monitorización y ajuste del sistema I/O
2. Monitorización y ajuste CPU
3. Monitorización y ajuste Memoria Virtual
4. Monitorización y ajuste de red

75. Monitorización y ajuste del sistema I/O

En muchas ocasiones, un rendimiento pobre de acceso a disco puede lastrar el rendimiento de un servidor.

Conocer cómo funcionan los discos duros y sus características es vital para cualquier administrador de un sistema operativo, de cualquier índole.

Un sistema de ficheros se compone de una secuencia de bloques lógicos, cada uno de los cuales tiene un tamaño fijo, generalmente múltiplo de 512 bytes.

Elegir el tamaño de bloque para un sistema es importante, cuanto mayor sea, mejores tiempos de transferencia vamos a encontrarnos, pero tendremos un menor aprovechamiento del disco.

La estructura de un sistema de ficheros en **UNIX System V** se compone de 4 partes:

- bloque boot (o de arranque)
- superbloque
- tabla de i-nodos
- bloques de datos

El bloque de arranque (o **boot**) es un bloque situado al inicio del sistema de ficheros, y puede contener el código de arranque.

Este código es un pequeño programa que se encarga de buscar el sistema operativo y cargarlo en memoria para inicializarlo

El **superbloque** describe el estado de un sistema de ficheros.

Alguna información que contiene, incluye:

- Tamaño del sistema de ficheros
- Lista de bloques libres disponible
- Índice del siguiente bloque libre en la lista de bloques libres
- Tamaño de lista de i-nodos, total de i-nodos libres y la lista
- Índice al siguiente i-nodo libre de la lista de i-nodos

Los **i-nodos** (o nodos índice) contienen información para cada fichero del sistema, pero no sus datos.

- Identificador del propietario del fichero (usuario y grupo)
- Tipo del fichero (datos, directorios, tuberías o especiales)
- Tipo de acceso al fichero (el nivel de permisos para el usuario, el grupo y otros)
- Número de enlaces del fichero (un fichero puede tener varios enlaces simbólicos)
- Datos de acceso al fichero (Fecha de creacion, último acceso..)
- Tamaño del fichero
- Entradas para los bloques de dirección (los bloques que forman un fichero no tienen por qué estar contiguos)

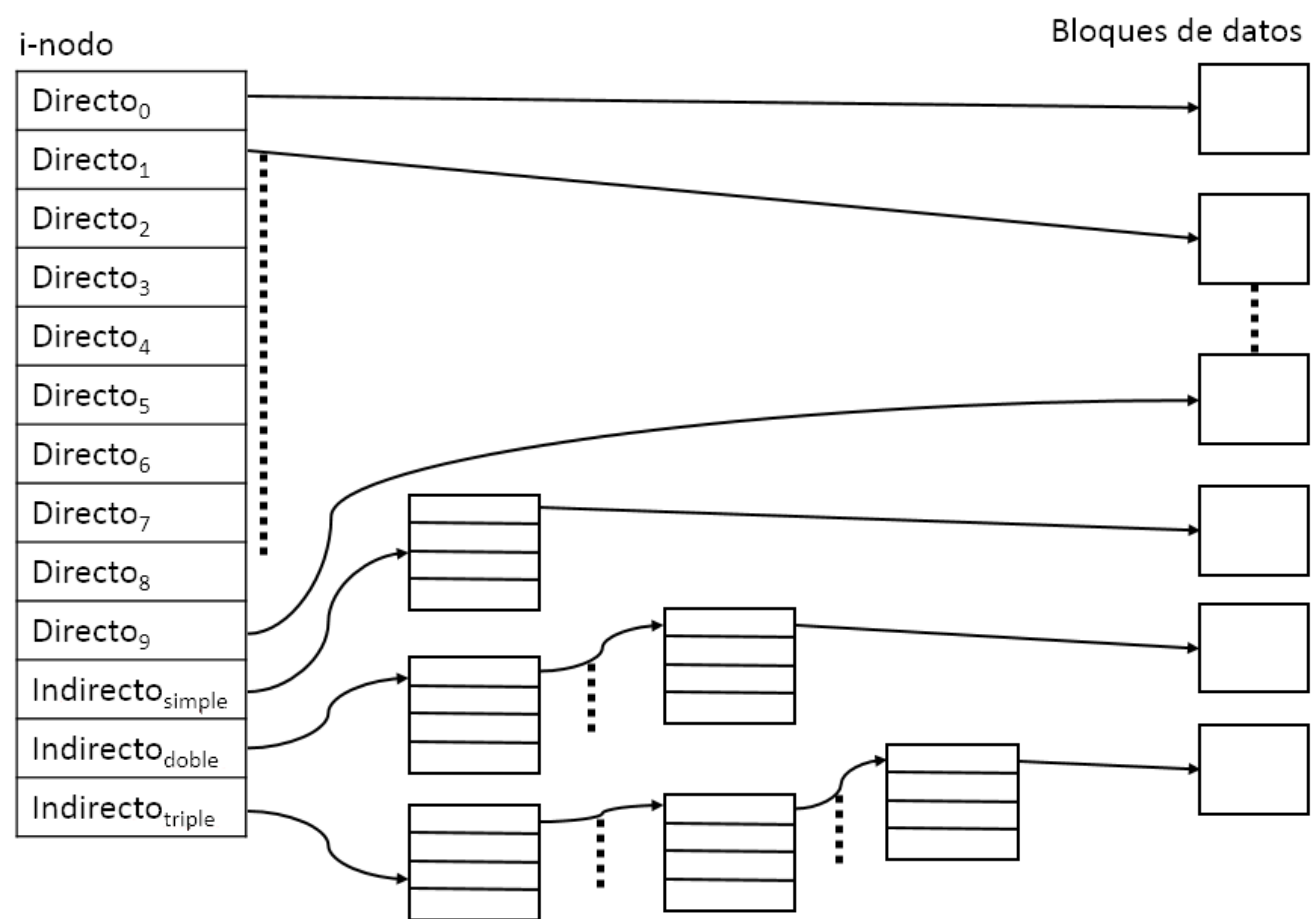
Los **bloques de datos** ocupan la mayor parte del sistema de ficheros. Son una serie de bloques de un tamaño determinado que van a ser referenciados a través de los **i-nodos**.

Es decir, un fichero contiene una descripción informada en el **i-nodo**, y unos datos formados por una serie de bloques. Estos bloques están referenciados a través de los **i-nodos**.

Un **i-nodo**, en **System V** posee 13 entradas para bloques de datos

- 10 entradas directas (contienen la dirección de un bloque de datos)
- 1 entrada indirecta simple (apunta a un bloque de datos que se va a interpretar como una lista de entradas directas)
- 1 entrada indirecta doble (apunta a un bloque de datos que se va a interpretar como una lista de entradas indirectas simples)
- 1 entrada indirecta triple (apunta a un bloque de datos que se va a interpretar como una lista de entradas indirectas dobles)

El siguiente dibujo pretende mostrar la tabla de direcciones de un i-nodo



Las herramientas de monitorización juegan un papel vital a la hora de detectar la ubicación de nuestro cuello de botella, y poder atacar la causa de dicho problema.

Concretamente, vamos a ver las siguientes herramientas:

- IOSTAT
- IOTOP
- DSTAT
- IOPING

Antes de nada, vamos a instalar las herramientas

```
yum install iotop
```

```
yum install dstat
```

```
wget ftp://104.219.172.11/gentoo/distfiles/ioping-0.6.tar.gz  
tar zxvf ioping-0.6.tar.gz  
cd ioping-0.6  
make  
make install  
cp ioping /usr/bin/
```


Empecemos con **iostat**:

iostat es una herramienta usada para informar de las lecturas/escrituras realizadas en un disco durante un intervalo. Para ello, recoge estadísticas de uso de disco, espera una cantidad determinada de tiempo y las recoge nuevamente, mostrando las diferencias en la salida.

Estas estadísticas incluyen las estadísticas de **CPU**, y las estadísticas del uso del disco.

