

Taller de programación shell

Este es el material de apoyo para el *taller de programación shell* organizado por la Asociación de Usuarios de Software Libre de Elche, Kleenux, el 3 de Abril de 2004. Esta es la revisión 1.

Autor: Juan J. Martínez <jjm@usebox.net>, con la colaboración de Paco Brufal <pbrufal@mutoid.org>

Copyright © 2004 Juan J. Martínez y Paco Brufal. Se permite la copia textual y distribución de este documento en su totalidad, por cualquier medio, siempre y cuando se mantenga esta nota de copyright.

Más información sobre la Asociación de Usuarios de Software Libre de Elche en <http://www.kleenux.org>.

Tabla de contenidos

1. Introducción: ¿Qué es un shell?	1
2. Manejo básico del shell	2
3. Comandos UNIX	6
4. Programación shell	12
A. Ejercicios resueltos	19
B. Inicio del CD-ROM del taller (Knoppix)	21
C. Documentación adicional	22

1. Introducción: ¿Qué es un shell?

Es una parte fundamental de todo sistema operativo que se encarga de ejecutar órdenes básicas para el manejo del sistema.

Suelen incorporar características como:

- control de procesos
- redirección de ficheros
- lenguaje para escribir pequeños programas

Hay muchos:

- command.com cmd.exe - DOS y derivados
- ksh - korn shell de UNIX
- csh - C shell, similar en sintaxis al lenguaje de programación C

- bsh - Bourne shell
- tcsh, zsh, ash, ...
- bash - Bourne Again shell, el shell mayoritario de sistemas Linux

2. Manejo básico del shell

Introduzcamos cuatro comandos básicos para ver esta parte:

- echo : repite los argumentos en la salida estándar (ej. pantalla)
- ls : lista el contenido de un directorio
- cat : muestra el contenido de un fichero
- more : muestra el contenido de un fichero haciendo pausas entre pantallas si el fichero es muy largo

2.1. La línea de comandos

Escribimos:

```
$ ls
```

Pulsamos **ENTER**.

En la línea de comandos podemos usar las siguientes (combinaciones de) teclas:

IZQUIERDA

Se mueve a la izquierda (anda!)

DERECHA

Se mueve a la derecha

ARRIBA

Vamos hacia atrás en el historial de comandos

ABAJO

Volvemos hacia adelante en el historial de comandos

FIN

CTRL + e

Vamos al final de la línea

INICIO

CTRL + a

Vamos al principio de la línea

CTRL + d

Fin de fichero

CTRL + l

Borra la pantalla

Mayúsculas + RePag

Hace scroll hacia arriba en la pantalla

Mayúsculas + AvPag

Hace scroll hacia abajo en la pantalla

2.2. Patrones de sustitución

Hay ciertos caracteres que el shell sustituirá por otro contenido en base a unas reglas.

*

cualquier cadena de texto

Ejemplo:

```
$ echo /usr/*
```

?

un solo caracter cualquiera

Ejemplo:

```
$ echo /usr/?bin
```

```
$ echo /usr/????
```

[...]

cualquiera de los caracteres entre corchetes

Ejemplo:

```
$ echo /usr/[aeiou]*
```

2.3. Redirección de ficheros

Definición clásica: Un fichero informático es una entidad lógica compuesta por una secuencia de bits, almacenada en un sistema de archivos ubicada en la memoria de un ordenador.

En UNIX todo es un fichero. Es decir, se pueden aplicar a todos los objetos las acciones que soporta un fichero en su definición clásica: abrir, cerrar, leer, escribir, etc.

Ficheros estándar que están abiertos para todo programa en ejecución:

- Entrada estándar /dev/stdin (ej. Teclado)
- Salida estándar /dev/stdout (ej. Pantalla)
- Salida de error /dev/stderr (ej. Pantalla)

Otros ficheros especiales:

- /dev/null -> "la nada de UNIX"
- /dev/zero -> fuente infinita de ceros

- /dev/random -> datos aleatorios

¿Qué pasa cuando ejecutamos **ls**?

