2️⃣ Llamadas al sistema para el sistema de archivos (II)

Máscara de creación

Cuando creamos archivos y directorios, los permisos que se proporcionan son los especificados en el argumento `mode` de la llamada al sistema. Sin embargo, estos ajustes son modificados por la máscara de modo de creación, conocida como *umask*. Este es un atributo de proceso que especifica qué bits de permisos siempre deben permanecer *off* cuando se crean nuevos directorios o archivos por un procesos.

Normalmente, un proceso hereda su *umask* del terminal padre. En la mayoría de shells, los archivos de inicialización inician su máscara al valor octal 022 (`----w--w-`), para no permitir la escritura por parte de *grupo* o por parte de *otros*.

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 de un archivo

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): creación de ficheros, modificación de permisos

Modificación de máscara de creación de ficheros: `umask()`

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

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

mode_t umask(mode_t mask);
```

- **Valor de retorno:** siempre devuelve de forma satisfactoria el valor del *umask* anterior.
  - `pathname`: identifica el archivo a abrir. Si es un nombre simbólico, es derreferenciado.
- `mask`: es especificado haciendo OR (`|`) de las constantes para bits de permisos de archivos (ver glosario práctico de sesión anterior, constantes `S_IRUSR`, `S_IWGRP`...).
- **Comportamiento:** establece la máscara de usuario a `mask & 0777`.

Modificación de permisos: `chmod()` y `fchmod()`.

> Cambian los permisos de un archivo.

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

int chmod(const char *pathname, mode_t mode);
```

```
#include <sys/stat.h>
#define _XOPEN_SOURCE 500  // o: #define _BSD_SOURCE

int fchmod(int fd, mode_t mode);
```

- **Valor de retorno:** ambos devuelven 0 en éxito, -1 en error.
- `chmod()` modifica los permisos del archivo nombrado en `pathname`. Si el archivo es un enlace simbólico, modifica los permisos del archivo al que referencia (un archivo simbólico siempre se crea con los permisos de lectura, escritura y ejecución para todos los usuarios, y no pueden ser cambiados; estos permisos se ignoran en la dereferenciación).
- `fchmod()` modifica los permisos del archivo referido por el descriptor `fd`.
- `mode`: especifica los nuevos permisos del archivo, tanto numéricamente (octal) como haciendo OR (`|`) de las constantes para bits de permisos de archivos (ver glosario práctico de sesión anterior, constantes `S_IRUSR`, `S_IWGRP`...).
    - Para cambiar los permisos de un archivo, el proceso debe ser privilegiado (`CAP_FOWNER`, permisos de root o superusuario con UID efectivo 0) o su ID de usuario efectivo debe coincidir con el propietario (ID de usuario) del archivo.

      Si el UID efectivo del proceso no es cero y el grupo del fichero no coincide con el ID de grupo efectivo del proceso o con uno de sus ID's de grupo suplementarios, el bit `S_ISGID` se desactivará, aunque esto no provocará que se devuelva un error.

      Dependiendo del sistema de archivos, los bits `S_ISUID` y `S_ISGID` podrían desactivarse si el archivo es escrito. En algunos sistemas de archivos, solo el root puede asignar el *sticky bit*, lo cual puede tener un significado especial (por ejemplo, para directorios, un archivo sólo puede ser borrado por el propietario o el root).

Glosario práctico (II): manejo de directorios

Lectura de directorios: `opendir()` y `readdir()`

> Nos permiten abrir un directorio y recopilar los nombres de los archivos que contienen, uno a uno.

```
#include <dirent.h>

DIR *opendir(const char *dirpath);
```

- **Valor de retorno:** devuelve un puntero a la estructura de tipo `DIR`, llamada *stream* de directorio, o `NULL` en caso de error.
    - `DIR` es un *stream de directorio*, un *handle* que el invocador pasa a otras funciones.
- `dirpath`: especifica el directorio a abrir.

- **Comportamiento:** tras la devolución de `opendir()`, el *stream* de directorio es posicionado a la primera entrada en la lista de directorios.

También existe la función `fdopendir()`, que hace lo mismo que `opendir()`, sólo que en lugar de la ruta se le pasa un descriptor de archivo:

```
#include <dirent.h>

DIR *fdopendir(int fd);
```

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

- **Valor de retorno:** devuelve un puntero a la estructura alojada estáticamente describiendo la siguiente entrada del directorio, o `NULL` en caso de fin de directorio o en error.
- **Comportamiento:** lee la entrada donde esté situado el puntero de lectura de un directorio ya abierto cuyo stream se pasa a la función. Después de la lectura adelanta el puntero una posición.

```
struct dirent {
    ino_t d_ino;  // número de inodo del archivo
    char  d_name[]; // nombre del archivo, terminado en NULL
}
```

Esta estructura se sobreescribe en cada llamada a `readdir()`.