Las estadísticas de disco proporcionan información como "el total de peticiones de I/O por segundo (tps)", la tasa de lecturas y escrituras (KB_read/s y KB_wrtn/s)

El uso de **iostat** es sencillo, se invoca usando los siguientes parámetros:

- **n** → Número de segundos entre lectura y lectura
- **-d** → Para no mostrar las estadísticas de CPU*
- **-y** → Descarta la primera lectura, ya que por lo general es poco útil (muestra los datos desde el inicio del sistema)

```
iostat -y 1
```

Nuestra siguiente herramienta de monitorización es **iotop**.

iotop muestra la actividad del disco en tiempo real, y los procesos que están realizando Entrada/Salida, junto al ancho de banda que están usando dichos procesos.

Como parámetros, se suele usar la opción **-o**, que muestra sólo la actividad de los procesos que realizan Entrada/Salida, omitiendo el resto de procesos.

```
iotop -o
```

Si al lanzarlo vemos por qué hay un campo "total" y otro "actual", y por qué son diferentes, la respuesta es debido al File System Caching.

dstat es una versión más amigable del **iostat**, y puede mostrar mucha más información, no sólo el ancho de banda, y permite de hecho la inclusión de plugins hechos por terceros.

Los parámetros para invocarse son muchos, pero vamos a mostrar algunos comunes:

- -c → Muestra estadísticas de cpu
- -d → Muestra estadísticas de disco
- -g → Muestra estadísticas de paginación
- -r → Activa estadísticas de E/S
- -n → Activa estadísticas de red

```
dstat -cd
```

Por último, tenemos una herramienta muy útil, el **ioping**.

ioping medidor de latencias de disco, especialmente útil para comprobar si los problemas de rendimiento de disco son debidos a problemas de red, de hardware...

Para hacer uso de **ioping**, primero debemos consultar los discos existentes con **lsblk**

```
[root@Miskatonic ~]# lsblk
```

NAME	MAJ:MIN	RM	SIZE	RO	TYPE	MOUNTPOINT
sr0	11:0	1	1024M	0	rom	
sda	8:0	0	40G	0	disk	
└─sda1	8:1	0	500M	0	part	/boot
└─sda2	8:2	0	39,5G	0	part	
└─vg_miskatonic-lv_root (dm-0)	253:0	0	35,6G	0	lvm	/
└─vg_miskatonic-lv_swap (dm-1)	253:1	0	3,9G	0	lvm	[SWAP]

Una vez hecho esto, podemos usar el **ioping** sobre uno de los discos mostrados

Las opciones para invocar a **ioping** no son muchas, cabe resaltar tal vez la opción "-c n" para realizar sólo n peticiones.

```
ioping /dev/sda1
```

Números bajos (menos de 1 ms), y sin mucha varianza, son indicadores de un sistema sano.

Si detectamos que nuestros accesos a discos no son muy buenos, existen un par de enfoques que nos pueden permitir mejorar el rendimiento:

- Limitar el número de ficheros que un usuario puede tener abiertos
- Configurar una caché de lectura específica llamada **read ahead**

La caché **read ahead** es muy útil en discos magnéticos, pero carece de utilidad en discos **SSD**. La idea es leer siempre más de lo solicitado, pues es muy probable que el siguiente dato que se vaya a leer estuviera justo detrás.

Existen tres formas de configurarla:

- A través de **/sys** (/sys/class/block/sda/queue/read_ahead_kb)
- Mediante **blockdev**
- Mediante **hparm**

Con **hparm** hay que usar la opción **-a** que permite consultar (o modificar) el **readahead**

Con **blockdev** hay que usar la opción **--getra** para consultar y **--setra N** para modificar.

```
hdparm -a /dev/sda
blockdev --setra 128 /dev/sda
blockdev --getra /dev/sda
```


El comando **ulimit** nos permite cambiar el número de ficheros que puede tener un usuario abierto.

Hay que tener en cuenta que prácticamente todo es un fichero en **Unix**, puesto que un pipe, es un fichero, un socket, también lo es...

Tener este número alto puede dar lugar a problemas, pero reducirlo demasiado puede hacer que nuestros usuarios no estén operativos.

Los usos comunes de ulimit son **-a** para consultar y **-n X** para cambiar el número de ficheros que se pueden tener abiertos.

```
ulimit -a
```

```
ulimit -n 2048
```

76. Monitorización y ajuste CPU

Un **programa** es una colección de instrucciones y datos que se encuentran almacenados en un fichero, y que son susceptibles de ser ejecutados.

Aunque en un sistema **UNIX** puede haber ejecutables de muchos tipos (por ejemplo, **shells**), vamos a centrarnos en los procesos creados por el compilador de **C**.

Un programa consta de las siguientes partes:

- Cabeceras que describen atributos del fichero
- Un bloque donde se encuentran las instrucciones en lenguaje máquina
- Un bloque con la información de los datos que tienen que ser inicializados al ejecutarse (como por ejemplo el espacio reservado en memoria para dicho proceso)
- Otras secciones (como las tablas de símbolos)

Cuando un **programa** es leído de disco por el núcleo y cargado en memoria para ejecutarse, se convierte en un **proceso**. Además de una copia del programa, existe información de control añadida por el núcleo:

- **Segmento de texto** → contiene las instrucciones en código máquina de nuestro programa
- **Segmento de datos** → Contiene los datos que deben ser inicializados al arrancar el proceso (e.g. variables globales y estáticas)
- **Segmento de pila** → Se gestiona por el núcleo, es un conjunto de **marcos de pila**, que es donde se almacena los parámetros de una función, las variables locales y la información necesaria para restaurar un marco anterior

Es interesante conocer que, ya que en los sistemas **UNIX** se pueden ejecutar procesos tanto como usuario como en modo root, el núcleo del sistema maneja sendas pilas por separado, para evitar problemas de seguridad.

Al permitir el multiproceso, **UNIX** cuenta con un **planificador**, que es la parte del núcleo encargada de gestionar la **CPU** y decidir qué proceso pasa a hacer uso de la misma en un determinado instante.

Cuando un proceso en **UNIX** (a excepción del primero, proceso número 0, del que hablaremos en breve) son creados mediante una llamada al método **fork**.

El proceso que llama al **fork** se denomina **padre** y el creado tras dicha llamada, lo denominamos **hijo**.

Al crear un **proceso**, el sistema le asigna un número que actúa como identificador, que no va a modificarse y no va a compartir con ningún otro proceso. A este identificador le llamamos **PID** (Process Identification).

El proceso cuyo **PID** es el **0**, es un proceso especial, ya que es el que se crea al arrancar el sistema. Al arrancar crea un proceso mediante un **fork**, y acto seguido se convierte en el proceso **intercambiador**, y será el encargado de la gestión de memoria virtual.

El proceso creado con ese primer **fork** se llama **init**, su **PID** es **1**, y es el encargado de arrancar todos los demás procesos del sistema (**/etc/inittab**)

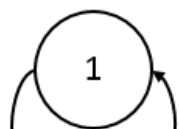
Durante la vida de un proceso, este puede pasar por diferentes estados, cada uno con unas características determinadas. Conocer bien estos estados es fundamental para poder aprender a gestionar procesos.