2.3.1. Redirección stdout a fichero

```
$ ls > salida
```

```
$ cat salida
```

2.3.2. Redirección stderr a fichero

```
$ ls pirulotropical 2> error
```

```
$ cat error
```

2.3.3. Redirección stdin a fichero

```
$ cat < /dev/stdin
```

2.3.4. Añadir redirección a fichero

```
$ echo añadir datos >> salida
```

```
$ cat salida
```

```
$ ls uh 2>> error
```

```
$ cat error
```

2.3.5. Redirección "documentos empotrados"

```
$ cat << FINDOCUMENTO
```

Esto es un documento empotrado. El comando `cat` va a tomar un fichero desde `stdin` que acaba cuando encuentra el delimitador...

```
FINDOCUMENTO
```

2.4. Tuberías

Una tubería es un fichero especial con dos extremos de, forma que lo que escribimos en un lado de la tubería va a parar al otro. En inglés: *pipe*.

```
$ ls /usr/bin | more
```

La salida del comando `ls /usr/bin` va a parar a la entrada del comando `more`.

Las tuberías permiten combinar la funcionalidad de distintos comandos. Para ello la mayor parte de los programas permiten acceder a sus funcionalidades desde la línea de comandos.

2.5. Control de procesos

Un proceso es un programa en ejecución con características propias (memoria, pila, puntero de programa, número identificador único, etc.).

En UNIX todo proceso es 'hijo' de otro, es decir, es creado por otro proceso, considerado como el proceso 'padre'. Todos los procesos descienden de un proceso principal llamado 'Init', que al iniciar el sistema tiene el número de proceso 1. El proceso 'init' es el único que no tiene 'padre'. Existe la posibilidad que un proceso 'padre' termine de manera inesperada y errónea (debido a un fallo de programación, por ejemplo), este proceso 'padre' terminaría, pero podría dejar procesos 'hijo' en ejecución. Cuando un proceso 'hijo' queda descolgado del proceso 'padre', se dice que entra en un estado 'zombie' porque su padre no está esperando cuando acaba.

Cuando ejecutamos un comando en el shell se crea un proceso que es hijo de ese shell y el shell espera a que el proceso termine para volver a tomar el control.

También es importante destacar que cada nuevo proceso creado hereda ciertas características de su padre, como: las variables de entorno y los ficheros abiertos.

```
[ shell ] - ejecuta ls -> [ ls : shell espera ] - fin ls -> [ shell ]
```

Desde el shell podemos, a parte de crear procesos, manipular a los hijos creados por ese shell.

Control de procesos del shell:

CTRL + c

finaliza el proceso

comando &

permite ejecutar un comando en segundo plano

Ejemplo:

```
$ ls &
bin/ doc/ mail/ src/ tmp/
[1] 23704
[1] + Done                ls -F
```

CTRL + z

parar un proceso en primer plano el shell toma el control y 'duerme' al proceso que estaba en ejecución

Ejemplo:

```
$ ls /bin/ | more
CTRL + z
[1] + Done                ls -F /bin/ |
Stopped                  more
```

jobs

muestra información sobre los procesos parados dependientes de este shell

Ejemplo:

```
$ jobs
[1] + Done                ls -F /bin/ |
Stopped                   more
```

fg

continua un proceso parado, en primer plano. Si existen varios procesos parados, podemos ponerle como argumento el índice que el comando jobs nos mostró.

bg

continua un proceso parado, en segundo plano. Si existen varios procesos parados, podemos ponerle como argumento el índice que el comando jobs nos mostró.

3. Comandos UNIX

Hay más de 300 comandos UNIX distintos. Vamos a estudiar por encima las características de unos cuantos, los suficientes como para poder hacer pequeños programas.

Aunque los nombres de los comandos siguen una lógica, muchas veces esta se ha perdido con el tiempo. En consecuencia tenemos muchos comandos que no sabemos lo que hacen y, por lo tanto, no los utilizamos. No hay que saber como funcionan todos los comandos, pero sí es interesante conocerlos y más o menos saber que hacen. Para un uso más en profundidad consultaremos la página del manual.

ls

muestra el contenido de un directorio

echo

hace eco en pantalla

Ejemplo:

```
$ echo hola mundo!
```

cat

muestra el contenido de un fichero

more

muestra el contenido de un fichero haciendo pausas entre pantallas si el fichero es largo

man

muestra la página del manual de un comando

Ejemplo:

```
$ man ls
```

clear

borra la pantalla

cp

copia ficheros y directorios

Ejemplo:

```
$ cp fichero_original fichero_copia
```

mv

mueve ficheros

Ejemplo:

```
$ mv fichero fichero2
```

rm

borra ficheros

Ejemplo:

```
$ rm fichero
```

ln

enlazar (referenciar) ficheros

Ejemplo de enlace "duro" (hardlink):

```
$ ln fichero enlace
```

Ejemplo de enlace "suave" (softlink):

```
$ ln -s fichero enlace_simbólico
```

cd

cambia de directorio de trabajo si no se indica directorio,
nos traslada a \$HOME

