Tema 1. Introducción a Linux

1.0. Contenido

1.1. Directorios

1.1.1. *Pathname*

Denominamos ***pathname*** a la ruta que se sigue para acceder a un fichero desde el directorio principal, denominado ***root***. Existen distintas formas de formular el *pathname*:

- *Pathname* completo: `/home/user/Documentos/Pruebas/prueba.txt`
- Usando `~`: `~/Documentos/Pruebas/prueba.txt`
- Usando `.` (desde el directorio `Documentos`): `./Pruebas/prueba.txt`
- Usando `..` (desde otro directorio `Imagenes`): `../Pruebas/prueba.txt`

Los nombres de los ficheros son ***case sensitive***, es decir, no es lo mismo `prueba.txt` que `Prueba.txt`.

1.1.2. Directorios principales

Existen una serie de directorios que son comunes a todo sistema Linux. Algunos de estos son:

- ***home***: funciona como nuestra carpeta de trabajo.
- ***bin***: almacena los programas que se usan durante la inicialización del sistema.
- ***dev***: guarda ficheros especiales, que incluyen impresoras o terminales.
- ***etc***: contiene programas y ficheros para poder administrar el sistema.

- ***usr***: tiene aplicaciones y recursos de los usuarios.

1.1.3. Permisos

Para gestionar estos ficheros, Linux define una serie de permisos que hacen su acceso más o menos restrictivo. Existen 3 tipos distintos de permisos que se pueden combinar entre ellos:

- Modo lectura (***r***): permite ver el contenido de un fichero o listar el contenido de una carpeta.
- Modo escritura (***w***): permite editar el contenido de un fichero o crear y eliminar contenido de una carpeta.
- Modo ejecución (***x***): permite ejecutar un fichero o entrar en un directorio.

Cada fichero tiene asociados una serie de permisos, que se muestran en 3 bloques de la forma `rwx`, el primero se corresponde a los permisos del propietario, el segundo a los del grupo y el último a los del usuario.

1.2. *Shell*

La ***shell*** es la interfaz entre el usuario y el sistema operativo, es decir, el programa encargado de procesar los comandos que introduce el usuario para que los entienda el núcleo del sistema. En Linux, predominan la *shell* del tipo **línea de texto** (CLI).

1.2.1. Comandos

El usuario se comunica con la *shell* a través de **comandos**, una serie de instrucciones mediante el uso de palabras clave. Estos comandos pueden ir acompañados de **argumentos**. Existen distintos tipos de comandos:

Comandos de Linux

Existe la posibilidad de proporcionar a la *shell* varios comandos para que los ejecute de distintas formas. Para ello utilizamos los siguientes comando especiales:

- `;`: ejecuta dos o más comandos de forma consecutiva.
- `&`: ejecuta dos o más comandos a la vez.
- `|`: ejecuta dos o más comandos de forma consecutiva, utilizando para algunos comandos la salida de los anteriores.

1.2.2. Comandos de entrada y salida

Cuando se ejecutan varios comandos a la vez, en ocasiones no se sabe cuál va a acabar primero, si se van a pisar, etc. Para evitar los problemas que esto pueda ocasionar, frecuentemente se imprime la salida de estos comandos en un fichero, en lugar de mostrarla por pantalla. Para ello, se pueden usar dos estructuras:

- `<<comando1>> & <<comando2>> > <<fichero>>`: crea o sobrescribe el fichero.
- `<<comando1>> & <<comando2>> >> <<fichero>>`: crea o añade la información al final del fichero.

Si no queremos que la información se muestre por pantalla, pero tampoco queremos guardarla en un fichero, podemos dirigir la salida a la dirección `/dev/null`.

También se pueden crear redirecciones de entrada, utilizando la estructura `<<comando>> < <<fichero>>`.

1.3. *Scripts*

Un ***script*** es un fichero de texto plano que contiene una secuencia de comandos. Por lo general, a los *scripts* de *bash* se les pone la extensión `.sh`.

La *shell* sabe cómo debe ejecutar estos ficheros gracias a una primera línea denominada ***shebang*** que tiene la estructura `#![ruta del intérprete que ejecuta el fichero]`.