En una primera aproximación, podemos resumir los estados posibles como:

- **1** → El proceso se está ejecutando en modo usuario
- **2** → El proceso se está ejecutando en modo supervisor
- **3** → El proceso no se está ejecutando, pero está listo para ser ejecutado (seleccionable por el planificador)
- **4** → El proceso está durmiendo (no puede continuar con su ejecución, e.g. está esperando una E/S)

El siguiente gráfico muestra estos estados y la transición entre ellos

Ejecutándose en modo usuario



Llamada al sistema / Interrupción

Retorno

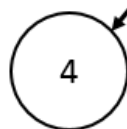
Ejecutándose en modo supervisor



Interrupción / retorno de interrupción

Dormir

Orden de ejecución



Durmiendo



Listo para ejecutarse

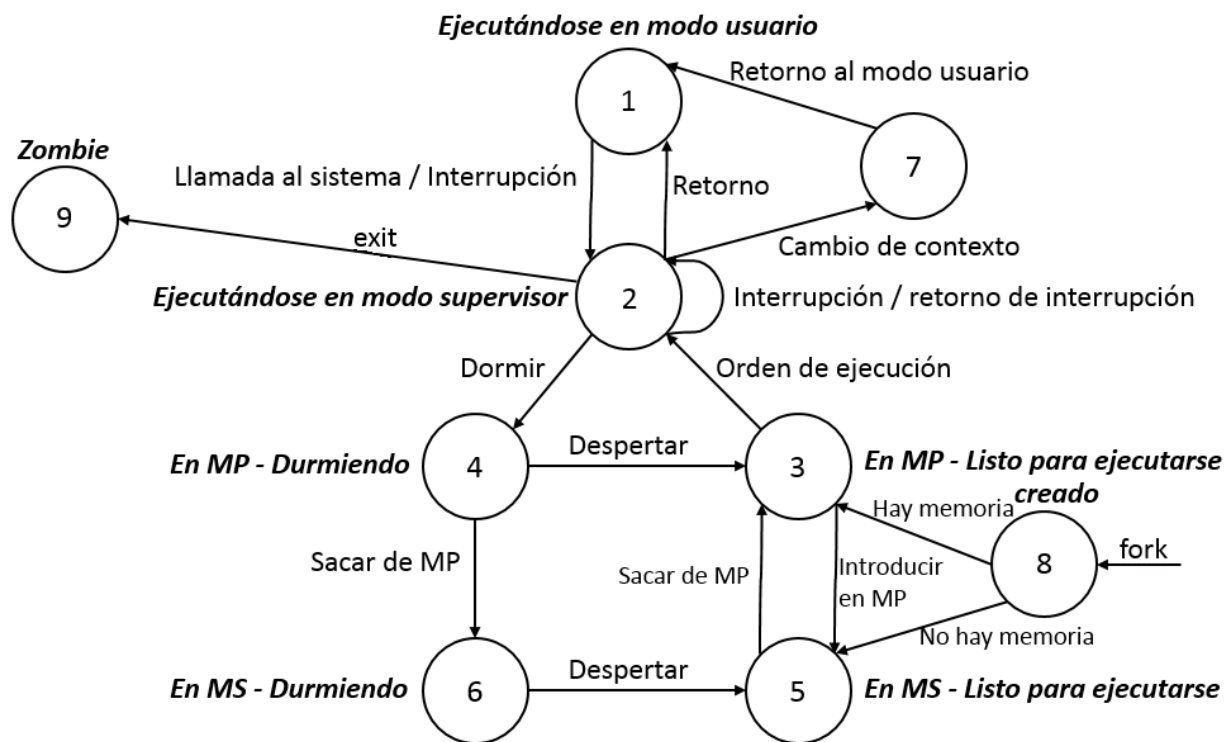
Despertar

Si bien la simplificación que hemos visto es de utilidad, es una simplificación de los estados existentes, pero por ejemplo puede pasar que un proceso en estado listo para ejecutarse no esté cargado en la memoria principal, si no en la secundaria, por lo que antes hay que realizar la carga en memoria de dicho proceso

- **1** → El proceso se está ejecutando en modo usuario (sin cambios)
- **2** → El proceso se está ejecutando en modo supervisor (sin cambios)
- **3** → El proceso no se está ejecutando, pero está listo para ser ejecutado y está cargado en memoria
- **4** → El proceso está durmiendo pero está cargado en memoria

- 5 → El proceso está listo para ser ejecutado, pero el **planificador** debe cargarlo en memoria
- 6 → El proceso está durmiendo y el intercambiador ha decidido descargar el proceso de la memoria principal a una secundaria
- 7 → El proceso está volviendo del modo supervisor, pero el núcleo se apodera del proceso y hace un cambio de contexto, otro proceso pasa a ejecutarse en modo usuario
- 8 → El proceso acaba de ser creado, existe pero no está ni preparado para ejecutarse ni durmiendo. Este es el estado inicial de todos los procesos (salvo el 0)
- 9 → El proceso ha finalizado y ya no existe, pero deja ciertos datos para el proceso padre (como el código de salida y los tiempos de ejecución)

El siguiente esquema muestra de manera completa los estados y transiciones definidos.



Todo proceso tiene asociado una entrada en una tabla de procesos y un área de usuario. Estas estructuras describen el estado de proceso facilitan al núcleo su control.

La tabla de procesos contiene entre otros:

- Estado del proceso
- Información para localizar el proceso (memoria principal, secundaria, área de usuario)
- Identificadores de usuario (UID) y de proceso (PID)
- Señales del sistema que ha recibido pero que todavía no ha tratado
- Contadores para contabilizar el tiempo de uso de cpu, recursos...

En el área de usuario existe información sólo necesaria cuando el proceso se está ejecutando, entre otros:

- Puntero a la entrada de la tabla de procesos correspondiente a dicha área de usuario
- Identificadores de usuario que determinan privilegios de acceso
- Contadores para saber cuánto tiempo lleva en modo usuario y/o modo supervisor
- Array con cómo responde el proceso a las señales recibidas
- Terminal asociado al proceso (si existe)
- Registro de errores en llamadas al sistema, y otro con el resultado de las llamadas
- Directorio actual y los descriptores de ficheros abiertos actualmente

Ahora que ya conocemos cómo funciona un proceso en memoria, vamos a conocer las herramientas con las que podemos monitorizar nuestro sistema.

Vamos a ver algunos ejemplos de dos de ellas:

- `top` → Permite visualizar los procesos que más están impactando en el sistema
- `ps` → Permite conocer el estado de todos los procesos existentes

El uso de **top** es muy sencillo, al ejecutarlo nos muestra los procesos que más penalizan al sistema y su uso de CPU y MEM.

Se va actualizando constantemente, y hay que lanzar una señal de interrupción para volver a obtener el control

```
top - 13:19:43 up 2:12, 4 users, load average: 0.00, 0.00, 0.00
Tasks: 179 total, 1 running, 178 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.1%us, 0.1%sy, 0.0%ni, 99.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 3916740k total, 1033696k used, 2883044k free, 64328k buffers
Swap: 4063228k total, 0k used, 4063228k free, 670932k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
42	root	20	0	0	0	0	S	0.3	0.0	0:04.27	ata_sff/0
1	root	20	0	19356	1532	1224	S	0.0	0.0	0:03.77	init
2	root	20	0	0	0	0	S	0.0	0.0	0:00.02	kthreadd
3	root	RT	0	0	0	0	S	0.0	0.0	0:00.03	migration/0
4	root	20	0	0	0	0	S	0.0	0.0	0:00.01	ksoftirqd/0
5	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	stopper/0
6	root	RT	0	0	0	0	S	0.0	0.0	0:00.01	watchdog/0
7	root	RT	0	0	0	0	S	0.0	0.0	0:00.03	migration/1
8	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	stopper/1
9	root	20	0	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/1
10	root	RT	0	0	0	0	S	0.0	0.0	0:00.01	watchdog/1
11	root	RT	0	0	0	0	S	0.0	0.0	0:00.03	migration/2
12	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	stopper/2
13	root	20	0	0	0	0	S	0.0	0.0	0:00.00	ksoftirqd/2
14	root	RT	0	0	0	0	S	0.0	0.0	0:00.01	watchdog/2
15	root	RT	0	0	0	0	S	0.0	0.0	0:00.03	migration/3
16	root	RT	0	0	0	0	S	0.0	0.0	0:00.00	stopper/3

El comando **ps** por la contra, es bastante más complejo de usar, pero también mucho más flexible. **ps** nos va a mostrar el estado de nuestros procesos, y algunas de las invocaciones más comunes son:

```
*ps aux* -> información detallada de todos los procesos  
*ps -f -u user* -> Muestra sólo los procesos del usuario *user*  
*ps -f -p 3150,3123* -> Muestra los procesos con el PID 3150,3123*
```

Si queremos asignar distintas prioridades a nuestros procesos, podemos hacer uso de **renice**.

En **Unix**, las prioridades de los procesos van del **-20** al **19**. Con **renice** podemos cambiar la prioridad de un proceso.

Hay que tener en cuenta que la máxima prioridad es el **-20** siendo la más baja prioridad el **19**

Para usar **renice** hay que especificar el **PID** y la prioridad que queremos asignarle:

```
renice -20 -p 31524
```

Podemos consultar la prioridad de un proceso con **ps -o ni PID**

```
ps -o ni 31524
```

77. Monitorización y ajuste Memoria Virtual

La memoria virtual (o memoria de intercambio, **swap**), es la solución usada por el sistema operativo para cuando no se dispone de suficiente memoria **RAM**, que pasa por volcar parte de la información a disco, de donde será subida posteriormente en caso de necesidad.

Aunque el uso de la memoria **swap**, en principio, es una buena idea, en caso de estar mal implementada, puede suponer un uso excesivo de disco duro, y por lo tanto, lastrar el rendimiento del sistema.

Existen de hecho varias aplicaciones que recomiendan desactivar por completo la **swap**, puesto que prefieren un fallo y saber que hay un problema a poder experimentar un problema de rendimiento.

En resumidas cuentas, la **swap** es útil cuando sirve de apoyo a nuestra **RAM**, pero un uso inadecuado lastra el rendimiento del sistema.

Es por ello que estudiaremos cómo activarla y desactivarla, y cómo configurarla.

La prima opción que podemos querer tomar es desactivar la memoria swap. Para ello existen dos comandos que nos van a ayudar:

- `swapoff` → Permite desactivar la memoria **swap** (en uno o varios discos)
- `swapon` → Permite activar la memoria **swap** (mismo uso que `swapoff`)

Para usarlo, podemos indicar el device sobre el que aplicarlo, o desactivarlos/activarlos todos mediante la opción **-a**

```
swapoff -a  
swapon -a
```


Para gestionar el uso de la **memoria virtual**, existe una propiedad del sistema llamada **swappiness**. Por defecto, este valor es **60**. Se puede consultar con el siguiente comando:

```
cat /proc/sys/vm/swappiness
```

Este número, que puede estar entre 0 y 100, indica qué porcentaje de memoria RAM debe estar lleno para utilizar la memoria virtual (el mínimo es 1, ya que 0 significa que desactivamos la memoria virtual por completo).

Una configuración de 1 significa que hasta que no lleguemos al 99% de uso de nuestra RAM no haremos uso de la memoria virtual, mientras que un valor de 100 significa que usaremos constantemente la memoria virtual.

Podemos modificar este valor cambiando el valor de la propiedad **swappiness**

```
sysctl -w vm.swappiness=50
```

78. Monitorización y ajuste de red

Para monitorizar y configurar nuestra red podemos hacer uso de las herramientas **ethtools**.

Para empezar, vamos a mostrar la configuración de un interfaz de red:

```
ethtool eth0
```

Entre otras opciones, podemos ver los tres tipos de **Duplex** existentes.

- Full → Puede enviar Y recibir paquetes al mismo tiempo (útil para cuando está conectado a un switch)
- Half → Puede enviar O recibir (útil para cuando está conectado a un hub)
- auto-negotiation → El dispositivo elige cuándo usar uno u otro

Podemos cambiar por ejemplo el parámetro de **auto-negotiation**, para ello paramos el interfaz de red, realizamos el cambio y lo levantamos de nuevo:

```
ifdown eth0  
ethtool -s eth0 autoneg off  
ethtool eth0 #Para consultar el cambio  
ifup eth0
```

Cambiar la velocidad de la tarjeta de red:

```
ifdown eth0  
ethtool -s eth0 speed 100 autoneg off  
ethtool eth0 #Para consultar el cambio  
ifup eth0
```

También podemos mostrar las estadísticas de red de un dispositivo específico:

```
ethtool -S eth0
```

O incluso encender el **LED** de una tarjeta para poder identificarla en el servidor

```
ethtool -p eth0
```

Por último, otra herramienta de mucha utilidad para estudiar el uso de la red es **netstat**, que nos permite consultar qué procesos están haciendo uso de la red:

- **netstat -nap** → Nos muestra el puerto usado y el pid del proceso
- **netstat -a** → Muestra todos los puertos TCP y UDP abiertos
- **netstat -at** → Sólo los TCP
- **netstat -au** → Sólo los UDP
- **netstat -l** → Listening connections

79. Gestión e instalación de paquetes

1. YUM
2. RPM
3. Instalación mediante compilación

80. YUM

yum es un paquete administrador de software (software package manager). Es una muy útil herramienta para instalar, actualizar y remover paquetes junto con sus dependencias en distribuciones Linux basados en RPM (redhat, fedora, centos, etc.). Automáticamente determina las dependencias necesarias y lo que debe realizarse para instalar paquetes.

yum apareció inicialmente en la distribución **yellowdog**, y de ahí viene su nombre **yellowdog updater modified**. No es la única herramienta existente, otros sistemas (**Debian** o **Ubuntu**) tienen **apt-get** cuyo uso es similar.

yum va a buscar el software en los repositorios que tengamos configurados. Los siguientes archivos son usados por **yum**:

- /etc/yum.conf → archivo de configuración.
- /etc/yum.repos.d → directorio de depósitos o repositorios.
- /var/log/yum.log → archivo de bitácora.

Su uso más simple para instalar un paquete es como se muestra:

```
yum install dialog
```

Vamos a conocer los usos más comunes del comando **yum**:

- **yum -y install paquete** → Instala el paquete sin pedir confirmación
- **yum -y update** → Actualiza los paquetes del sistema
- **yum -y update paquete** → Actualiza sólo el paquete indicado
- **yum list** → Muestra todos los paquetes disponibles para instalación, actualización o ya instalados.
- **yum list installed** → Lista de todos los paquetes instalados en el sistema.
- **yum remove paquete** → Remueve el paquete indicado.
- **yum search paquete** → Busca el paquete entre todos los posibles para instalar (puede ser palabra parcial)

81. RPM

RPM fue creado por **Red Hat** en 1995, is es a día de hoy el sistema de gestión de paquetes usado con **LSB** (Linux Standard Base).

Existen cinco comandos básicos que podemos ejecutar con **rpm**:

- **Install** → Para instalar paquetes RPM.
- **Remove** → Desinstalar o borrar paquetes RPM.
- **Upgrade** → Se usa para actualizar un paquete RPM.
- **Verify** → Se usa para verificar un paquete RPM.
- **Query** → Se usa para consultar datos de un paquete RPM.

Cada una de estas opciones tiene a su vez sus opciones propias. Vamos a ver algunos ejemplos de uso del comando **rpm**.

Una cosa bastante útil, antes de instalar nada, es comprobar la firma de un paquete **RPM**, esto se puede hacer con la opción **--checksign**

```
rpm --checksig jdk-7u71-linux-x64.rpm
```

El propio comando debe devolver algo como lo mostrado a continuación:

```
jdk-7u71-linux-x64.rpm: sha1 md5 BIEN
```


Una vez comprobada la firma, vamos a realizar la instalación. El parámetro **-i** se encarga de ello. El siguiente comando realizaría la instalación de un paquete **RPM**

```
rpm -ivh jdk-7u71-linux-x64.rpm
```

Hemos añadido un par de parámetros más:

- **-v** → Verbose, para mostrar información por pantalla.
- **-h** → Hashes, para mostrar el avance de la instalación mediante una barra que se completa con almohadillas.

Vamos ahora a comprobar las dependencias de un paquete. Para ello, tenemos que lanzar una instrucción de tipo query **-q**:

```
rpm -qpR jdk-7u71-linux-x64.rpm
```

Al igual que antes, hemos usado un par de opciones adicionales:

- **-p** → Muestra todas las dependencias de las que este paquete depende.
- **-R** → Muestra todas las dependencias de las que dependen de este paquete.

También podemos realizar una instalación de un paquete ignorando las dependencias, útil cuando creemos que están satisfechas las dependencias pero **rpm** cree que no. Para ello usamos la opción **--nodeps**, junto con un comando de instalación:

```
rpm -ivh --nodeps jdk-7u71-linux-x64.rpm
```

Si sencillamente queremos saber si tenemos un paquete instalado, tan sólo tenemos que usar la opción **-q** (query), y nos dirá si el paquete está instalado (NOTA: el nombre del paquete no tiene por qué ser el nombre del archivo rpm)

```
rpm -q jdk-1.7.0_71-fcs.x86_64
jdk-1.7.0_71-fcs.x86_64
```

Se puede hacer una búsqueda parcial, poniendo sólo parte del paquete

```
rpm -q jdk
jdk-1.7.0_71-fcs.x86_64
```

Si queremos saber cuales son todos los ficheros instalados con un paquete, podemos usar la opción Query List **-ql** (también admite búsquedas parciales):

```
rpm -ql jdk-1.7.0_71-fcs.x86_64
/etc
/etc/.java
/etc/.java/.systemPrefs
/etc/.java/.systemPrefs/.system.lock
/etc/.java/.systemPrefs/.systemRootModFile
/etc/init.d/jexec
/usr
/usr/java
/usr/java/jdk1.7.0_71
/usr/java/jdk1.7.0_71/COPYRIGHT
[...]
```

Si queremos consultar a todos los paquetes, con **-qa** (query all) podemos hacerlo. La siguiente instrucción muestra todos los paquetes instalados

```
rpm -qa
```

Pero también podemos preguntar sólo por los últimos, añadiendo la opción **--last** (que además nos da la fecha)

```
rpm -qa --last
```

Si ya tenemos un paquete instalado, y queremos actualizarlo a una versión más moderna, podemos usar la opción **-U** (upgrade), que mantiene una copia de la versión anterior por si hay que volver a ella

```
rpm -Uvh jdk-7u71-linux-x64.rpm
```

Para borrar un paquete previamente instalado hay que usar la opción **-e** (erase), recordad que se usa el nombre del paquete:

```
rpm -ev jdk-1.7.0_71-fcs.x86_64
```

En ocasiones no se puede eliminar un paquete debido a que otros poseen dependencias hacia él. Podemos forzar la eliminación usando la opción **--nodeps**

```
rpm -ev --nodeps jdk-1.7.0_71-fcs.x86_64
```


En otras ocasiones tenemos que saber a qué paquete pertenece un archivo que hemos encontrado en nuestro sistema. La opción **-qf** (query file) nos resuelve estos problemas

```
rpm -qf /usr/java/jdk1.7.0_71/release
```

Si queremos consultar la información de un paquete, tenemos que usar la opción **qi** (query information), de la siguiente manera

```
rpm -qi jdk-1.7.0_71-fcs.x86_64
```

También se puede consultar la información del paquete antes de instalar, en el fichero **rpm**, con la opción **-qip** (query info package):

```
rpm -qip jdk-7u71-linux-x64.rpm
```

Existen más opciones, como verificar un paquete **rpm** (`rpm -Vp paquete.rpm`), o importar claves GPG (`rpm --import clave`). No nos centraremos más en ello, pero si vamos a explicar cómo se puede reconstruir la base de datos de paquetes, ya que en ocasiones se corrompe, y puede mermar la funcionalidad del sistema. Para solucionarlo, tenemos que borrar la base de datos existente, y pedir que se reconstruya:

```
cd /var/lib
rm __db*
rpm --rebuilddb
rpmdb_verify Packages
```

82. Instalación mediante compilación

Existe una tercera vía para instalar software en nuestros sistemas, y es la compilación del código fuente.

Por lo general, esto implica que debemos bajarnos el código fuente de un repositorio, y a partir de él generar el programa que necesitamos.

Esta vía es la más compleja de las tres vistas, en primer lugar porque la gestión de dependencias recae por completo en nuestros hombros, en segundo lugar porque es la más delicada (ya que podríamos tener compiladores con distintas versiones o estándares que hagan que esto falle) y por último por que cada software se compila de una forma predeterminada (aunque por lo general, todos depende de ficheros makefile con distintas metas definidas).

Como resumen, **buscad siempre un fichero LEEME, README o INSTALL, y leedlo exhaustivamente.**

83. Planificación

1. Introducción
2. nohup
3. at
4. crontab
5. dialog

84. Introducción

En múltiples ocasiones, tenemos que planificar procesos para que sean ejecutados a una hora o en un momento concretos, o delegar los procesos para que no dependan del terminal sobre el que estamos trabajando.

En el presente tema, vamos a estudiar 3 herramientas:

- `nohup` → Cuya finalidad es continuar con la ejecución del proceso incluso aunque la comunicación con el terminal se pierda
- `at` → Para una planificación puntual de una ejecución
- `crontab` → Para procesos que se deben ejecutar de manera repetitiva

85. `nohup`}

El comando **`nohup`** permite mantener la ejecución de un comando (el cual le pasamos como un argumento) pese a salir de la terminal (logout), ya que hace que se ejecute de forma independiente a la sesión.

Básicamente, lo que hace es ignorar la señal HUP (señal que se envía a un proceso cuando la terminal que lo controla se cierra), esto implica que aunque cerremos la terminal, el proceso se siga ejecutando.

Es de especial utilidad cuando estamos realizando conexiones remotas y podemos perder la conexión de red.

Su uso es muy simple (pero no debemos olvidar que para lanzarlo en segundo plano debemos usar el `&`)

```
nohup proceso &
```

Para probar el uso de este comando, haremos lo siguiente. Crearemos un script que mediante el comando **sleep** ejecute 60 instrucciones una por segundo.

Primero lo ejecutaremos mediante una conexión remota por putty o similar, y en mitad del proceso cerramos la ventana

La segunda vez haremos lo mismo, pero habiéndolo ejecutado con el comando **nohup**

86. at

El comando **at** permite la planificación puntual de uno o varios comandos en una fecha determinada. Su uso es muy sencillo, en primer lugar invocamos el comando **at** especificando la hora y el día en el que queremos realizar esta ejecución

```
at 21:35 Tue
```

Después, nos pedirá los comandos a ejecutar. Para finalizar la captura de comandos hay que usar Ctrl+d

```
at> echo "hola"  
at> ^D
```

Otra forma de ejecutarlo es mediante el comando **echo**, pasándole de esta manera el comando a ejecutar

```
echo "proceso"|at 21:35 Tue
```

A la hora de planificar podemos:

- Especificar sólo la hora (en formato 24h o 12h con AM o PM). Se ejecuta hoy a dicha hora
- Especificar hora y el día de la semana. Se ejecutará ese día de la semana a esa hora
- Especificar la fecha completa
- Usar un helper

Algunos ejemplos para que entendamos esto:

```
at 10:00 AM
```

```
at 10:00 Sun
```

```
at 10:00 AM 6/22/2018  
at 10:00 AM 6.22.2018
```

```
at 10:00 AM tomorrow  
at 10:00 AM next month  
at now + 30 minutes  
at now + 1 week  
at now + 2 years  
at midnight
```

87. crontab

De todas las herramientas que hemos visto, **crontab** es la más potente. En ella, podemos llevar un control de todas las tareas planificadas para ser lanzadas de manera más cómoda.

Para planificar tareas con **crontab** tan sólo tenemos que acceder a la hoja de la planificación con permisos de edición

