Práctica 1 – Edición y ejecución de un guion o shellscript en MINIX

1. Introducción.

El **intérprete de comandos** o "*shell*" es la interfaz principal entre el usuario y el sistema, permitiéndole a aquél interactuar con los recursos de éste. El usuario introduce sus órdenes, el intérprete las procesa y genera la salida correspondiente.

Por lo tanto, un intérprete de comandos de Unix es tanto una interfaz de ejecución de órdenes y utilidades, como un lenguaje de programación, que admite crear nuevas órdenes – denominadas **guiones** o "*shellscripts*"–, utilizando combinaciones de comandos y estructuras lógicas de control, que cuentan con características similares a las del sistema y que permiten que los usuarios y grupos de la máquina cuenten con un entorno personalizado.

En Unix existen 2 familias principales de intérpretes de comandos: los basados en el intérprete de Bourne (BSH, KSH o BASH) y los basados en el intérprete C (CSH o TCSH).

Esa práctica permitirá comprender, ejecutar y empezar a programar en la *Shell*, haciendo referencia especialmente a POSIX POSIX (acrónimo de Portable Operating System Interface, y X viene de UNIX como seña de identidad de la API) es una norma escrita por la IEEE, que define una interfaz estándar del sistema operativo y el entorno, incluyendo un intérprete de comandos (o "shell")

2. Modos de operación.

2.1. La línea de comandos.

La línea de comandos es el interfaz del usuario con el sistema, que permite personalizar el entorno de operación y ejecutar programas y guiones.

El formato típico de una línea consta de una orden y unos modificadores y parámetros opcionales, aunque puede incluir algunos caracteres especiales, que modifican el comportamiento típico.

```
Comando [Modificador ...][Parámetro ...]
```

Un caso especial es el de las líneas que comienzan por la almohadilla (**#**), que se consideran comentarios. También puede insertarse un comentario en mitad de una línea, a la derecha de una orden.

```
# Esto es un comentario
ls -al    # lista el contenido del directorio actual
```

Pueden agruparse varias órdenes en la misma línea separadas por el punto y coma (**;**), que se ejecutan siempre en secuencia. Asimismo, si un comando es muy largo o engorroso, puede usarse el carácter de escape (****) para continuar escribiéndolo en la línea siguiente.

```
cd /var/log; grep –i error *
find /tmp /var/tmp ! –user root –type f \
     -perm +2 –print
```

2.2. Edición y ejecución de un guion.

Un guion interpretado por SHELL es un archivo de texto normal que consta de una serie de bloques de código formados por líneas de comandos que se ejecutan en secuencia. El usuario debe tener los permisos de modificación (escritura) en el directorio –para crear un nuevo programa– o sobre el propio archivo, para modificar uno existente.

Como un programa binario, el usuario debe tener permiso de ejecución en el archivo del guion, y se ejecuta tecleando su nombre completo junto con sus opciones y parámetros. Asimismo, si el programa se encuentra en un directorio incluido en la variable de entorno **`PATH`**, sólo se necesita teclear el nombre, sin necesidad de especificar el camino.

El proceso completo de edición y ejecución de un guion es el siguiente:

```
vi prueba.sh              # o cualquier otro editor de textos
chmod u+x prueba.sh       # activa el permiso de ejecución
./prueba.sh               # ejecuta el guion
prueba.sh                 # si está en un directorio de $PATH
```

Existe una manera especial para ejecutar un guion, precediéndolo por el signo punto, que se utiliza para exportar todas las variables del programa al entorno de ejecución del usuario. El siguiente ejemplo ilustra el modo de ejecutar

```
apachectl start           # Ejecución normal de un guion.
. miprofile               # Ejecución exportando las variables.
source miprofile          # Equivalente a la línea anterior.
```

Un "*script*" puede –y debe– comenzar con la marca **`#!`** para especificar el camino completo y los parámetros del intérprete de comandos que ejecutará el programa. Esta marca puede usarse para ejecutar cualquier intérprete instalado en la máquina (BASH, BSH, PERL, AWK, etc.).

El siguiente cuadro muestra un pequeño ejemplo de guion.

```
#!/bin/sh
# ejemplo1: informe de la capacidad de la cuenta

echo "Usuario: $USER" echo
"Capacidad de la cuenta:"
du –s $HOME               # suma total del directorio del usuario
```

3. Redirecciones.

Unix hereda 3 archivos especiales del lenguaje de programación C, que representan las funciones de entrada y salida de cada programa. Éstos son:

- **Entrada estándar**: procede del teclado; abre el archivo con descriptor **0** (`stdin`) para lectura.
- **Salida estándar**: se dirige a la pantalla; abre el archivo con descriptor **1** (`stdout`) para escritura.
- **Salida de error**: se dirige a la pantalla; abre el archivo con descriptor **2** (`stderr`) para escritura y control de mensajes de error.

El proceso de **redirección** permite hacer una copia de uno de estos archivos especiales hacia o desde otro archivo normal. También pueden asignarse los descriptores de archivos del 3 al 9 para abrir otros tantos archivos, tanto de entrada como de salida.

El archivo especial **`/dev/null`** se utiliza para descartar alguna redirección e ignorar sus datos.

3.1. Redirección de entrada.

La **redirección de entrada** sirve para abrir para lectura el archivo especificado usando un determinado número descriptor de archivo. Se usa la entrada estándar cuando el valor del descriptor es 0 o éste no se especifica.

El siguiente cuadro muestra el formato genérico de la redirección de entrada.

```
[N]<Archivo
```

La redirección de entrada se usa para indicar un archivo que contiene los datos que serán procesados por el programa, en vez de teclearlos directamente por teclado. Por ejemplo:

```
miproceso.sh < fichdatos
```

Sin embargo, conviene recordar que la mayoría de las utilidades y filtros de Unix soportan los archivos de entrada como parámetro del programa y no es necesario redirigirlos.

3.2. Redirecciones de salida.

De igual forma a los descrito en el apartado anterior, la **redirección de salida** se utiliza para abrir un archivo –asociado a un determinado número de descriptor– para operaciones de escritura.

Se reservan 2 archivos especiales para el control de salida de un programa: la salida normal (con número de descriptor 1) y la salida de error (con el descriptor 2).

En la siguiente tabla se muestran los formatos genéricos para las redirecciones de salida.

