1️⃣ Llamadas al sistema para el sistema de archivos (I)

Entrada/Salida

Todas las llamadas al sistema para I/O se refieren a archivos abiertos mediante *descriptores de archivo*, un entero no negativo (normalmente pequeño). Son usados para referir cualquier clase de archivo. Cada proceso tiene su propio set de descriptores de archivo.

Por convención, la mayoría de programas deben poder usar tres descriptores de archivo estándar:

Descriptor de archivo	Propósito	Nombre POSIX	Stream `stdio`
0	entrada estándar	`STDIN_FILENO`	`stdin`
1	salida estándar	`STDOUT_FILENO`	`stdout`
2	error estándar	`STDERR_FILENO`	`stderr`

Cuando nos referimos a estos descriptores podemos usar los enteros 0, 1 o 2, o preferible los estándares POSIX (definidos en `<unistd.h>`).

Metadatos de un archivo

Tipos de archivo

Propietarios de archivo

Cada archivo tiene asociado un ID de usuario (UID) y un ID de grupo (GID). Estos IDs determinan a qué usuario y grupo pertenece el archivo.

Propiedad de archivos nuevos

Cuando se crea un archivo, el ID del usuario es tomada por el ID del usuario efectivo del proceso. El ID del grupo puede ser tomada por el ID del grupo efectivo del proceso (comportamiento System V), o del ID del grupo del directorio padre (comportamiento BSD).

Permisos

En archivos regulares

La máscara de permisos de archivo, correspondiente a los últimos 12 bits del campo `st_mode` de la estructura `stat`, se divide en tres categorías:

- **Propietario o usuario:** permisos al usuario del archivo.
- **Grupo:** permisos concedidos a todos los usuarios miembros del grupo del archivo.
- **Otros:** permisos concedidos al resto.

Pueden otorgarse tres permisos en cada categoría: de **lectura**, de **escritura** o de **ejecución**.

En directorios

Tienen el mismo esquema que en archivos. Sin embargo, los permisos se interpretan diferente:

- **Lectura:** los contenidos del directorio pueden ser listados (por ejemplo, por `ls`).
- **Escritura:** los archivos pueden ser creados y eliminados del directorio.
- **Ejecución:** los archivos pueden ser accedidos. Normalmente se denomina permiso de **búsqueda**.

Glosario práctico I: llamadas al sistema para entrada/salida

Abrir un archivo: `open()`

> Abre un archivo existente o crea y abre un nuevo archivo.

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags, ... /* mode_t mode */);
```

- **Valor de retorno:** devuelve descriptor de archivo si éxito, -1 en error.
  - Devuelve el menor descriptor de archivo
- `pathname`: identifica el archivo a abrir. Si es un nombre simbólico, es derreferenciado.
- `flags`: máscara de bits que especifican el *modo de acceso* al archivo. Pueden ser (aunque hay más, ver `man 2 open`):

Modo de acceso (flag)	Descripción
`O_RDONLY`	Abrir el archivo para sólo lectura
`O_WRONLY`	Abrir el archivo para sólo escritura
`O_RDWR`	Abrir el archivo tanto para lectura como para escritura

- `mode` especifica los permisos del archivo.

Lectura de archivos: `read()`

> Lee datos de un archivo referenciado por su descriptor.

```
#include <unistd.h>

ssize_t read(int fd, void *buffer, size_t count);
```

- **Valor de retorno:** el número de bytes leídos, 0 en EOF (end-of-file) o -1 en error.
  - `ssize_t` es un entero con signo usado para retener un recuento de bytes o una indicación de error con -1.
- `fd`: descriptor de archivo.
- `buffer`: da la dirección del búfer de memoria del que se leerán los datos. Debe ser al menos de `count` bytes de longitud.
- `count`: especifica el número máximo de bytes a leer.
  - `size_t` es un tipo entero sin signo.
- **Comportamiento:** puede que una llamada a `read` cuente menos bytes que los pedidos. Esto puede ser, en archivos regulares, que estábamos cerca del fin de archivo. Tiene también otras particularidades en otros tipos de archivo.

Escritura a archivos: `write()`

> Escribe datos a un archivo abierto.

```
#include <unistd.h>

ssize_t write(int fd, void *buffer, size_t count);
```

- **Valor de retorno:** número de bytes escritos, -1 en error.
- `fd`: descriptor de archivo.
- `buffer`: da la dirección del búfer de memoria del que se escribirán los datos. Debe ser al menos de `count` bytes de longitud.
- `count`: especifica el número máximo de bytes a leer.

- **Comportamiento:** una devolución satisfactoria de `write()` no garantiza que los datos se hayan transferido a disco, porque el kernel realiza el búfering de la E/S para reducir la actividad de disco y expedir llamadas `write()`.

Cambiando la posición actual (*offset*): `lseek()`

> Ajusta la posición actual de un archivo

Para cada archivo, el kernel guarda una posición actual, denominada *offset* del archivo. Esta es la localización donde comenzará el próximo `write()` o `read()`. Es expresado como una posición en bytes relativa al comienzo del archivo. El primer byte de un archivo está en el offset 0.

Cuando se abre un archivo, es ajustado directamente a 0, y es automáticamente ajustado tras llamar a `read()` o a `write()`, para que apunte al bit justo después del último leído/escrito. De este modo, podemos hacerlos de forma sucesiva.

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

- **Valor de retorno:** nuevo *offset* si éxito, -1 en error.
- `fd`: descriptor de archivo.
- `offset`: especifica un valor en bits.
  - `off_t` es un tipo entero con signo.
- `whence`: indica el punto desde el que se interpreta el *offset*, es uno de los siguientes valores:

`whence` (en `lseek()`)	**Descripción**
`SEEK_SET`	El *offset* del archivo se establece `offset` bytes desde el principio del archivo.
`SEEK_CUR`	El *offset* del archivo es ajustado `offset` bytes relativo al *offset* actual.
`SEEK_END`	El *offset* del archivo es ajustado al tamaño del archivo más `offset`. En otraspalabras, `offset` es interpretado respecto al byte siguiente al último byte del archivo.

Cerrar un archivo: `close()`

> Cierra un descriptor de archivo, liberándolo para un uso posterior por parte del proceso.

Cuando un proceso termina, todos sus descriptores de archivo son automáticamente cerrados.