Ejemplo:

```
$ cd directorio
```

pwd

muestra el directorio de trabajo actual

mkdir

crea directorios

Ejemplo:

```
$ mkdir directorio
```

rmdir

borra directorios (vacíos)

Ejemplo:

```
$ rmdir directorio
```

env

muestra las variables de entorno del programa

head

muestra las n primeras líneas de un fichero (10 por defecto)

Ejemplo:

```
$ head fichero
```

tail

muestra las n últimas líneas de un fichero (10 por defecto)

Ejemplo:

```
$ tail fichero
```

grep

busca ocurrencias de una cadena en un fichero

Ejemplo:

```
$ grep cadena fichero
```

ps

muestra los procesos en el sistema

kill

Envía una señal a un proceso indicando su PID (Process IDentifier, o número único que identifica a cada proceso)

Ejemplo:

```
$ kill 1002
```

export

Exporta una variable al entorno del programa

Ejemplo:

```
$ export VARIABLE=valor
```

read

Lee una línea de la entrada estándar y la almacena en una variable

Ejemplo:

```
$ read linea
```

\$

Delante de una variable permite acceder a su contenido

Ejemplo:

```
$ echo $SHELL
```

;

Separa dos comandos en una misma línea

Ejemplo:

```
$ read linea ; echo se ha leído: $linea
```

file

indica de qué tipo es un fichero

cal

muestra el calendario del mes actual

wc

cuenta líneas, palabras o bytes en ficheros

Ejemplo:

```
$ echo hola que tal | wc
```

date

muestra hora y fecha actuales

Ejemplo:

```
$ date
```

Ejemplo de fecha en formato yyyy-mm-dd:

```
$ date "+%Y-%m-%d"
```

passwd

cambia la contraseña de un usuario

chmod

cambia los permisos de un fichero

chown

cambia el propietario de un fichero

chgrp

cambia el grupo propietario de un fichero

reset

restaura la terminal de texto

whereis

indica donde se puede encontrar un fuente, binario o manual

Ejemplo:

```
$ whereis ls
```

which

indica donde está un comando

Ejemplo:

```
$ which ls
```

locate

busca ficheros

find

búsqueda avanzada de ficheros

who

quién tiene sesión abierta en la máquina

tac

concatena ficheros y los muestra a la inversa

touch

actualiza la fecha y hora de un fichero, si no existe lo crea

Ejemplo:

```
$ touch fichero_inexistente
```

less

una versión más elaborada de **more** que permite desplazarnos por el texto, hacer búsquedas, etc.

df

muestra el espacio libre y ocupados de los discos

du

calcula el espacio de disco usado

mail

programa simple para enviar y leer correo

tar

empaquetar ficheros

Ejemplo empaquetar:

```
$ tar cvf fichero.tar directorio
```

Ejemplo desempaquetar:

```
$ tar xvf fichero.tar
```

gzip

comprimir un fichero

gunzip

descomprimir un fichero comprimido con **gzip**

zcat

muestra el contenido de un fichero comprimido con **gzip**

ldd

muestra las librerías que usa un programa

halt

apaga la máquina

reboot

reinicia la máquina

shutdown

apaga o reinicia la máquina

true

cierto, o uno

false

falso, o cero

exit

termina la sesión y muestra el *login* del sistema

logout

termina la sesión y muestra el *login* del sistema

seq

genera una secuencia de números

Ejemplo:

```
$ seq 1 10
```

cut

elimina partes de ficheros

Ejemplo:

```
$ echo hola que tal | cut -d " " -f 2
```

awk

escáner de patrones y lenguaje de programación para procesar textos

Ejemplo:

```
$ echo hola que tal | awk '{ print $1 "!", $2, $3 "?" }'
```

tr

elimina o traduce caracteres

Ejemplo:

```
$ echo hola que tal | tr a A
```

sed

realiza transformaciones en flujos de bytes

Ejemplo:

```
$ echo hola que tal | sed 's/a/A/g'
```

(substituye las 'a' por 'A' en todo el flujo)

fmt

da formato a cada párrafo de un fichero

sort

ordena ficheros de texto

sleep

detiene el proceso durante n segundos

Ejemplo:

```
$ sleep 5 ; echo Han pasado 5 segundos
```

uniq

lee de **stdin** y compara líneas adyacentes escribiendo las líneas únicas a **stdout**

4. Programación shell

Ahora que sabemos manejarnos con el shell y conocemos unos pocos comandos, vamos a comenzar a hacer pequeños programas que interpretará el shell.