```
crontab -e
```

Esto nos llevará a un fichero de texto que podemos editar, y que permite planificar la ejecución de los procesos

El fichero de planificación de **crontab** tienen los siguientes campos (se pueden especificar varios valores separándolos con comas):

- minuto → Es el primer campo, se especifica el minuto de 0 a 59
- hora → La hora en la que queremos ejecutar esto, de 0 a 23
- día → El día del mes en el que queremos ejecutar el proceso (1-31)
- mes → Mes en el que queremos ejecutar el proceso (de 1 a 12)
- día de la semana → el día de la semana que queremos que ejecute nuestro proceso (0=domingo a 6)
- Comando a ejecutar

Si no queremos especificar algún campo, basta con dejar un asterisco en él, y esto indica que se ejecutará siempre que se cumplan el resto de campos

También se pueden usar unos comandos especiales predefinidos, en cuyo caso sólo habría que poner dicho comando y el proceso a ejecutar. Estos son:

- @reboot → Una única ejecución, al arrancar el pc
- @yearly → Una vez al año (equivale a "0 0 1 1 *")
- @annually → equivale a @yearly
- @monthly → Una vez cada mes (equivale a "0 0 1 * *")
- @weekly → Una vez cada semana (equivale a "0 0 * * 0")
- @daily → Una vez cada día (equivale a "0 0 * * *")
- @midnight → equivale a @daily
- @hourly → Una vez cada hora (equivale a "0 * * * *")

88. dialog

En ocasiones, nos interesa hacer una interfaz gráfica para nuestro script. Existe una herramienta llamada **dialog** que nos permite dotar de interfaz gráfica modo texto a nuestros scripts, cuyo uso es muy sencillo.

En primer lugar, tenemos que instalar dicha utilidad con el siguiente comando:

```
yum install dialog
```

Para invocar un cuadro de dialogo hay que usar la siguiente sintaxis:

```
dialog OpcionesComunes TipoDeCaja "Texto" Alto Ancho OpcionesDeCaja
```

Vamos a ver algunos ejemplos:

Crear una caja con un mensaje y un tamaño específico:

```
dialog --msgbox "Mensaje." 10 50
```

Crear una caja con un mensaje y un título:

```
dialog --title "Titulo" --msgbox "Mensaje." 10 50
```

Crear una caja que presente las opciones **si** y **no** (podemos añadirle título con **--title "texto"** como en el caso anterior)

```
dialog --yesno "¿Te gustan los higos?" 10 50
```

Una vez ejecutado este comando, dejará en **\$?** el valor respondido, siendo **Si** el **0** y **No** el **1** (y **255** si se presiona **Esc** y no se responde)

```
echo $?
```

Otra opción es pedir un texto a un usuario. En este caso, lo que hayamos escrito se mostrará por la salida de error (2), con lo que tendremos que redirigirlo a un fichero para poder leerlo a continuación.

Un ejemplo de uso:

```
dialog --inputbox "Escribe algo" 10 50 2>/tmp/a.txt  
respuesta=`cat /tmp/a.txt`  
echo "Ha escrito $respuesta"
```

Y al igual que pasaba con el cuadro de **Si/No**, devolvemos en **\$?** un **0** si la opción escogida es **Aceptar**, un **1** si es **Cancelar** y **255** si hemos pulsado **Esc**.

Otro cuadro interesante es el **textbox**, que nos permite mostrar y navegar por el contenido de un fichero:

```
dialog --textbox /etc/passwd 10 50
```

También podemos solicitar al usuario un dato, pero no mostrarlo en pantalla (contraseñas y similares)

```
dialog --title "Password" --insecure --clear --passwordbox "Escriba" 10 30 2>  
/tmp/a.txt
```

Ojo, el **--insecure** hace que se muestren asteriscos al escribir, sin ella, no se ve nada al escribir.

Por último, mostraremos unas opciones adicionales, pero no profundizaremos en ellas, aunque las comentaremos por si alguien las quiere buscar:

Menús:

```
dialog --help-button --clear --backtitle "Hola" --title "[ Demo Menubox ]" \  
--menu "Usa arriba, abajo, los números o la letra de tu opcion" 15 50 4 \  
Algo "Haz algo" Otro "Otra cosa" Exit "Sal" 2>/tmp/a.txt
```

Checklists (salen las opciones escogidas separadas por espacio):

```
dialog --checklist "Escoge varias" 10 50 2 \  
"a" "Opcion A" "off" "b" "Opcion B" "off"
```


Radiolist:

```
dialog --radiolist "Escoge una" 10 50 2 \  
"a" "Opcion A" "off" "b" "Opcion B" "on"
```

Progress Bar (lee de la entrada estándar los valores del porcentaje):

```
#!/bin/bash  
declare -i COUNTER=1  
{  
    while test $COUNTER -le 100  
    do  
        echo $COUNTER  
        COUNTER=COUNTER+1  
        sleep 1  
    done  
} | dialog --gauge "Ejecutando rm -fr /" 10 50 0
```

89. Gestión de usuarios

1. Comando Unix para la gestión de usuarios
2. Módulos PAM
3. LDAP y NIS

90. Comando Unix para la gestión de usuarios

La base de la gestión de permisos en **Unix** son los usuarios y grupos. La máscara de permisos es algo que ya hemos estudiado en temas anteriores, pero no nos hemos adentrado en la creación y administración de usuarios y grupos.

Para esta finalidad, nos adentraremos en los siguientes comandos:

- `useradd` → Para añadir un usuario
- `usermod` → Modificar un usuario (poner fecha de expiración de usuario, cambiar su contraseña, forzar un uid...)
- `userdel` → Borrar un usuario
- `passwd` → cambia la contraseña de un usuario
- `adduser` → En algunos sistemas existen unos **helper** que realizan más tareas y nos facilitan la vida.

Cuando creamos un usuario, las configuraciones básicas de dicho usuario se copian del directorio **/etc/skel**. Deentro encontraremos los ficheros iniciales de los usuarios, algunos de los más importantes son:

- **.bash_logout** → Al hacer logout se ejecuta esto
- **.bash_profile** → Cuando un usuario se conecta se ejecuta esto
- **.bashrc** → Cuando un usuario se conecta de manera remota se ejecuta esto

Los grupos son otro tema a considerar en nuestra seguridad, al igual que con los usuarios podemos usar:

- `groupadd` → Crea un grupo nuevo
- `groupdel` → Elimina un grupo
- `groupmod` → Modifica un grupo existente

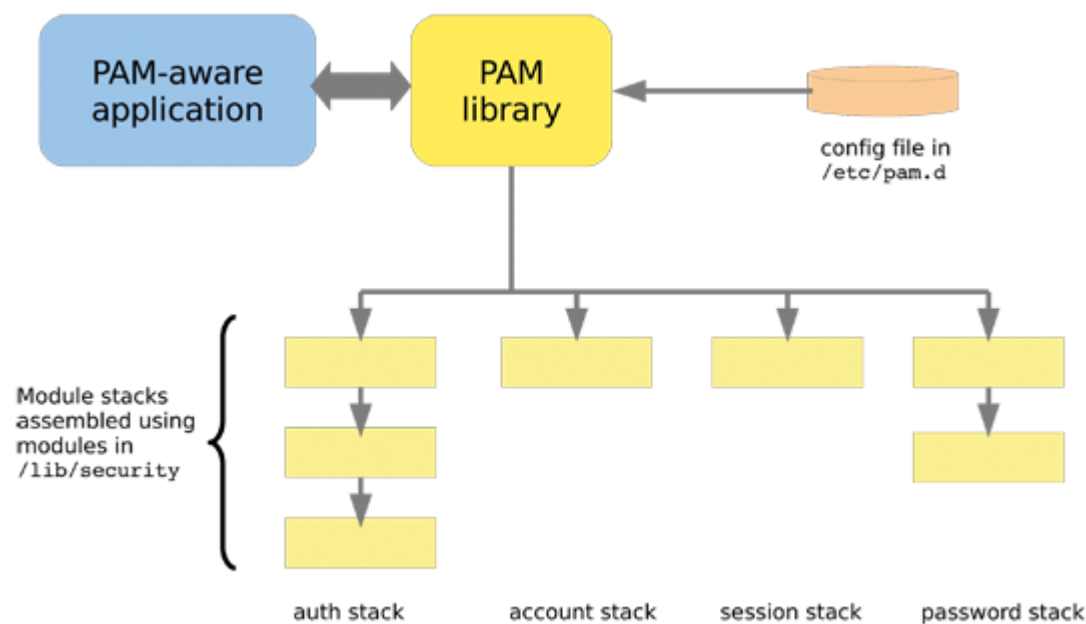
Hay que tener en cuenta que un usuario puede estar en más de un grupo, teniendo permisos de todos ellos.