```
#include <unistd.h>

int close(int fd);
```

- **Valor de retorno:** 0 en éxito, -1 en error.

Glosario práctico II: metadatos de archivos

Recopilar información de archivos: `stat()` (`lstat()`, `fstat()`)

> Recopilan información de un archivo, principalmente del inodo del archivo.

```
#include <sys/stat.h>

int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

- **Valor de retorno:** todos devuelven 0 en éxito, -1 en error.

Estas llamadas difieren en la forma en el que se especifica el archivo:

- `stat()` devuelve información sobre un archivo nombrado.
- `lstat()` es similar a `stat()`, excepto que si el archivo es un enlace simbólico, devuelve la información del enlace en sí, en lugar del archivo al que referencia.
- `fstat()` devuelve información sobre un archivo referido mediante su descriptor.

Tanto `stat()` como `lstat()` son llamadas al sistema que no requieren permisos sobre el archivo. Sin embargo, el permiso de ejecución (búsqueda) es requerido en todos los directorios padre especificados en `pathname`. La llamada a `fstat()` siempre es satisfactoria, si se le proporciona un descriptor válido.

La estructura `stat`

Todas estas llamadas al sistema devuelven una estructura `stat` en el búfer apuntado por `statbuf`. Esta estructura tiene la siguiente forma:

```
struct stat {
    dev_t     st_dev;   // IDs del dispositivo donde se encuentra el archivo
    ino_t     st_ino;   // número de inodo del archivo
    mode_t    st_mode;    // tipo de archivo y permisos
    nlink_t   st_nlink;   // número de enlaces (duros) al archivo
    uid_t     st_uid;   // ID de usuario del propietario de archivo
    gid_t     st_gid;   // ID de grupo del propietario del archivo
    dev_t     st_rdev;    // IDs para archivos especiales de dispositivos
    off_t     st_size;    // tamaño total del archivo (bytes)
    blksize_t st_blksize; // tamaño de bloque óptimo para I/O (bytes)
    blkcnt_t  st_blocks;  // número de bloques (de 512B) asignados
    time_t    st_atime;   // último acceso al archivo
    time_t    st_mtime;   // última modificación al archivo
    time_t    st_ctime;   // último cambio de estado
};
```

`st_mode`: tipo de archivo y permisos

El campo `st_mode` es una máscara de bits que tiene un doble propósito: identificar el tipo de archivo y especificar sus permisos. Los bits se configuran de la siguiente forma:

```
|| Tipo archivo  ||                     Permisos                        ||
-------------------------------------------------------------------------
||   |   |   |   || U | G | T || R | W | X || R | W | X || R | W | X ||
-------------------------------------------------------------------------
                              ||  Usuario  ||   Grupo   ||   Otros   ||
```

- Extraer el tipo de archivo: hacemos AND (`&`) con `S_IFMT`.
    - Ahora podemos compararlo con las constantes siguientes para ver qué tipo de archivo, del tipo `S_ISxxx`, o usando las macros del mismo nombre definidas en `<sys_stat.h>`.

Constante	Macro POSIX	Descripción
`S_ISLNK`	`S_ISLNK()`	Enlace simbólico
`S_ISREG`	`S_ISREG()`	Archivo regular
`S_ISDIR`	`S_ISDIR()`	Directorio
`S_ISCHR`	`S_ISCHR()`	Dispositivo de caracteres
`S_ISBLK`	`S_ISBLK()`	Dispositivo de bloques
`S_ISFIFO`	`S_ISFIFO()`	Cauce con nombre (FIFO)
`S_ISSOCK`	`S_ISSOCK()`	Socket

- Extraer los permisos del archivo: realizamos comprobaciones con las constantes definidas en `<sys/stat.h>` del tipo `S_ISxxx`.

Constante	Bit de permiso
`S_ISUID`	Set-user-ID
`S_ISGID`	Set-group-ID
`S_ISVTX`	Sticky (no POSIX)
`S_IRUSR`	Lectura para usuario
`S_IWUSR`	Escritura para usuario
`S_IXUSR`	Ejecución para usuario
`S_I[RWX]GRP`	Lectura/escritura/ejecución para grupo
`S_I[RWX]OTH`	Lectura/escritura/ejecución para otros

Sobre set-user/group-ID y *sticky bits*

Más información en:

- Set-user-ID y set-group-ID: sección 9.3 de *The Linux Programming Interface*, M. Kerrisk.
- Sticky bits: sección 15.4.5 de *The Linux Programming Interface*, M. Kerrisk.