En esta parte necesitaremos un editor de texto plano, como pueden ser: vi, emacs, joe, mcedit, nano, kwrite, gedit, etc. Cualquiera de ellos vale, siempre que guardemos el texto como **text/plain**.

4.1. Mira mamá, soy un script!

Vamos a crear un fichero, **script.sh**, con el siguiente contenido:

```
#!/bin/sh
echo Mira mamá, soy un script!
```

Intentamos ejecutarlo con **./script.sh** y no funciona. Esto es porque la extensión **sh** no es lo que hace que sea ejecutable. Para que se pueda ejecutar tenemos que darle permisos de ejecución:

```
$ chmod +x script.sh
```

Ahora sí es ejecutable.

El programa consta de dos líneas:

- Un comentario, desde la aparición de **#** hasta el final de esa línea, donde se indica al shell con la secuencia **#!/bin/sh** que **/bin/sh** es el programa que se debe usar para ejecutar este fichero.
- Un comando que muestra un texto en **stdin**. Es la primera línea "ejecutable", ya que los comentarios son ignorados por el intérprete.

4.2. Variables

Una variable es un contenedor. Consta de un identificador que la distingue de otra (su nombre) y de un contenido. La relación entre variable y contenido es de equivalencia.

Por lo general las variables en shell no tienen tipos asociados y se definen de la siguiente forma:

```
identificador = contenido
```

Ejemplos:

```
# i vale 1
i=1
```

```
# I vale echo
I=echo
```

```
# msg vale Hola mundo!
msg="Hola mundo!"
```

Cuidado: si dejamos espacios entre el = y el identificador o el valor el shell creará que son comandos a ejecutar y no la asignación de una variable.

Para acceder al contenido de una variable empleamos **\$** delante de su identificador:

```
$identificador
```

Ejemplos:

```
$ i=1 ; echo $i
```

```
$ msg="Mola mundo!" ; echo $msg
```

```
$ fu=echo; $fu goo!
```

Cuando empleamos **\$identificador** el shell busca el valor almacenado en la variable asociada a ese identificador y lo utiliza para reemplazar esa ocurrencia de **\$identificador**.

4.3. Línea de comandos

Cuando se ejecuta nuestro programa en shell hay una serie de variables que siempre estarán disponibles, entre ellas las que nos permiten acceder a los distintos argumentos con los que fue ejecutado nuestro script.

\$0
contiene el nombre nombre de nuestro script

\$#
el número de parámetros con los que se ha invocado al shell

\$n
los parámetros, con n de 1 a 9 (a **\$#**)

\$\$
el PID de nuestro proceso

\$*
todos los parámetros menos **\$0**

4.4. La salida de los programas

Cuando se ejecuta un programa, un comando UNIX es un programa, podemos, a parte de redirigir su entrada y su salida, recoger el resultado de su ejecución y su salida.

El resultado es un valor numérico, por lo general cero si todo ha ido bien, y distinto de cero si ha habido alguna clase de error.

La salida del programa es lo que obtendríamos en **stdin** y **stdout**.

\$?
resultado del último programa ejecutado

Ejemplo:

```
$ ls pirulotropical 2> /dev/null ; echo $?
```

```
$ ls > /dev/null ; echo $?
```

\$(comando)
la salida de comando (esto es equivalente al uso de comillas invertidas, pero por simplicidad vamos a utilizar esta versión)

Ejemplo:

```
$ salida_ls=$(ls) ; echo $salida_ls
```

exit ENTERO

termina nuestro programa con el valor de salida ENTERO

Ejercicio 1: realizar un script que dado un directorio, cree un archivo tar comprimido con gzip y con nombre igual a la fecha en formato yyyy-mm-dd seguido del nombre del directorio acabado en .tar.gz. Ejemplo: aplicado sobre tmp obtendríamos -> 2004-04-03tmp.tar.gz.