Los *scripts* se puede ejecutar de distintas formas:

- Ejecutando el comando `bash` y pasando el nombre del script como argumento.
- Haciendo el *script* ejecutable, con el comando `chmod +x`, para que la *shell* haga la llamada.
- Copiando el código del *script* y pegándolo en la *shell*, de forma manual o con el comando `source`.

1.3.1. Variables en *bash*

Las variables en *bash* se identifican por su nombre, usando letras, números y guiones bajos. Se definen cuando se inicializan y se accede a su valor precediendo el nombre por `$`.

Se pueden usar variables sin inicializar, aunque estás estarán vacías. Para estos casos, se puede establecer un valor por defecto en las variables, haciendo que cuando la variable dependa de un argumento que no se proporciona, esta no se muestre vacía. Para ello, basta con escribir `${<<variable>>:-<<valor>>}`.

Si se desea eliminar una variable se puede usar el comando `unset`, mientras que si se quiere aislar una variable, para usarla en la definición de otra, por ejemplo, esta se escribe entre `{}`.

El ámbito de una variable es la *shell* en la que se define, no pudiendo acceder a ellas desde una nueva *shell*. Sin embargo, se puede especificar que una variable esté disponible en las *shell* hijas de la *shell* en la que se define utilizando el comando `export`.

Para evitar la modificación del valor de una variable (si es que así se desea) se puede declarar esta como una variable de ‘solo lectura’, usando el comando `readonly`.

1.3.2. Variables como resultado de comandos

Uno de los principales objetivos de las variables es almacenar resultados intermedios obtenidos a partir de la ejecución de algunos comandos. Para ello, se utiliza la sustitución de comandos, que cuenta con dos sintaxis diferentes: `'<<comando>>'` o `$(<<comando>>)`.

1.3.3. Variables de entrada y salida

Otro de los objetivos de las variables es almacenar información que el usuario introduce por teclado. Para ello, se usa el comando `read <<variable1>> <<variable2>>`, que permite guardar la entada en una variable.

1.3.4. Operaciones con *bash*

Bash permite hacer operaciones básicas con números enteros. Para ello, existen dos aproximaciones diferentes:

- El comando `expr` que requiere separar cada término con un espacio en blanco y marcar con `\` los caracteres especiales (`expr 2 \* 3`).

- El comando `let` o `(())` siguiendo las mismas normas. En este caso, el comando no devuelve nada, solo realiza la operación.

1.4. Condicionales

Cuando se realiza un *script* con cierta complejidad es fundamental el control de flujo de la ejecución, para poder detectar errores. Para ello, se usa el comando `test <<expr>>` o `[<<expr>>]`, añadiendo los operadores que se requieran.

```
file = "file"
if [ -e $file ]; then
  echo "$file exists"
else
  echo "$file is a nice name"
fi
```

Este código comprueba si `file` existe o no mostrando el resultado por pantalla.

1.5. Bucles

Otro aspecto muy importante en la programación es la repetición de tares. Para ello podemos utilizar *bucles for*, en los que una variable va tomando todos los distintos valores de una lista.
Por otra parte, los *bucles while* repiten un procedimiento mientras se cumpla una condición.

```
for var in {1..8..2}
do
  echo $var
done
```

Este código muestra en pantalla los números 1, 3, 5 y 7.

```
i=1
while [ $i -lt 6]
do
  echo $i
  ((i=i+1))
done
```

Este código muestra por pantalla los números del 1 al 5.

1.6. Funciones

Para evitar las repeticiones de código, se pueden usar funciones, que facilitan la tarea.

```
msg="Initial message"
hello() {
  # local msg
  if [ $# -ge 1 ]; then
    msg="Hello, $#"
  else
    msg="Hello, uknown user"
  fi
}

hello student
echo "Message: $msg"
```

1.7. Compilación de proyectos

1.7.1. Modularizar

Cuando programamos en C, estos son los tipos de fichero que usamos:

- **Archivo fuente** (.c): contiene el código escrito en lenguaje C que implementa las funciones, tipos de datos, etc.
- **Archivo de cabecera** (.h): contiene las bibliotecas, la definición de los tipos de datos, los prototipos de las funciones, etc.
- **Archivo compilado** (.o): es el resultado de compilar los ficheros .c, es decir, es la traducción a lenguaje máquina.
- **Biblioteca estática** (.a): es una colección de ficheros .c compilados.