```
usermod -a -G grupo1 usuario
usermod -a -G grupo2 usuario
```

91. Módulos PAM

PAM (del inglés *pluggable authentication modules*), es un framework que permite a las aplicaciones realizar actividades vinculadas con la autenticación. Las piezas cores de **PAM** son la librería (`libpam`) y los módulos, que son librerías (`.so`) que se encuentran en `/lib/security`

Cada módulo realiza una tarea específica, y una aplicación **PAM-aware** usa una pila de módulos para hacer el trabajo.



PAM reconoce cuatro tipos distintos de actividades relacionadas con la autenticación, que son las siguientes:

- Autenticación → La actividad de proporcionar quién eres aportando credenciales para ello. La forma más típica para hacer esto es mediante el nombre de usuario y contraseña, pero existen más métodos (como tokens OTP, identificadores biométricos...)
- Cuenta → Este módulo es el encargado de, una vez que estás autenticado correctamente, decidir si vamos a dejarte entrar o no. Por ejemplo, un módulo que permite conectarse sólo a ciertas horas o días de la semana sería sin duda de este tipo de categoría.
- Sesión → Esta categoría proporciona los recursos que un usuario puede necesitar durante al iniciar una sesión. Un ejemplo sería montar un directorio remoto para él, poner unos límites de uso, mostrar un mensaje de bienvenida, etc...
- Password → Este módulo permite actualizar las credenciales de un usuario (generalmente, su password)

¿Quién usa **PAM**?

Cualquier programa que necesite autenticar usuarios, controlar las conexiones, proporcionar recursos o actualizar credenciales puede usar **PAM**. Algunos ejemplos podrían ser:

- **login**, el programa que te permite conectar a un terminal shell
- **gdm** (Gnome Display Manager) te permite conectarte a un escritorio gráfico
- **su** te permite empezar un nuevo terminal con una nueva identidad
- **passwd** te permite cambiar el password de un usuario

Usar **PAM** permite separar los detalles de cómo se realizan las actividades de autenticación de las aplicaciones que necesiten realizar dichas actividades. Esto nos permite de manera relativamente sencilla cambiar nuestra política de seguridad para añadir nuevos mecanismos de autenticación, tan sólo reconfigurando **PAM**.

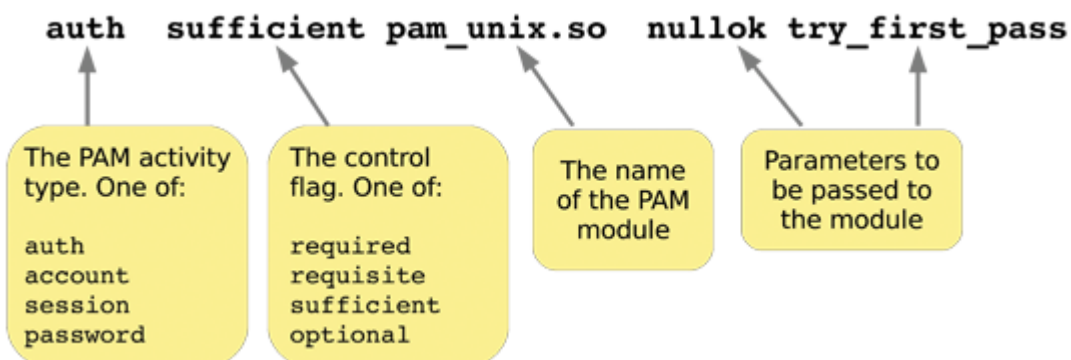
¿Cómo podemos saber si un programa hace uso de **PAM**? En teoría, la manera más cómoda de mirar esto es viendo el fichero de configuración de **PAM**, ubicados en **/etc/pam.d**

```
ls /etc/pam.d
```

Pero también podemos mirar las dependencias con las librerías **PAM** a través del comando **ldd** (que nos dice qué dependencias tiene nuestro ejecutable)

```
ldd /bin/login | grep libpam
```

La pila de módulos que una aplicación usa para permitir cada uno de las cuatro actividades relacionadas con la autenticación está ubicada en la ruta que habíamos indicado antes, **/etc/pam.d**, y tienen el mismo nombre que la aplicación que hace uso de ello. Es decir, si quisieramos ver el fichero de configuración del **sshd**, deberíamos ver el fichero **/etc/pam.d/sshd**, que posee varias líneas como la mostrada debajo:



La idea es básicamente la siguiente: Cuando una aplicación necesita que **PAM** realice una actividad, le pide a la librería que haga ese trabajo. La librería **PAM** llama a cada uno de los módulos en la pila de actividades del fichero de configuración, uno a uno. Cada uno de los módulos hace su tarea, y devuelve un **pass/fail** a la librería.

La librería **PAM** combina estos resultados y devuelve un **pass/fail** global, referente a la invocación de la totalidad de los pasos. Este resultado es el que se devuelve a la aplicación. Para realizar esta combinación de resultados individuales, se hace uso del **control flag** asociado a cada módulo (segundo campo del fichero de configuración), que tiene como posibles valores:

- **requisito** → Si este módulo falla, **PAM** deja de ejecutar lo que quede de la pila y devuelve inmediatamente un fallo a la aplicación. No se ejecutan más módulos.
- **required** → Si este módulo falla, **PAM** devuelve un fallo a la aplicación, pero continúa la invocación al resto de módulos de la pila.
- **sufficient**: Si este módulo tiene éxito, **PAM** devuelve un **pass** a la aplicación, y no se ejecutan más módulos de la pila.
- **optional** → El resultado de la llamada a este módulo es ignorada, la razón por la que se llama a este módulo es para realizar alguna otra operación (por ejemplo, el módulo `pam_keyinit` se usa para crear un nuevo session keyring para el login cuando se invoca al `sshd`)

Los módulos más usados en **PAM**:

- `pam_unix` → (auth,session,password) Autenticaciones tradicionales Unix.
- `pam_limits` → (session) Para poner límites a los recursos del sistema durante una sesión (por defecto se cogen en `/etc/security/limits.conf`). Podemos poner límites al número de procesos, tamaño máximo de fichero, máximo de uso de CPU, etc...
- `pam_rootok` → (auth) Si eres root tiene éxito, si no, falla.
- `pam_cracklib` → (password) Realizar una comprobación de la robustez del password, probándolo contra un diccionario del sistema y una serie de reglas para evitar elecciones malas

- `pam_passwdqc` → (password) También comprueba la robusted del password, y es configurable por nosotros.
- `pam_permit` → (auth,account,session,password) Este módulo siempre dice que **SI** a todo.
- `pam_deny` → (auth,account,session,password) Este módulo siempre dice que **NO** a todo.
- `pam_warn` → (auth,account,session,password) Hace log de un mensaje al `syslogd`, se usa por ejemplo al final de la pila de ejecución para conocer por qué se va a rechazar la conexión (justo antes de rechazarla con un **pam_deny**)
- `pam_motd` → (session) Muestra un mensaje del día (por defecto, el existente en `/etc/motd`), por lo generla después de un login exitoso

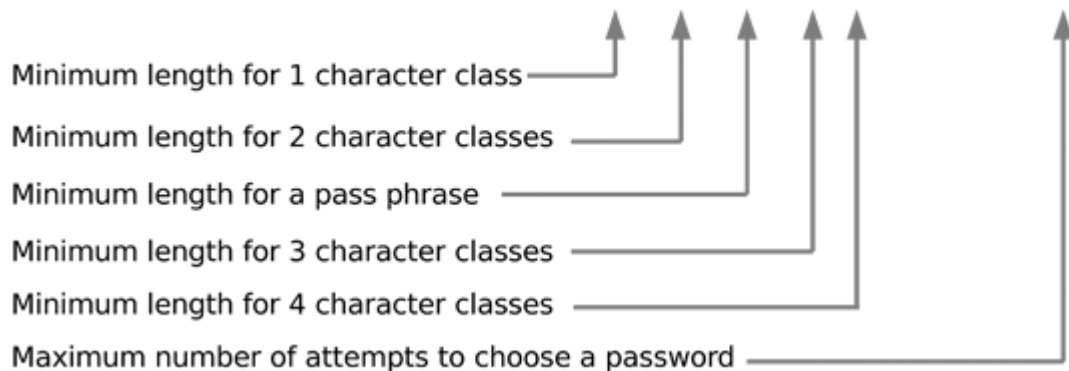
- `pam_securetty` → (auth) Este módulo restringe las conexiones de **root** a los terminales, permitiendo sólo conexiones de los terminales indicados en `/etc/securetty` (tty es terminal device). Este módulo no tiene efectos para usuarios no root
- `pam_wheel` → (auth, account) Usado por programas como **su**, este módulo permite accesos root sólo si el usuario es miembro del grupo llamado **wheel** (el uso de ese grupo atiende a razones históricas)
- `pam_winbind` → (auth) Autentica el usuario contra un dominio de windows usando el demonio winbind (que habla con un controlador de dominio de windows). Así podemos integrar nuestro Linux en una infraestructura Windows y mantener los usuarios
- `pam_nologin` → Evita que los usuarios (que no sean root) puedan conectarse en el sistema si existe fichero en `/etc/nologin`.

Vamos a estudiar algunos ejemplos de uso de **PAM**.

Por ejemplo, vamos a poner una política de passwords distinta. Podemos ir a `/etc/pam.d/passwd` y añadir la siguiente línea:

```
password    requisite    pam_passwdqc.so  min=12,10,10,8,6  retry=3
```

min=12,10,10,8,6 **retry=3**



Podemos permitir a los usuarios hacer **su** sin que solicite su password. Para ello, podemos ir al fichero **/etc/pam.d/su**, comentar todas las líneas referentes a la autenticación y poner una nueva línea de autenticación:

```
auth    sufficient    pam_permit.so
```

Otra coas que podemos hacer, es prohibir el uso del comando **su** a cualquier usuario. Para ello, podemos ir al fichero **/etc/pam.d/su** y dejar como única política la siguiente línea:

```
auth    requisite    pam_deny.so
```

O cambiar la política de autenticación de un proceso para que, en caso de ser root, no se requiera password, y en caso de no serlo nos lo solicite:

auth	sufficient	pam_rootok.so
auth	required	pam_unix.so

92. LDAP y NIS

NIS es un sistema de gestión centralizada de autenticación de usuarios, host, servicios y demás, que sustituye a los ficheros locales `/etc/passwd`, `/etc/groups`, `/etc/hosts`, `/etc/shadow`, `/etc/services`, `/etc/networks`, `/etc/protocols`...

Los clientes **NIS** obtienen a través del servidor **NIS** esta información en su lugar.

El clúster de servidores que hacen uso de un servidor **NIS** es lo que se conoce como **Dominio NIS**, y debe estar sincronizado mediante un protocolo como el **NTP** (Network Time Protocol).

Actualmente, **NIS** está cayendo en desuso, y la autenticación centralizada basada en **LDAP** está sustituyendo a este método de autenticación.

LDAP (del inglés, Lightweight Directory Access Protocol) es un protocolo a nivel de aplicación que permite el acceso a un servidor de directorio ordenado y distribuido para buscar información en un entorno de red (también se considera como una base de datos sobre la que se pueden realizar consultas).

Un **directorio** es un conjunto de objetos con atributos, organizados de una manera lógica y jerárquica.

Un árbol de directorio **LDAP** usa por lo general, como nivel más alto de la jerarquía, un sistema de nombres de dominio (DNS), y conforme se desciende, pueden aparecer entradas que representan países, personas, unidades organizacionales, documentos, grupos de personas...

LDAP - dc=basis,dc=com - OpenLDAP Linux VM - Apache Directory Studio

LDAP Browser

DIT

- Root DSE (2)
- dc=basis,dc=com (4+)
 - cn=admin
 - ou=groups
 - ou=bbjpermissions (3)
 - cn=admin
 - cn=jash
 - cn=jdoe
 - ou=users (2)
 - uid=jash
 - uid=jdoe

Searches

Bookmarks

Connection

LDAP Ser

OpenLDAP Linux VM

dc=basis,dc=com

DN: dc=basis,dc=com

Attribute Description	Value
objectClass	dcObject (auxiliary)
objectClass	organization (structural)
objectClass	top (abstract)
dc	basis
o	BASIS

Modification Logs

Search Logs

```
#!RESULT OK
#!CONNECTION ldap://172.16.120.207:389
#!DATE 2015-03-26T18:37:50.092
dn: cn=jash,ou=bbjpermissions,dc=basis,dc=com
changetype: modify
replace: sn
sn: jash
-
```

Outline

- dc=basis,dc=com
 - objectClass (3)
 - dc (1)
 - o (1)

Progress

No operations to display at this

Para poder configurar de manera centralizada el acceso al sistema mediante **LDAP**, tenemos que configurar por un lado el servidor central, por el otro los clientes (que deben ser capaces de localizar el servidor **LDAP** y autenticarse para poder hacer consultas), y por último, modificar el contenido de **/etc/pam.d/system-auth** para añadir un nuevo módulo **PAM** (que puedes necesitar instalar), el **pam_ldap.so**

Sin entrar en detalles, por lo general una configuración de **LDAP** (usando **openldap**) sobre un **RedHat** sería

- `yum remove sssd` → No es obligatorio, pero puede dar conflicto por cachear credenciales
- `yum install openldap-clients nss-pam-ldapd` → Para permitir acceso vía **LDAP** y disponer del módulo **PAM** adecuado
- Ejecutar **authconfig** (está en `/usr/bin/authconfig`) y elegir **Use LDAP**
- Revisar la configuración de `/etc/openldap/ldap.conf` (asegurarse de que tenemos un certificado válido)
- Revisar `/etc/nslcd.conf` → para confirmar que hacemos uso de `ssl` y que `certfile` apunta a un fichero válido.

93. Práctica Final

1. Práctica final 1
2. Práctica final 2

94. Práctica final 1

Nuestra empresa necesita disponer de una serie de servicios levantados en uno de nuestros servidores. Estos servicios se ejecutan a través de un fichero **jar**, y permiten la conexión de terceros a nuestros sistemas, que además deben tener la máxima prioridad.

El problema es, que en ocasiones puntuales, el servicio puede caerse, por lo que no estará disponible hasta que alguien no lo vuelva a levantar.

A Sigfrido se le ha ocurrido una idea ¿Y si hacemos un script que automáticamente se encargue de todo esto?

Su idea es la siguiente, usar **crontab** para planificar la ejecución de un script de supervisión cada 5 minutos.

Este script, mediante el comando **ps** comprobará si el servicio está o no levantado (guardando trazas en un fichero de log).

En caso de no estar levantado, arranca una nueva instancia del software y cambia si prioridad (con `renice`) a la más prioritaria posible. (de esto, también hay que dejar trazas).

95. Práctica final 2

Haciendo uso de **DIALOG**, crear un programa que permita

- Buscar procesos en ejecución mediante un nombre no completo y mostrar sus datos
- Matar un proceso mediante su pid
- Ejecutar un proceso en segundo plano pasado por parámetro
- Consultar los usuarios conectados actualmente en el sistema
- Borrar los ficheros temporales

96. Fin

Gracias y hasta otra

Programación Avanzada en Shell Scripting

Julio 2017

Javier Gómez Santos

jgomez@pronoide.es