Redirección	Descripción
`[N]>` *Archivo*	Abre el archivo de descriptor *N* para escritura. Por defecto se usa la salida estándar (*N*=1). Si el archivo existe, se borra; en caso contrario, se crea.
`[N]>|` *Archivo*	Como en el caso anterior, pero el archivo debe existir previamente.
`[N]>>` *Archivo*	Como en el primer caso, pero se abre el archivo para añadir datos al final, sin borrar su contenido.
`&>` *Archivo*	Escribe las salida normal y de error en el mismo archivo.

El siguiente ejemplo crea un archivo con las salidas generadas para configurar, compilar e instalar una aplicación GNU.

```
configure     > aplic.sal make
>> aplic.sal make install >>
aplic.sal
```

3.3. Combinación de redirecciones.

Pueden combinarse más de una redirección sobre el mismo comando o grupo de comandos, interpretándose siempre de izquierda a derecha.

Ejercicio 3.1: interpretar las siguientes órdenes:

```
ls –al /usr /tmp /noexiste >ls.sal 2>ls.err find
/tmp –print >find.sal 2>/dev/null
```

Otras formas de combinar las redirecciones permiten realizar copias de descriptores de archivos de entrada o de salida. La siguiente tabla muestra los formatos para duplicar descriptores.

Redirección	Descripción
[*N*]**<&***M*	Duplicar descriptor de entrada *M* en *N* (*N*=0, por defecto).
[*N*]**<&-**	Cerrar descriptor de entrada *N*.
[*N*]**<&***M***-**	Mover descriptor de entrada *M* en *N*, cerrando *M* (*N*=0, por defecto).
[*N*]**>&***M*	Duplicar descriptor de salida *M* en *N* (*N*=1, por defecto).
[*N*]**>&-**	Cerrar descriptor de salida *N*.
[*N*]**>&***M***-**	Mover descriptor de salida *M* en *N*, cerrando *M* (*N*=1, por defecto).

Conviene hacer notar que –siguiendo las normas anteriores– las 2 líneas siguientes son equivalentes y ambas sirven para almacenar las salidas normal y de error en el archivo indicado:

```
ls –al /var/* &>ls.txt ls
–al /var/* >ls.txt 2>&1
```

Sin embargo, el siguiente ejemplo muestra 2 comandos que no tienen por qué dar el mismo resultado, ya que las redirecciones se procesan de izquierda a derecha, teniendo en cuenta los posibles duplicados de descriptores hechos en líneas anteriores.

```
ls –al * >ls.txt 2>&1  # Salida normal y de error a "ls.txt". ls
–al * 2>&1 >ls.txt  # Asigna la de error a la normal anterior
# (puede haberse redirigido) y luego                          #
manda la estándar a "ls.txt".
```

3.4. Redirección de entrada/salida.

La **redirección de entrada y salida** abre el archivo especificada para operaciones de lectura y escritura y le asigna el descriptor indicado (0 por defecto). Se utiliza en operaciones para modificación y actualización de datos. El formato genérico es:

```
[N]<>Archivo
```

El siguiente ejemplo muestra una simple operación de actualización de datos en un determinado lugar del archivo [4].

```
echo 1234567890 > fich    # Genera el contenido de "fich" exec
3<> fich             # Abrir "fich" con descriptor 3 en E/S read
-n 4 <&3             # Leer 4 caracteres echo -n , >&3
# Escribir coma decimal exec 3>&-                  # Cerrar
descriptor 3 cat fich                  # ➲ 1234,67890
```

3.5. Documento interno.

La **redirección de documento interno** usa parte del propio programa –hasta encontrar un delimitador de final– como redirección de entrada al comando correspondiente. Suele utilizarse para mostrar o almacenar texto fijo, como por ejemplo un mensaje de ayuda.
El texto del bloque que se utiliza como entrada se trata de forma literal, esto es, no se realizan sustituciones ni expansiones.

El texto interno suele ir tabulado para obtener una lectura más comprensible. El formato **<<** mantiene el formato original, pero en el caso de usar el símbolo **<<-**, el intérprete elimina los caracteres de tabulación antes de redirigir el texto.

La siguiente tabla muestra el formato de la redirección de documento interno.

Redirección	Descripción
<<[**-**] *Delimitador* *Texto* ... *Delimitador*	Se usa el propio *shellscript* como entrada estándar, hasta la línea donde se encuentra el delimitador. Los tabuladores se eliminan de la entrada en el caso de usar la redirección **<<-** y se mantienen con **<<**.

Como ejemplo se muestra un trozo de código y su salida correspondiente, que presentan el texto explicativo para el formato de uso de un programa.

```
echo << FIN
Formato:
config OPCION ...

OPCIONES:
    --cflags
    --ldflags
    --libs
    --version
    --help
FIN
```

```
Formato:
config OPCION ...

OPCIONES:
    --cflags
    --ldflags
    --libs
    --version
    --help
```

3.6. Tuberías.

La **tubería** es una herramienta que permite utilizar la salida normal de un programa como entrada de otro, por lo que suele usarse en el filtrado y depuración de la información, siendo una de las herramientas más potentes de la programación con intérpretes Unix.

Pueden combinarse más de una tubería en la misma línea de órdenes, usando el siguiente formato:

```
Comando1 | Comando2 ...
```

Unix incluye gran variedad de filtros de información. La siguiente tabla recuerda algunos de los más utilizados.

Comando	Descripción
head	Corta las primeras líneas de un archivo.
tail	Extrae las últimas líneas de un archivo.
grep	Muestra las líneas que contienen una determinada cadena de caracteres o cumplen un cierto patrón.
cut	Corta columnas agrupadas por campos o caracteres.
uniq	Muestra o quita las líneas repetidas.
sort	Lista el contenido del archivo ordenado alfabética o numéricamente.
wc	Cuenta líneas, palabras y caracteres de archivos.
find	Busca archivos que cumplan ciertas condiciones y posibilita ejecutar operaciones con los archivos localizados
sed	Edita automáticamente un archivo.
diff	Muestra las diferencias entre 2 archivos, en formato compatible con la orden **sed**.
comm	Compara 2 archivos ordenados.
tr	Sustituye grupos de caracteres uno a uno.
awk	Procesa el archivo de entrada según las reglas de dicho lenguaje.

El siguiente ejemplo muestra una orden compuesta que ordena todos los archivos con extensión "`.txt`", elimina las líneas duplicadas y guarda los datos en el archivo "`resultado.sal`".

```
cat *.txt | sort | uniq >resultado.sal
```

La orden **tee** es un filtro especial que recoge datos de la entrada estándar y lo redirige a la salida normal y a un archivo especificado, tanto en operaciones de escritura como de añadidura. Esta es una orden muy útil que suele usarse en procesos largos para observar y registrar la evolución de los resultados.