4.5. Operaciones aritméticas

Para que el shell evalúe una operación aritmética y no la tome como argumentos de un comando, por ejemplo:

```
$ echo 1+1
```

Si queremos que sustituya la operación por su valor emplearemos:

\$(expresión)

evalúa la expresión aritmética y reemplaza el bloque por el resultado

Ejemplo:

```
$ echo $((1+1))
```

Algunos operadores aritméticos soportados:

```
+ suma
* multiplicación
- resta
/ división entera
% resto de la división entera
( ) agrupar operaciones
```

Ejercicio 2: realizar un script que dado un número 'n' muestre los diez primeros elementos de su tabla de multiplicar, mostrando el resultado en la forma: i x n = resultado.

4.6. Condicionales

Existe un comando para evaluar condiciones, y que nos permitirá que nuestros programas "tomen decisiones":

```
test expresion
[ expresion ]
```

Este comando evalúa *expresion*, y si evalúa a cierto, devuelve cero (**true**), o en otro caso 1 (**false**). Si no hay expresión, **test** siempre devuelve falso. Este comportamiento puede ser algo confuso, ya en lógica los valores cierto y falso suelen ser al contrario.

test soporta gran cantidad de operadores, algunos son:

```
-d fichero
    cierto si fichero existe y es un directorio
-e fichero
    cierto si fichero existe, independientemente del tipo que sea
-f fichero
    cierto si fichero existe y es un fichero normal
-r fichero
    cierto si fichero existe y se puede leer
-s fichero
    cierto si fichero existe y tiene tamaño mayor que cero
-w fichero
    cierto si fichero existe y es se puede escribir sobre él
-x fichero
    cierto si fichero existe y es ejecutable

n1 -eq n2
    cierto si los enteros n1 y n2 son iguales
n1 -ne n2
    cierto si los enteros n1 y n2 no son iguales
n1 -gt n2
    cierto si el enteros n1 es mayor que n2
n1 -ge n2
    cierto si los enteros n1 y n2 son iguales o n1 es mayor
    que n2
n1 -lt n2
    cierto si el enteros n1 es menor que n2
n1 -le n2
    cierto si los enteros n1 y n2 son iguales o n1 es menor
    que n2

s1 = s2
    cierto si las cadenas de texto s1 y s2 son idénticas
s1 != s2
    cierto si las cadenas de texto s1 y s2 no son idénticas
s1 < s2
    cierto si la cadena de texto s1 es menor que s2
s1 > s2
    cierto si la cadena de texto s1 es mayor que s2
-n cadena
    cierto si la longitud de la cadena de texto es distinta de cero

! expresion
    cierto si expresion es falsa (negación)
expresion1 -a expresion2
    cierto si expresion1 y expresion2 son ciertas
expresion1 -o expresion2
    cierto si expresion1 o expresion2 son ciertas
```

Además existen los operadores lógicos **&&** (AND, multiplicación lógica) y **||** (OR, suma lógica), que se puede aplicar al valor de salida de los programas:

```
$ true && true ; echo $?
```

```

$ true && false ; echo $?
$ false && true ; echo $?
$ false && false ; echo $?

$ true || true ; echo $?
$ true || false ; echo $?
$ false || true ; echo $?
$ false || false ; echo $?

```

El sistema de evaluación del shell es *perezoso* y va de izquierda a derecha. Si se encuentra la suma lógica **true || ALGO**, ALGO no se evaluará porque se asume que cierto o falso o cierto o cierto siempre es cierto (toma ya).

4.6.1. if ... then ... [else ...]

Esta es la principal estructura que nos permitirá ejecutar un bloque de código, o (alternativamente) otro, dependiendo de como se evalúe una condición.

```

if CONDICION; then
    bloque de comandos
fi

if CONDICION; then
    bloque de comandos b1
else
    bloque de comandos b2
fi

```

En el primer caso el bloque de comandos se ejecutará solo si la condición es evaluada a cierto. En el segundo caso el bloque b1 se ejecutará si la condición es evaluada a cierto, y sino se ejecutará el bloque b2.