1.7.2. Compilación

La **compilación** es el proceso mediante el que un archivo escrito en lenguaje C se traduce a lenguaje máquina. Esto se puede hacer a través de entornos como *VsCode* o *NetBeans*. Sin embargo, también se pueden compilar ficheros utilizando comandos.

El comando `gcc` nos permite compilar y enlazar el programa completo. De forma básica, se usa el comando `gcc -c <<archivo.c>>`, aunque se puede cambiar la bandera `-c` por otras como `-Wall`, `-pedantic` o `-ansi`, que aportan más información:

- `-Wall:` : se muestran todos los *warnings* de compilación, como variables no usadas, no inicializadas, falta de *return* en funciones, etc.
- `-ansi`: nos asegura que el código cumple con el estándar C ANSI.
- `-pedantic`: es un estándar más restrictivo.
- `-g`: incluye información para depuración.

1.7.3. Enlazado

Un programa puede estar compuesto por varios ficheros de código fuente (.c), ficheros de cabecera (.h) y bibliotecas (.a). Para poder generar un archivo ejecutable es necesario, además de compilar los ficheros de código fuente, enlazarlos con todos los anteriores.

Para realizar el proceso de enlazado, basta con usar el comando `gcc` y la bandera `-o`, seguido del nombre del ejecutable y todos los ficheros .o a enlazar, `gcc -o <<ejectuable>> <<archivo1.o>> <<archiv2o.o>>`.

En caso de que sea necesario introducir bibliotecas externas, se pueden usar las siguientes banderas:

- `-L <<ruta del directorio>>`: incluye carpetas con librerías adicionales.
- `-l <<librería>>`: incluye una librería concreta.
- `-lm`: incluye la librería matemática.

1.7.4. *Makefile*

Cuando un programa se compone de varios ficheros de código fuente y cabecera, el proceso de compilación y enlazado se complica. Para ello, usamos los fichero ***makefile***, que nos permiten

automatizar este proceso, además de ahorrar tiempo, ya que con él, solo se recompilan los ficheros .c que hayan sufrido cambios desde la última compilación.

Los ficheros *makefile* suelen incluir una serie de **variables** declaradas al comienzo del fichero. Las más comunes son:

- `EXE`: indica el nombre del ejecutable.
- `CFLAGS`: indica las banderas de compilación.
- `CC`: indica el comando de compilación.
- `CLIB`: indica las librearías.

Para usar alguna de las variables definidas basta con invocarla entre paréntesis, precedida por `$`.

Los **comodines** permiten reducir el código del *makefile*, algunos de ellos son:

- `$@`: es el item que aparece a la izquierda de `:`.
- `$<`: es el primer item en la lista de dependencias.
- `$^`: son todos los archivos que se encuentran a la derecha de `:`.

```
calculadora: main.o real.o complejo.o
  gcc -o calculadora main.o real.o complejo.o
```

```
$(EXE): main.o real.o complejo.o
  $(CC) -o $@ $^
```

Es frecuente que el *makefile* incluya otras tareas, como limpiar los ficheros de compilación. Para ello, se usa la siguiente instrucción:

```
clean:
  rm -f *.o $(EXE)
```

1.8. Errores de programación

A la hora de programar se pueden cometer diversos tipos de errores:

- **Errores de codificación**: una vez se tiene el programa codificado, la compilación falla, impidiendo la creación del ejecutable.
- **Avisos de compilación o *warnings*:** al igual que con los errores de codificación, la compilación falla, pero sí se puede crear el ejecutable, aunque puede que el programa no funcione como debería.
- **Errores de ejecución**: el programa compila, pero no se ejecuta como es debido, en muchas ocasiones, estos errores son consecuencia de los avisos de compilación.
- **Errores ocultos**: el programa compila y se ejecuta correctamente, aunque deja de hacerlo en situaciones concretas.