El siguiente ejemplo muestra y registra el proceso de compilación e instalación de una aplicación GNU.

```
configure   2>&1 | tee aplic.sal make
2>&1 | tee –a aplic.sal make instal
2>&1 | tee –a aplic.sal
```

Ejercicio 3.2: interpretar la siguiente orden:

```
ls | tee salida | sort -r
```

4. Variables.

Al contrario que en otros lenguajes de programación, SHELL no hace distinción en los tipos de datos de las variables; son esencialmente cadenas de caracteres, aunque –según el contexto– también pueden usarse con operadores de números enteros y condicionales. Esta filosofía de trabajo permite una mayor flexibilidad en la programación de guiones, pero también puede provocar errores difíciles de depurar [4].

Una variable SHELL se define o actualiza mediante operaciones de asignación, mientras que se hace referencia a su valor utilizando el símbolo del dólar delante de su nombre.

Suele usarse la convención de definir las variables en mayúsculas para distinguirlas fácilmente de los comandos y funciones, ya que en Unix las mayúsculas y minúsculas se consideran caracteres distintos.

```
VAR1="Esto es una prueba     # asignación de una variable
VAR2=35                      # asignar valor numérico
echo $VAR1                   # ➲ Esto es una prueba
echo "VAR2=$VAR2"            # ➲ VAR2=35
```

4.1. Tipos de variables.

Las variables del intérprete SHELL pueden considerarse desde los siguientes puntos de vista:

- Las **variables locales** son definidas por el usuario y se utilizan únicamente dentro de un bloque de código, de una función determinada o de un guion.
- Las **variables de entorno** son las que afectan al comportamiento del intérprete y al de la interfaz del usuario.
- Los **parámetros de posición** son los recibidos en la ejecución de cualquier programa o función, y hacen referencia a su orden ocupado en la línea de comandos.
- Las **variables especiales** son aquellas que tienen una sintaxis especial y que hacen referencia a valores internos del proceso. Los parámetros de posición pueden incluirse en esta categoría.

4.1.1. Variables locales.

Las variables locales son definidas para operar en un ámbito reducido de trabajo, ya sea en un programa, en una función o en un bloque de código. Fuera de dicho ámbito de operación, la variable no tiene un valor preciso.

Una variable tiene un nombre único en su entorno de operación, sin embargo, pueden –aunque no es nada recomendable– usarse variables con el mismo nombre en distintos bloques de código.

El siguiente ejemplo muestra los problemas de comprensión y depuración de código que pueden desatarse en caso de usar variables con el mismo nombre. En la primera fila se presentan 2 programas que usan la misma variable y en la segunda, la ejecución de los programas (nótese que el signo **>** es el punto indicativo del interfaz de la "*shell*" y que lo tecleado por el usuario se representa en letra negrita).

`#!/bin/sh`
`# prog1 – variables prueba 1`
`VAR1=prueba`
`echo $VAR1`
	`#!/bin/sh`
	`# prog2 – variables prueba 2`
	`VAR1="otra prueba"`
	`echo $VAR1`

`> echo $VAR1`
`> prog1`
`prueba >`
`prog2 otra`
`prueba >`
`prog1`
`prueba`	

Por lo tanto, para asignar valores a una variable se utiliza simplemente su nombre, pero para hacer referencia a su valor hay que utilizar el símbolo dólar (**$**). El siguiente ejemplo muestra los modos de referirse a una variable.

```
ERR=2                          # Asigna 2 a la variable ERR.
echo ERR                       # O ERR    (cadena "ERR").
echo $ERR                      # O 2      (valor de ERR).
echo ${ERR}                    # O 2      (es equivalente).
echo "Error ${ERR}: salir"     # O Error 2: salir
```

El formato **${***Variable***}** se utiliza en cadenas de caracteres donde se puede prestar a confusión o en procesos de sustitución de valores.

4.1.2. Variables de entorno.

Al igual que cualquier otro proceso Unix, la "*shell*" mantiene un conjunto de variables que informan sobre su propio contexto de operación. El usuario –o un guion– puede actualizar y añadir variables exportando sus valores al entorno del intérprete (comando **`export`**), lo que afectará también a todos los procesos hijos generados por ella. El administrador puede definir variables de entorno estáticas para los usuarios del sistema (como, por ejemplo, en el caso de la variable **`IFS`**).

La siguiente tabla presenta las principales variables de entorno.

Variable de entorno	Descripción	Valor por omisión
`SHELL`	Camino del programa intérprete de comandos.	La propia *shell*.

HOME	Directorio personal de la cuenta.	Lo define **root**.
LOGNAME	Nombre de usuario que ejecuta la *shell.*	Activado por **login**
PATH	Camino de búsqueda de comandos.	Según el sistema
PAGER	Herramienta para paginar las terminales	more
EDITOR	Editor usado por defecto.	vi
TERM	Tipo de terminal.	minix
PS1 **PS2**	Prompt indicativos primario. Prompt indicativo secundario.	# **>**
TZ	Configuración del “Time Zone” o zona horaria	Según el sistema

Debe hacerse una mención especial a la variable **PATH**, que se encarga de guardar la lista de directorios con archivos ejecutables. Si no se especifica el camino exacto de un programa, el sistema busca en los directorios especificados por **PATH**, siguiendo el orden de izquierda a derecha. El carácter separador de directorios es dos puntos.

El administrador del sistema debe establecer los caminos por defecto para todos los usuarios del sistema y cada uno de éstos puede personalizar su propio entorno, añadiendo sus propios caminos de búsqueda (si no usa un intérprete restringido).

Ejercicio 4.1: Imprimir el contenido de todas las variables de entorno

Recomendaciones de seguridad:
Siempre deben indicarse caminos absolutos en la definición de la variable **PATH** y, sobre todo, nunca incluir el directorio actual (.) ni el directorio padre (..).
Declarar la variable **IFS** de sólo lectura, para evitar intrusiones del tipo “caballos de Troya”.