La condición puede ser, por ejemplo, una llamada al comando **test** o una operación lógica entre los valores de salida de diferentes comandos.

```

read linea
# comparamos cadenas de texto, así que usamos comillas
if [ "$linea" = "secreto" ]; then
    echo bingo!
fi

if ! $(ping -c 1 192.168.0.100 > /dev/null); then
    echo La máquina 192.168.0.100 no responde
else
    echo La máquina 192.168.0.100 está ahí!
fi

```

Ejercicio 3: realizar un script que, dado un número, indique si es o no divisible entre 101. Si no se proporciona un número debe mostrar como usar el programa.

4.7. Bucles

El shell aporta mecanismos para realizar tareas repetitivas mediante el empleo de estructuras que permiten repetir un bloque de comandos.

4.7.1. for ... in ...

Esta estructura permite repetir un bloque de comandos asignando valores de una serie a una variable en cada iteración.

```
for VARIABLE in SERIE; do
    bloque de comandos
done
```

En cada iteración la variable `VARIABLE` toma un valor de `SERIE`, que en caso de no contener elementos hará que no se ejecute nada y se devuelva un valor 0. En caso de que se ejecuten comandos, el resultado devuelto tras el bucle es el del último comando ejecutado.

Ejemplos de bucle:

```
# equivalente a seq 1 5
for i in 1 2 3 4 5; do
    echo $i
done
```

```
# lo mismo pero con palabras
for palabra in uno dos tres cuatro cinco; do
    echo $palabra
done
```

Ejercicio 4: realizar un script que dado un número 'n' muestre los diez primeros elementos de su tabla de multiplicar, mostrando el resultado en la forma: $i \times n = \text{resultado}$. Emplear un bucle y `seq` (si está disponible). Si no se proporciona un número, mostrar como se usa el programa.

Ejercicio 5: realizar un script que dado una lista de directorios, cree un archivo tar comprimido con `gzip` con nombre igual a la fecha en formato `yyyy-mm-dd.tar.gz`. Además se generará un fichero `yyyy-mm-dd.lst` con los nombres de los directorios contenidos en el archivo tar, UNO POR LINEA usando un bucle. Si el fichero `lst` existe, mostrar un error y terminar el programa. Si alguno de los elementos no es un directorio, mostrar un error y finalizar el programa.

4.7.2. Rompiendo un bucle: break

En cualquier momento un bucle puede ser interrumpido mediante el uso de `break`, de forma que tras ser ejecutado ese comando el control pasa al siguiente comando después del `done`.

Ejemplo de uso de `break`:

```
for elemento in *; do
    echo Primer elemento $elemento

    break

    echo Esto nunca se llega a ejecutar
```

```
done

echo Seguimos con el programa
```

4.7.3. while ...

Se trata de otra estructura de bucle que permite ejecutar un bloque de comandos mientras se evalúe una condición a cierto:

```
while CONDICION; do
    bloque de comandos
done
```

Cada iteración se evalúa la condición y en el momento que no sea cierta, el bucle termina.

Ejemplos de bucles:

```
# equivalente a seq 1 5
i=1
while [ $i -lt 6 ]; do
    echo $i
    i=$((i+1))
done

# lee de stdin hasta que se introduzca 'quit'
read linea

while [ "$linea" != "quit" ]; do
    read linea
done
```

Ejercicio 6: realizar un script que permita adivinar al usuario cuál es su PID. El script pide un número al usuario y cada vez que lo haga debe indicar al usuario si el PID es mayor o menor que el número introducido. Cuando se adivina el valor, se deben mostrar los intentos empleados.

A. Ejercicios resueltos

Ejercicio 1

```
#!/bin/sh

tar cvfz $(date "+%Y-%m-%d")$1.tar.gz $1
```

Ejercicio 2

```
#!/bin/sh

echo 1 x $1 = $((1*$1))
echo 2 x $1 = $((2*$1))
```

```
echo 3 x $1 = $((3*$1))
echo 4 x $1 = $((4*$1))
echo 5 x $1 = $((5*$1))
echo 6 x $1 = $((6*$1))
echo 7 x $1 = $((7*$1))
echo 8 x $1 = $((8*$1))
echo 9 x $1 = $((9*$1))
echo 10 x $1 = $((10*$1))
```

Ejercicio 3

```
#!/bin/sh

if [ $# -ne 1 ]; then
    echo Uso: $0 numero
    exit 1
fi

if [ $((($1%101)) -eq 0 ]; then
    echo $1 es divisible entre 101
else
    echo $1 no es divisible entre 101
fi
```

