Llamadas al Sistema para el sistema de archivos (I)

📝 Ejercicio 1

¿Qué hace el siguiente programa? Probad tras la ejecución del programa las siguientes órdenes del shell: `$> cat archivo` y `$> od -c archivo`

> Ver archivo `tarea1.c`

Solución al ejercicio 1

Este programa hace lo siguiente:

1. Intenta abrir `archivo`, si lo hace almacena su descriptor de archivo en `fd`, si no lo consigue muestra un mensaje de error. Para ello usa `open()`, creando un **descriptor de archivo**, con:
    - Los siguientes atributos en `flags`:
        - `O_CREAT`: Si no existe `archivo`, lo crea.
        - `O_WRONLY`: Se abre `archivo` únicamente con permisos de escritura.
    - Los siguientes atributos en `mode` especifican permisos que se crean si hay una creación de archivo involucrada.
2. Se escribe en `fd`, pasándolo como primer argumento. A continuación, pasamos `buf1`, array que contiene los datos que se desean escribir en el archivo. Finalmente especificamos el número de bytes que se desean escribir. Devuelve el número de caracteres que realmente ha escrito (por eso mostramos error si no se han escrito diez caracteres).
3. Con `lseek` nos posicionamos en un punto intermedio del archivo (no muy importante).
4. Finalmente se termina de escribir con el segundo búfer.

En efecto, una vez ejecutado el programa:

```
mianfg@mianfg-PE62-7RD:~/Sesion1$ cat archivo
abcdefghijABCDEFGHIJmianfg@mianfg-PE62-7RD:~/Sesion1$ od -c archivo
0000000   a   b   c   d   e   f   g   h   i   j  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040  \0  \0  \0  \0  \0  \0  \0  \0   A   B   C   D   E   F   G   H
0000060   I   J
0000062
```

Apréciense los caracteres nulos.

Con respecto a descriptores de archivos

Los **descriptores de archivos** son números que identifican a cada archivo abierto. Los 0, 1, 2 corresponden con la entrada estándar, la salida estándar y la salida de error estándar.

Por ejemplo, `ls` es un programa que ha sido diseñado para escribir en `1`. Podemos escribir en pantalla o redirigir la salida a un archivo (usando `>`). Otros, como `cut`, usan el `0` para leer, pudiendo usar la redirección de entrada (usando `|`).

📝 Ejercicio 2

Implementa un programa que realice la siguiente funcionalidad. El programa acepta como argumento el nombre de un archivo (pathname), lo abre y lo lee en bloques de tamaño 80 Bytes, y crea un nuevo archivo de salida, salida.txt, en el que debe aparecer la siguiente información:

```
Bloque 1
// Aquí van los primeros 80 Bytes del archivo pasado como argumento.
Bloque 2
// Aquí van los siguientes 80 Bytes del archivo pasado como argumento.
...
Bloque n
// Aquí van los siguientes 80 Bytes del archivo pasado como argumento.
```

Si no se pasa un argumento al programa se debe utilizar la entrada estándar como archivo de entrada.

Ampliación: ¿Cómo tendrías que modificar el programa para que una vez finalizada la escritura en el archivo de salida y antes de cerrarlo, pudiésemos indicar en su primera línea el número de etiquetas "Bloque i" escritas de forma que tuviese la siguiente apariencia?:

```
El número de bloques es n
Bloque 1
// Aquí van los primeros 80 Bytes del archivo pasado como argumento.
Bloque 2
// Aquí van los siguientes 80 Bytes del archivo pasado como argumento.
...
Bloque n
// Aquí van los siguientes 80 Bytes del archivo pasado como argumento.
```

Solución al ejercicio 2

> Ver archivo `ejercicio2.c`

Solución al ejercicio 2 (ampliación)

> Ver archivo `ejercicio2amp.c`

Metadatos de un archivo

- `stat` contiene metadatos del archivo (un punto cualquiera del árbol de archivos y directorios, cualquiera de los tipos de archivos especificados en el punto 4.1).
  - Hora último cambio se refiere a cambio de metadatos del archivo.
  - El **número de inodo** es el número de bloques donde se almacena.
- Como orden, `stat` nos especifica todos los metadatos de la estructura.
- `stat` varía ligeramente de `lstat`, en el caso de un archivo de tipo enlace simbólico. Con `lstat` tenemos los datos del enlace simbólico, y con `stat` los del archivo original.

> Ver archivo `tarea2.c`

Estructura `stat`: campo `st_mode`

Haciendo:

```
if ((atributos.st_mode & 0170000) == 0100000) /// archivo regular
////////                   S_IFMT     S_IFREG
```

Obtenemos el tipo de archivo, pues 017 es 1111 en binario (el resto de ceros es cada uno de los campos de permisos, en octal).

Del mismo modo podemos realizar esta macro:

```
if (S_ISREG(archivo.st_mode))
    printf("regular file\n");
```

- El **usuario efectivo** es el usuario que sirve en comprobaciones de seguridad.

```
$ ls -l /bin/ln
> -rwxr-xr-x 1 root root 67808 ene 18  2018 /bin/ln
```

No tiene el bit establecido, si lo tuviese el usuario efectivo sería `root`, pudiendo acceder el archivo al sistema como `root`.

- Ídem con **grupo efectivo**, si tuviese el bit al lanzar el proceso su grupo efectivo sería el grupo propietario, no el grupo del usuario que lo ha lanzado.
- El **sticky bit** sirve para no perder memoria (inicialmente en Unix). Ver explicación en el libro.

📝 Ejercicio 3

¿Qué hace el siguiente programa?

> Ver archivo `tarea2.c`

Solución al ejercicio 3

Este programa especifica el tipo de archivo de cada uno de los archivos pasados al ejecutar el programa, con el siguiente formato:

```
$ ./programa archivo1 archivo2 ... archivon
> archivo1: tipo de archivo
> archivo2: tipo de archivo
> ...
> archivon: tipo de archivo
```

📝 Ejercicio 4

Define una macro en lenguaje C que tenga la misma funcionalidad que la macro `S_ISREG(mode)` usando para ello los flags definidos en `<sys/stat.h>` para el campo `st_mode` de la struct `stat`, y comprueba que funciona en un programa simple. Consulta en un libro de C o en internet cómo se especifica una macro con argumento en C.

```
#define S_ISREG2(mode) ..
```

Solución al ejercicio 4

```
#define S_ISREG2(mode) (mode & S_FMT == S_IFREG)
```