5. Expr – Evaluador de expresiones de SHELL

El comando **expr** es usado para evaluar expresiones

El intérprete SHELL de MINIX no cuenta con un mecanismo para realizar operaciones aritméticas simples. A partir de UNIX V, se agregó el programa expr para realizar la evaluación de expresiones. Las distintas expresiones soportadas por el intérprete pueden englobarse en las siguientes categorías:

- **Aritméticas**
- **Lógicas**
- **Relacionales**
- **Cadenas**

Las expresiones complejas cuentan con varios parámetros y operadores, se evalúan de izquierda a derecha. Sin embargo, si una operación está encerrada entre paréntesis se considera de mayor prioridad y se ejecuta antes.

A modo de resumen, la siguiente tabla presenta los operadores utilizados en los distintos tipos de expresiones SHELL.

Operador	Tipo	Significado
+	Aritmético	Suma
-	Aritmético	Resta
*	Aritmético	Multiplicación
/	Aritmético	División
%	Aritmético	Módulo
=	Relacional	Igual que
>	Relacional	Mayor que
>=	Relacional	Mayor o igual a
<	Relacional	Menor que
<=	Relacional	Menor or igual a
!=	Relacional	No igual que
|	Booleano	O
&	Booleano	y
:	Cadena	Coincidencia o sustituto

La forma general de usar las expresiones es:

```
término operador término
```

Primero, debe haber espacios entre los operadores y las expresiones, el siguiente ejemplo es incorrecto:

```
expr 2+2
```

La forma correcta es:

```
expr 2 + 2
```

Que imprime “4” en una salida estándar.

5.1. Expresiones aritméticas.

Las expresiones aritméticas representan operaciones números enteros o binarios (booleanos) evaluadas mediante el comando expr. SHELL no efectúa instrucciones algebraicas con números reales ni con complejos.

La valoración de expresiones aritméticas enteras sigue las reglas:

- Se realiza con números enteros de longitud fija sin comprobación de desbordamiento, esto es, ignorando los valores que sobrepasen el máximo permitido.
- La división por 0 genera un error que puede ser procesado.
- La prioridad y asociatividad de los operadores sigue las reglas del lenguaje C.

La siguiente tabla describe las operaciones aritméticas enteras y binarias agrupadas en orden de prioridad.

Operación	Descripción	Comentarios
E1 ***** *E2* *E1* **/** *E2* *E1* **%** *E2*	Multiplicación. División. Resto.	Operaciones de multiplicación y división entre números enteros.
E1 **+** *E2* *E1* **-** *E2*	Suma. Resta.	Suma y resta de enteros.
E1 **<** *E2* *E1* **<=** *E2 E1* **>** *E2* *E1* **>=** *E2*	Comparaciones (menor, menor o igual, mayor, mayor o igual).	
E1 **=** *E2* *E1* **!=** *E2*	Igualdad. Desigualdad.	
E1 **&** *E2*	Operación binaria Y.	
E1 **|** *E2*	Operación binaria O.	
E1 **=** *E2*	Asignación normal	Asigna el valor de E2 a E1

El cuadro que se muestra a continuación ilustra el uso de las expresiones aritméticas.

```
expr 2 "*" 10
a=5                                          # Asignación a=5
B=$(expr $a + 3 \* 9)
echo "a=$a, b=$b"                           #   a=5, b=32
c=$(expr $b / \( $a + 3 \))                  # c=b/(a+3)=4
a=$(expr $a + $c)                            #=9
c=$(expr $c - 1)                             #=3
echo "a=$a, c=$c"                                  #   a=9, c=3
                                        # CTE=b/c, RESTO=resto(b/c)
CTE=$ (expr $b / $c);RESTO=$(expr $b % $c)
```

```
echo "Cociente=$CTE, Resto=$RESTO"          #   Cociente=10, Resto=2
```

Ejercicio 5.1: explicar la siguiente expresión.

Escribir las siguientes expresiones en el shellscript

a=10

b= 2 * 10

c= a + b / 2

c = (2+3) * 10

div = c / 2

res = b % c

Imprime el resultado de todas las variables

5.2. Expresiones condicionales.

Las **expresiones condicionales** son evaluadas por los comandos internos del tipo **`test`**, dando como resultado un valor de cierto o de falso. Suelen emplearse en operaciones condicionales y ciclos, aunque también pueden ser empleadas en órdenes compuestas.

Existen varios tipos de expresiones condicionales según el tipo de parámetros utilizado o su modo de operación:

- **Expresiones con archivos**, que comparan la existencia , el tipo, los permisos o la fecha de archivos o directorios.
- **Expresiones comparativas numéricas**, que evalúan la relación de orden numérico entre los parámetros.
- **Expresiones comparativas de cadenas**, que establecen la relación de orden alfabético entre los parámetros.

Todas las expresiones condicionales pueden usar el modificador de negación (**`!`** *`Expr`*) para indicar la operación inversa. Asimismo, pueden combinarse varias de ellas en una expresión compleja usando los operadores lógicos Y (*`Expr1`* **`&&`** *`Expr2`*) y O (*`Expr1`* **`||`** *`Expr2`*).

5.2.1. Expresiones de archivos.

Son expresiones condicionales que devuelven el valor de cierto si se cumple la condición especificada; en caso contrario da un valor de falso. Hay una gran variedad de expresiones relacionadas con archivos y pueden agruparse en operaciones de tipos, de permisos y de comparación de fechas.

Conviene recordar que "todo en Unix es un archivo", por eso hay bastantes operadores de tipos de archivos. La siguiente tabla lista los formatos de estas expresiones.

Formato	Condición (cierto si...)
-e *Fich*	El archivo (de cualquier tipo) existe
-s *Fich*	El archivo no está vacío.
-f *Fich*	Es un archivo normal.
-d *Fich*	Es un directorio.
-b *Fich*	Es un dispositivo de bloques.
-c *Fich*	Es un dispositivo de caracteres.
-p *Fich*	Es una tubería.
-h *Fich* **-L** *Fich*	Es un enlace simbólico.
-S *Fich*	Es una "*socket*" de comunicaciones.
-t *Desc*	El descriptor está asociado a una terminal.
F1 **-ef** *F2*	Los 2 archivos son enlaces hacia el mismo archivo.

Las condiciones sobre permisos establecen si el usuario que realiza la comprobación puede ejecutar o no la operación deseada sobre un determinado archivo. La tabla describe estas condiciones.

Formato	Condición (cierto si...)
-r *Fich*	Tiene permiso de lectura.
-w *Fich*	Tiene permiso de escritura (modificación).
-x *Fich*	Tiene permiso de ejecución/acceso.
-u *Fich*	Tiene el permiso SUID.
-g *Fich*	Tiene el permiso SGID.
-t *Fich*	Tiene permiso de directorio compartido o archivo en caché.
-O *Fich*	Es el propietario del archivo.