Ejercicio 4

```
#!/bin/sh

for i in $(seq 1 10); do
    echo $i x $1 = $((i*$1))
done
```

Ejercicio 5

```
#!/bin/sh

if [ $# -eq 0 ]; then
    echo Uso: $0 directorios
    exit 1
fi

FECHA=$(date "+%Y-%m-%d")

if [ -e $FECHA.lst ]; then
    echo Error: El fichero lst ya existe.
    exit 1
fi

for directorio in $*; do
    if ! [ -d $directorio ]; then
        echo Error: $directorio no es un directorio
        exit 1
    fi
done
```

```
# podría ser: echo $* | tr " " "\n" > $FECHA.lst
# pero no se usaría el bucle ;)

touch $FECHA.lst

for directorio in $*; do
    echo $directorio >> $FECHA.lst
done

tar cfz $FECHA.tar.gz $*
```

Ejercicio 6

```
#!/bin/sh

PID=$$
INTENTOS=1

echo -n "Cual es mi PID? "
read linea

while [ $linea -ne $PID ]; do

    if [ $linea -gt $PID ]; then
        echo Mi PID es menor que $linea
    else
        echo Mi PID es mayor que $linea
    fi

    echo -n "Cual es mi PID? "
    read linea

    INTENTOS=$((INTENTOS+1))
done

echo Acertaste! Has empleado $INTENTOS intentos
```

B. Inicio del CD-ROM del taller (Knoppix)

El CD que se entrega contiene la distribución de Linux llamada Knoppix. Esta distribución se caracteriza por ser ejecutada desde el propio CD-ROM, sin alterar ningún dato del disco duro.

Para arrancar el sistema con la distribución Knoppix, encenderemos el ordenador, y durante la secuencia de inicio veremos un mensaje que nos indica qué tecla o combinación de teclas debemos pulsar para acceder a la configuración de la BIOS (Basic Input/Output) del sistema. Normalmente suele ser pulsando la tecla "Supr", "F2" o "Esc".

Una vez dentro de la BIOS, buscaremos un menú llamado "Boot", "Bios Features" o "Boot Sequence". Ésto cambia entre los distintos fabricantes de BIOS, e incluso entre distintas versiones de un mismo fabricante.

Lo ideal será buscar una opción que nos permita modificar la secuencia de arranque. Posiblemente tengas la siguiente configuración: "A,C,CDROM". Esto significa: primero intentar arrancar desde la unidad "A:" (el disquete), en caso de fallo intentar con la unidad "C:" (el primer disco duro), y por último el CD-ROM.

Pues bien, para poder arrancar con el CD de Knoppix debemos configurar esta opción para que la unidad CDROM sea probada ANTES que la unidad C. Cualquier combinación servirá: CDROM,A,C ,o CDROM,C,A , o CDROM,SCSI,A, son algunos ejemplos. Anota en un papel la configuración original para dejarla conforme estaba cuando hayas terminado.

Una vez configurada la secuencia de arranque, elegimos la opción "Save Changes and Exit". Antes de confirmar la salida, abrimos la bandeja del CD, insertamos el CDROM de Knoppix, y cerramos la bandeja. Confirmamos en la BIOS que queremos guardar los cambios y seguidamente el ordenador reiniciará.

Antes de ejecutar la distribución Knoppix, se nos presenta un "prompt" en el que pone "boot:".

En ese "prompt" podemos escribir una serie de parámetros que configurarán de inicio la distribución Knoppix. Pulsando F2 veremos la ayuda, y los distintos parámetros que podemos poner. Desde ahí podemos cambiar el lenguaje del teclado, o el entorno gráfico, e incluso desactivar el hardware que no vayamos a usar.

Recuerda: Knoppix se ejecuta directamente desde el CDROM, así que no tengas miedo, ningún dato del disco duro del ordenador se borrará si tu no quieres.

C. Documentación adicional

Algunos enlaces interesantes sobre programación shell:

- Advanced Bash-Scripting Guide: <http://www.tldp.org/LDP/abs/html/>
- Korn Shell (ksh) Programming: <http://www.bolthole.com/solaris/ksh.html>
- BASH Programming - Introduction HOW-TO:
<http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>
- Bourne Shell Programming: <http://steve-parker.org/sh/sh.shtml>
- Página de la asociación, Kleenux.org: <http://www.kleenux.org/>