-G *Fich*	El usuario pertenece al grupo con el GID del archivo.

Las operaciones sobre fechas –descritas en la siguiente tabla– establecen comparaciones entre las correspondientes a 2 archivos.

Formato	Condición (cierto si...)
-N *Fich*	El archivo ha sido modificado desde al última lectura.
F1 **-nt** *F2*	El archivo F1 es más nuevo que el F2.
F1 **-ot** *F2*	El archivo F1 es más antiguo que el F2.

Veamos algunos ejemplos extraídos del archivo de configuración `/etc/rc.d/rc.sysinit`.

```
# Si /proa/mdstat y /etc/raidtab son archivos; entonces
... if [ -f /proc/mdstat -a -f /etc/raidtab ]; then ...
# Si el camino representado por el contenido de la variable
#    $afile es un directorio; entonces
... if [ -d "$afile" ]; then ...
```

Ejercicio 5.2: Elaborar un script que use expresiones condicionales para validar 3 archivos. El primer archivo debe tener permisos de lectura, el segundo debe existir y tener permisos de ejecución. Mostrar quién es el propietario del tercer archivo. Mostrar cual de los tres es el archivo más antiguo.

5.3.2. Expresiones comparativas numéricas.

Aunque los operadores de comparación para números ya se han comentado en el apartado anterior, la siguiente tabla describe los formatos para este tipo de expresiones.

Formato	Condición (cierto si...)
N1 **-eq** *N2* *N1* **-ne** *N2* *N1* **-lt** *N2* *N1* **-gt** *N2*	Se cumple la condición de comparación numérica (respectivamente igual, distinto, menor y mayor).

Continuando con el ejemplo, se comentan algunas líneas del archivo de configuración `/etc/rc.d/rc.sysinit`.

```
# Si la variable RESULT es > 0 y
#    /sbin/raidstart es ejecutable; entonces ...
if [ $RESULT -gt 0 -a -x /sbin/raidstart ]; then
...
# Si el código de la ejecución del 1er comando es 0 y
#    el del 2° es distinto de 0; entonces ...
if grep -q /initrd /proc/mounts && \
   ! grep -q /initrd/loopfs /proc/mounts ; then
...
# Si la expresión de que existe el archivo /fastboot es cierta o
#    el código de salida del comando es correcto; entonces ...
if [ -f /fastboot ] || \
   grep -iq "fastboot" /proc/cmdline 2>/dev/null ; then ...
```

5.3.3. Expresiones comparativas de cadenas.

También pueden realizarse comparaciones entre cadenas de caracteres. La tabla indica el formato de las expresiones.

Formato	Condición (cierto si...)
Cad1 **=** *Cad2* *Cad1* **!=** *Cad2*	Se cumple la condición de comparación de cadenas (respectivamente igual y distinto).
[**-n**] *Cad*	La cadena no está vacío (su longitud no es 0).
-z *Cad*	La longitud de la cadena es 0.

Como en los párrafos previos, se revisa parte del código del archivo `/etc/rc.d/rc.sysinit`.

Ejercicio 5.3: Completa el siguiente código para verificar el contenido todas las variables de entorno de MINIX, tip usa env para conocer esas variables.

```
# Si LOGNAME es una variable vacía o #    tiene el
valor "(none)"; entonces ... if [ -z "$HOSTNAME" -o
"$HOSTNAME" = "(none)" ]; then .
```

7. Programación estructurada.

Las estructuras de programación se utilizan para generar un código más legible y fiable. Son válidas para englobar bloques de código que cumplen un cometido común, para realizar comparaciones, selecciones, ciclos repetitivos y llamadas a subprogramas.

7.1. Listas de comandos.

Los comandos SHELL pueden agruparse en bloques de código que mantienen un mismo ámbito de ejecución. La siguiente tabla describe brevemente los aspectos fundamentales de cada lista de órdenes.

Lista de órdenes	Descripción
Comando **&**	Ejecuta el comando en 2º plano. El proceso tendrá menor prioridad y no debe ser interactivo..
Man1 **|** *Man2*	Tubería. Redirige la salida de la primera orden a la entrada de la segundo. Cada comando es un proceso separado.
Man1; *Man2*	Ejecuta varios comandos en la misma línea de código.
Man1 **&&** *Man2*	Ejecuta la 2ª orden si la 1ª lo hace con éxito (su estado de salida es 0).
Man1 **||** *Man2*	Ejecuta la 2ª orden si falla la ejecución de la anterior (su código de salida no es 0).
(*Lista***)**	Ejecuta la lista de órdenes en un subproceso con un entorno común.
{ *Lista*; **}**	Bloque de código ejecutado en el propio intérprete.
Línea1 \ *Línea2*	Posibilita escribir listas de órdenes en más de una línea de pantalla. Se utiliza para ejecutar comandos largos.

7.2. Estructuras condicionales y selectivas.

Ambas estructuras de programación se utilizan para escoger un bloque de código que debe ser ejecutado tras evaluar una determinada condición. En la estructura condicional se opta entre 2 posibilidades, mientras que en la selectiva pueden existir un número variable de opciones.

7.2.1. Estructuras condicionales.

La **estructura condicional** sirve para comprobar si se ejecuta un bloque de código cuando se cumple una cierta condición. Pueden anidarse varias estructuras dentro del mismo bloques de código, pero siempre existe una única palabra **`fi`** para cada bloque condicional.

La tabla muestra cómo se representan ambas estructuras en SHELL.

Estructura de programación	Descripción
if *Expresión*; **then** *Bloque*; fi	**Estructura condicional simple**: se ejecuta la lista de órdenes si se cumple la expresión. En caso contrario, este código se ignora.

if *Expresión1;* **then Bloque1;** [**elif** *Expresión2;* **then** *Bloque2;* ...] [**else** *BloqueN;*] **fi**	**Estructura condicional compleja**: el formato completo condicional permite anidar varias órdenes, además de poder ejecutar distintos bloques de código, tanto si la condición de la expresión es cierta, como si es falsa.

Aunque el formato de codificación permite incluir toda la estructura en una línea, cuando ésta es compleja se debe escribir en varias, para mejorar la comprensión del programa. En caso de teclear la orden compleja en una sola línea debe tenerse en cuenta que el carácter separador (*;*) debe colocarse antes de las palabras reservadas: **then**, **else**, **elif** y **fi**.

Hay que resaltar la versatilidad para teclear el código de la estructura condicional, ya que la palabra reservada **then** puede ir en la misma línea que la palabra **if**, en la línea siguiente sola o conjuntamente con la primera orden del bloque de código, lo que puede aplicarse también a la palabra **else**).

Puede utilizarse cualquier expresión condicional para evaluar la situación, incluyendo el código de salida de un comando o una condición evaluada por la orden interna **test**. Este último caso se expresa colocando la condición entre corchetes (formato: **[** *Condición* **]**).

Véanse algunos sencillos ejemplos de la estructura condicional simple extraídos del "*script*" `/etc/rc.d/rc.sysinit`. Nótese la diferencia en las condiciones sobre la salida normal de una orden –expresada mediante una sustitución de comandos– y aquellas referidas al estado de ejecución de un comando (si la orden se ha ejecutado correctamente o si se ha producido un error).

```
# La condición más simple escrita en una línea:
# si RESULT>0; entonces rc=1 if [
$RESULT -gt 0 ]; then rc=1; fi
#
# La condición doble:
# si la variable HOSTNAME es nula o vale "(none)"; entonces ...
if [ -z "$HOSTNAME" -o "$HOSTNAME" = "(none)" ]; then
    HOSTNAME=localhost
fi
#
# Combinación de los 2 tipos de condiciones:
# si existe el archivo /fastboot o la cadena "fastboot" está
#    en el archivo /proc/cmdline; entonces ... if [ -f
/fastboot ] || grep -iq "fastboot" /proc/cmdline \
                                2>/dev/null; then
      fastboot=yes
fi
```

Ejercicio 7.2: Implementa un script que valide los parámetros del shellscript. Si tiene argumentos que imprima el número de argumentos y su contenido, si no tiene argumentos entonces imprime "Sin argumentos"

7.2.2. Estructura selectiva.

La **estructura selectiva** evalúa la condición de control y, dependiendo del resultado, ejecuta un bloque de código determinado. La siguiente tabla muestra el formato genérico de esta estructura.

Estructura de programación	Descripción
case *Variable* **in** [**(**]*Patrón1***)** *Bloque1* ;; ... **esac**	**Estructura selectiva múltiple**: si la variable cumple un determinado patrón, se ejecuta el bloque de código correspondiente. Cada bloque de código acaba con "**;;**". La comprobación de patrones se realiza en secuencia.

Las posibles opciones soportadas por la estructura selectiva múltiple se expresan mediante patrones, donde puede aparecer caracteres comodines, evaluándose como una expansión de archivos, por lo tanto el patrón para representar la opción por defecto es el asterisco (*****). Dentro de una misma opción pueden aparecer varios patrones separados por la barra vertical (**|**), como en una expresión lógica O.

Si la expresión que se comprueba cumple varios patrones de la lista, sólo se ejecuta el bloque de código correspondiente al primero de ellos, ya que la evaluación de la estructura se realiza en secuencia.

Obsérvense el siguientes ejemplos:

```
# Según el valor de la variable UTC:
# - si es "yes" o "true", ...
# - si es "no" o "false", ...
case "$UTC" in
   yes|true)   CLOCKFLAGS="$CLOCKFLAGS --utc";
CLOCKDEF="$CLOCKDEF (utc)";
   ;;
   no|false)   CLOCKFLAGS="$CLOCKFLAGS --localtime";
               CLOCKDEF="$CLOCKDEF (localtime)";
   ;;
esac
```

Ejercicio 7.3: Explica la diferencia entre estos dos códigos.

```
echo answer yes or noread wordcase $word inyes | YES )echo you
answered yes;;no | NO )echo you answered no;;esac
```

```
echo answer yes or noread wordcase $word
in[Yy]* )echo you answered yes;;[Nn]* )echo you
answered no;;* )echo you did not say yes or
no;;esac
```

7.3. Ciclos.

Los ciclos son estructuras reiterativas que se ejecutan repetitivamente, para no tener que teclear varias veces un mismo bloque de código. Un ciclo debe tener siempre una condición de salida para evitar errores provocados por ciclos infinitos.

La siguiente tabla describe las 2 órdenes especiales que pueden utilizarse para romper el modo de operación típico de un ciclo.

Orden	Descripción
break	**Ruptura inmediata de un ciclo** (debe evitarse en programación estructurada para impedir errores de lectura del código).
continue	**Salto a la condición del ciclo** (debe evitarse en programación estructurada para impedir errores de lectura del código).

Los siguientes puntos describen los distintos ciclos que pueden usarse tanto en un guion como en la línea de comandos de SHELL.

7.3.1. Ciclos genéricos.

Los ciclos genéricos de tipos "para cada" ejecutan el bloque de código para cada valor asignado a la variable usada como índice del ciclo o a su expresión de control. Cada iteración debe realizar una serie de operaciones donde dicho índice varíe, hasta la llegar a la condición de salida.

El tipo de ciclo **for** más utilizado es aquél que realiza una iteración por cada palabra (o cadena) de una lista. Si se omite dicha lista de valores, se realiza una iteración por cada parámetro posicional.

Por otra parte, SHELL soporta otro tipo de ciclo iterativo genérico similar al usado en el lenguaje de programación C, usando expresiones aritméticas. El modo de operación es el siguiente:

- Se evalúa la primera expresión aritmética antes de ejecutar el ciclo para dar un valor inicial al índice.
- Se realiza una comprobación de la segunda expresión aritmética, si ésta es falsa se ejecutan las iteraciones del ciclo. Siempre debe existir una condición de salida para evitar que el ciclo sea infinito.

- como última instrucción del bloque se ejecuta la tercera expresión aritmética –que debe modificar el valor del índice– y se vuelve al paso anterior.

La siguiente tabla describe los formatos de los ciclos iterativos genéricos (de tipo "para cada") interpretados por SHELL.

Ciclo	Descripción
for *Var* [**in** *Lista*]; **do** *Bloque* **done**	**Ciclo iterativo**: se ejecuta el bloque de comandos del ciclo sustituyendo la variable de evaluación por cada una de las palabras incluidas en la lista.

Véanse algunos ejemplos:

```
numeros="1 2 3 4 5 6 7 8 9 10"
 for i in $numeros
 do
     echo $i
done
```

Ejercicio 7.4: Explica qué hace este ciclo :

```
#!/bin/sh# lee el archivo /etc/password# usa $(cat
/etc/passwd) pero los espacios son tratados como líneas
nuevas, entonces se cambian los espacios por _for i in
$(tr ' ' '_' </etc/passwd)doset $(echo $i | tr ':' '
')echo user: $1, UID: $3, Home Directory: $6done
```

7.3.2. Ciclos condicionales "mientras" y "hasta".

Los ciclos condicionales evalúan la expresión en cada iteración del ciclo y dependiendo del resultado se vuelve a realizar otra iteración o se sale a la instrucción siguiente.

La siguiente tabla describe los formatos para los 2 tipos de ciclos condicionales soportados por el intérprete SHELL.

Ciclo	Descripción
while *Expresión;* **do** *Bloque* **done**	**Ciclo iterativo "mientras"**: se ejecuta el bloque de órdenes mientras que la condición sea cierta. La expresión de evaluación debe ser modificada en algún momento del ciclo para poder salir.
until *Expresión;* **do** *Bloque* **done**	**Ciclo iterativo "hasta"**: se ejecuta el bloque de código hasta que la condición sea cierta. La expresión de evaluación debe ser modificada en algún momento del ciclo para poder salir.

Ambos ciclos realizan comparaciones inversas y pueden usarse indistintamente, aunque se recomienda usar aquél que necesite una condición más sencilla o legible, intentando no crear expresiones negativas. Véase el siguiente ejemplo:

```
# Mientras haya parámetros que procesar, ...
while [ $# != 0 ] ; do          processdir
"$1"          shift done
```

Ejercicio 7.5: Explica qué hace este script

```
for number in 1 2 3 4 5 6
 do
        for letter in a b c d e f g
        do
                case $number in
                        3) break
                esac
                echo $number $letter
        done
 done
```

7.3.3. Ciclo de selección interactiva.

La estructura **`select`** no es propiamente dicho un ciclo de programación estructurada, ya que se utiliza para mostrar un menú de selección de opciones y ejecutar el bloque de código correspondiente a la selección escogida. En caso de omitir la lista de palabras, el sistema presenta los parámetros posicionales del programa o función. Este tipo de ciclos no suele utilizarse.

La siguiente tabla describe el formato del ciclo interactivo.

Ciclo	Descripción
`select` *Var* [**`in`** *Lista*]**`;`** **`do`** *Bloque1* **`...`** **`done`**	**Ciclo de selección interactiva**: se presenta un menú de selección y se ejecuta el bloque de código correspondiente a la opción elegida. El ciclo se termina cuando se ejecuta una orden **`break`**. La variable **`PS3`** se usa como punto indicativo.

8. Funciones.

Una función en SHELL es una porción de código declarada al principio del programa, que puede recoger parámetro de entrada y que puede ser llamada desde cualquier punto del programa principal o desde otra función, tantas veces como sea necesario.

El uso de funciones permite crear un código más comprensible y que puede ser depurado más fácilmente, ya que evita posibles errores tipográficos y repeticiones innecesarias.

Los parámetros recibidos por la función se tratan dentro de ella del mismo modo que los del programa principal, o sea los parámetros posicionales de la función se corresponden con las variables internas **`$0`**, **`$1`**, etc.

El siguiente cuadro muestra los formatos de declaración y de llamada de una función..

Declaración	LLamada
[**function**] *NombreFunción* **()** **{** *Bloque* ... [**return** [*Valor*]] ... **}**	*NombreFunción* [*Parámetro1 ...*]

La función ejecuta el bloque de código encerrado entre sus llaves y –al igual que un programa– devuelve un valor numérico. En cualquier punto del código de la función, y normalmente al final, puede usarse la cláusula **`return`** para terminar la ejecución y opcionalmente indicar un código de salida.

Las variables declaradas con la cláusula **`local`** tienen un ámbito de operación interno a la función. El resto de variables pueden utilizarse en cualquier punto de todo el programa. Esta característica permite crear funciones recursivas sin que los valores de las variables de una llamada interfieran en los de las demás.

En el ejemplo del siguiente cuadro se define una función de nombre `salida`, que recibe 3 parámetros. El principio del código es la definición de la función (la palabra **`function`** es opcional) y ésta no se ejecuta hasta que no se llama desde el programa principal. Asimismo, la variable `TMPGREP` se declara en el programa principal y se utiliza en la función manteniendo su valor correcto.

```
#/bin/sh
# comprus - comprueba la existencia de usuarios en listas y en
#           el archivo de claves (normal y NIS).
#           Uso: comprus ? | cadena
#                    ?: ayuda.

# Rutina de impresión.
# Parámetros:
#       1 - texto de cabecera.
#       2 - cadena a buscar.
#       3 - archivo de búsqueda.
salida ()         {
        if egrep "$2" $3 >$TMPGREP 2>/dev/null; then
echo "       $1:"            cat $TMPGREP         fi
        }

## PROGRAMA PRINCIPAL ##

PROG=$(basename $0)
TMPGREP=/tmp/grep$$
DIRLISTAS=/home/cdc/listas

if [ "x$*" = "x?" ] then
echo "
Uso:       $PROG ? | cadena
Propósito: $PROG: Búsqueda de usuarios.
```

```
                cadena: expresión regular a buscar.
"     exit 0 fi
if [ $# -ne 1 ]; then    echo "$PROG:
Parámetro incorrecto.
Uso: $PROG) ? | cadena
?: ayuda" >&2     exit 1 fi
echo
for i in $DIRLISTAS/*.lista; do
    salida "$1" "$(basename $i | sed 's/.lista//')" "$i" done
salida "$1" "passwd" "/etc/passwd" [ -e "$TMPGREP" ] && rm -f
$TMPGREP
```

Ejercicio 7.6: desarrollar en SHELL una función que reciba un único parámetro, una cadena de caracteres, y muestre el último carácter de dicha cadena. Debe tenerse en cuenta que el primer carácter de una cadena tiene como índice el valor 0.

8. Características especiales

En última instancia se van a describir algunas de las características especiales que el intérprete SHELL añade a su lenguaje para mejorar la funcionalidad de su propia interfaz y de los programas que interpreta.

Seguidamente se enumeran algunas de las características especiales de SHELL y en el resto de puntos de este capítulo se tratan en mayor profundidad las que resultan más interesantes:

- Posibilidad de llamar al intérprete SHELL con una serie de opciones que modifican su comportamiento normal.
- Comandos para creación de programas interactivos.
- Control de trabajos y gestión de señales.
- Manipulación y personalización del punto indicativo.
- Soporte de alias de comandos.
- Gestión del histórico de órdenes ejecutadas.
- Edición de la línea de comandos.
- Manipulación de la pila de últimos directorios visitados.
- Intérprete de uso restringido, con características limitadas.
- Posibilidad de trabajar en modo compatible con la norma POSIX 1003.2 [2].

8.1. Programas interactivos.

SHELL proporciona un nivel básico para programar "*shellscripts*" interactivos, soportando instrucciones para solicitar y mostrar información al usuario.

Las órdenes internas para dialogar con el usuario se describen en la siguiente tabla.

Comando	Descripción

read [**-p** *"Cadena"*] [*Var1 ...*]	**Asigna la entrada a variables**: lee de la entrada estándar y asigna los valoras a las variables indicadas en la orden. Puede mostrarse un mensaje antes de solicitar los datos. Si no se especifica ninguna variable, **REPLY** contiene la línea de entrada.
echo [**-n**] *Cadena*	**Muestra un mensaje**: manda el valor de la cadena a la salida estándar; con la opción `-n` no se hace un salto de línea.
printf *Formato Parám1 ...*	**Muestra un mensaje formateado**: equivalente a la función `printf` del lenguaje C, manda un mensaje formateado a la salida normal, permitiendo mostrar cadena y números con una longitud determinada.

Véanse los siguientes ejemplos.

```
# Las siguientes instrucciones son equivalentes y muestran
#     un mensaje y piden un
valor echo -n "Dime tu nombre: "
read NOMBRE
#
read -p "Dime tu nombre: " NOMBRE
...
# Muestra los usuarios y nombres completos
#   (la modificación de la variable IFS sólo afecta al
ciclo) while IFS=: read usuario clave uid gid nombre ignorar
do
  printf "%8s (%s)\n" $usuario $nombre
done </etc/passwd
```

Ejercicio8.1 Como ejercicio se propone explicar paso a paso el funcionamiento del ciclo anterior y los valores que se van asignando a las distintas variables en cada iteración.

8.2. Control de trabajos.

Las órdenes para el control de trabajos permiten manipular los procesos ejecutados por el usuario o por un guion SHELL. El usuario puede manejar varios procesos y subprocesos simultáneamente, ya que el sistema asocia un número identificador único a cada uno de ellos.

Los procesos asociados a una tubería tienen identificadores (PID) propios, pero forman parte del mismo trabajo independiente.

La siguiente tabla describe las instrucciones asociadas con el tratamiento de trabajos.

Comando	Descripción
Comando **&**	**Ejecución en 2º plano**: lanza el proceso de forma desatendida, con menor prioridad y con la posibilidad de continuar su ejecución tras la desconexión del usuario.

bg %*N°Trabajo*	**Retorna a ejecutar en 2º plano**: continúa la ejecución desatendida de un procesos suspendido.
fg %*N°Trabajo*	**Retorna a ejecutar en 1er plano**: vuelve a ejecutar el proceso asociado al trabajo indicado de forma interactiva.
jobs	**Muestro los trabajos en ejecución**, indicando el nº de trabajo y los PID de sus procesos.
kill *Señal PID1*|%*Trab1 ...*	**Manda una señal a procesos o trabajos**, para indicar una excepción o un error. Puede especificarse tanto el nº de señal como su nombre.
suspend	**Para la ejecución del proceso**, hasta que se recibe una señal de continuación.
trap [*Comando*] [*Señal1 ...*]	**Captura de señal**: cuando se produce una determinada señal (interrupción) en el proceso, se ejecuta el comando asociado. Las señales especiales EXIT y ERR se capturan respectivamente al finalizar el "*script*" y cuando se produce un error en una orden simple.
wait [*PID1*|%*Trab1*]	**Espera hasta que termine un proceso o trabajo**: detiene la ejecución del proceso hasta que el proceso hijo indicado hay finalizado su ejecución.

Revisar los siguientes ejemplos.

```
# Se elimina el archivo temporal si en el programa aparecen las
3    señales 0 (fin), 1, 2, 3 y 15.
trap 'rm -f ${FICHTEMP} ; exit' 0 1 2 3 15
...
# Ordena de forma independiente las listas de archivos de
#   usuarios, espera a finalizar ambos procesos y compara
#   los resultados.
(cat alum03.sal prof03.sal pas03.sal | sort | uniq > usu03) &
(cat alum04.sal prof04.sal pas04.sal | sort | uniq > usu04) &
wait diff usu03 usu04
```

8.3 Paso de parámetros mediante función y mediante ejecución

Para manejar argumentos mediante shell usando los parámetros y el caracter especial $. Este carácter indica manejo de variable, ahora se agrega un número que indica la posición en la línea de comando. Así, la cadena "$1" indica el primer parámetro y "$2" indica el segundo. Existe un segundo carácter especial que puede ser usado. El caracter "*" es similar al meta-carácter de nombre de archivo, la cadena "$*" indica que va trabajar con todos los argumentos.

```
#!/bin/sh
arg1="$1"
arg2="$2"
arg3="$3";
echo Primeros tres parámetros "$arg1" "$arg2" and "$arg3"
```

La variable “$#” muestra el número de argumentos que se pasaron al script. La variable “$$” muestra el ID del proceso que actualmente esta ejecutando shellscript. La variable “$?” muestra lael error del programa.

```
#!/bin/sh
 # script1
 exit 12
```

```
#!/bin/sh
 script1
 echo $?
```

9. Referencias.

1. B. Fox, C. Ramey: *“BASH(1)” (páginas de manuales de BASH v2.5b)”*. 2.002.
2. C. Ramey, B. Fox: “*Bash Reference Manual, v2.5b*”. Free Software Foundation, 2.002.
3. Mike G, trad. G. Rodríguez Alborich: *“Programación en BASH – COMO de Introducción”*. 2.000.
4. M. Cooper: *“Advanced Bash-Scripting Guide, v2.1”*. Linux Documentation Project, 2.003.
5. R. M. Gómez Labrador: *“Administración Avanzada de Sistemas Linux (3ª edición)”*. Secretariado de Formación del PAS (Universidad de Sevilla), 2.004.

i. Secretariado de Formación y Perfeccionamiento del P.A.S. de la Universidad de Sevilla: http://www.forpas.us.es/
ii. Proyecto GNU.: http://www.gnu.org/
iii. The Linux Documentation Project (TLDP): http://www.tldp.org/
iv. Proyecto HispaLinux (LDP-ES): http://www.hispalinux.es/