Cierre de directorios: `closedir()`

> Cierra el directorio especificado.

```
#include <dirent.h>

int closedir(DIR *dirp);
```

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

Vuelta al principio: `rewinddir()`

> Mueve el stream de vuelta al principio, de modo que la próxima llamada a `readdir()` comenzará de nuevo con el primer archivo del directorio.

```
#include <dirent.h>

void rewinddir(DIR *dirp);
```

Acceso avanzado: `seekdir()` y `telldir()`

- `seekdir()`: permite situar el puntero de lectura de un directorio (debe usarse en combinación con `telldir()`).
- `telldir()`: devuelve la posición del puntero de lectura de un directorio.

Más información en las entradas de manual.

Glosario práctico (III): recorrer un sistema de archivos con `nftw()`

`nftw()` permite "andar" recursivamente por todos los subdirectorios de un directorio realizando una operación (por ejemplo, llamar una función definida por el programador) para cada archivo del subárbol.

La llamada al sistema `nftw()` para *file tree walking*

> Recorre el árbol del directorio especificado por `dirpath` y llama a la función definida por el programador, `func`, una vez por cada entrada del árbol.

```
#define _XOPEN_SOURCE 500
#include <ftw.h>

int nftw(const char *dirpath,
         int (*func) (const char *pathname, const struct stat *statbuf,
                      int typeflag, struct FTW *ftwbuf),
         int nopenfd, int flags);
```

- **Valor de retorno**: 0 tras un recorrido satisfactorio del árbol completo, -1 en error, o el primer valor no-cero retornado por una llamada a `func`.

Comportamiento de la función

La función recorre el árbol de directorios especificado por `dirpath` y llama a la función `func` definida por el programador para cada archivo del árbol. Por defecto, `ntfw()` realiza un recorrido no ordenado en preorden del árbol, procesando primero cada directorio antes de procesar los archivos y subdirectorios dentro del directorio.

Mientras se recorre el árbol, la función `nftw()` abre al menos un descriptor de archivo por nivel del árbol.

El parámetro `nopenfd` especifica el máximo número de descriptores que puede usar. Si la profundidad del árbol es mayor que el número de descriptores, la función evita abrir más cerrando y reabriendo descriptores.

El argumento `flags` es creado mediante un OR (`|`) con cero o más constantes, que modifican la operación de la función:

Constante	Descripción
`FTW_DIR`	Realiza un `chdir()` (cambia de directorio) en cada directorio antes de procesar su contenido. Se utiliza cuando `func` debe realizar algún trabajo en el directorio en el que el archivo especificado por su argumento `pathname` reside.

`FTW_DEPTH`	Realiza un recorrido postorden del árbol. Esto significa que `nftw()` llama a `func` sobre todos los archivos (y subdirectorios) dentro del directorio antes de ejecutar `func` sobre el propio directorio.
`FTW_MOUNT`	No cruza un punto de montaje. Por consiguiente, si uno de los subdirectorios es un punto de montaje, no lo atraviesa.
`FTW_PHYS`	Por defecto, `nftw()` desreferencia los enlaces simbólicos. Este *flag* le dice que no lo haga. En su lugar, un enlace simbólico se pasa a `func` con un valor `typedflag` de `FTW_SL`.

Para cada archivo, `nftw()` pasa cuatro argumentos al invocar a `func`.

- El primero, `pathname`, indica el nombre del archivo, que puede ser absoluto o relativo dependiendo de si dirpath es de un tipo u otro.
- El argumento `statbuf` es un puntero a una estructura `stat` conteniendo información del archivo.
- El tercer argumento, `typeflag`, suministra información adicional sobre el archivo, y tiene uno de los siguientes nombres simbólicos:

Nombre simbólico	Descripción
`FTW_D`	Es un directorio.
`FTW_DNR`	Es un directorio que no puede leerse (no se lee sus descendientes).
`FTW_DP`	Estamos haciendo un recorrido posorden de un directorio, y el ítem actual es un directorio cuyos archivos y subdirectorios han sido recorridos.
`FTW_F`	Es un archivo de cualquier tipo diferente de un directorio o enlace simbólico.
`FTW_NS`	`stat` ha fallado sobre este archivo, probablemente debido a restricciones de permisos. El valor `statbuf` es indefinido.
`FTW_SL`	Es un enlace simbólico. Este valor se retorna sólo si `nftw()` se invoca con `FTW_PHYS`.
`FTW_SLN`	El ítem es un enlace simbólico perdido. Este se da cuando no se especifica `FTW_PHYS`.

- El cuarto elemento de func es `ftwbuf`, es decir, un puntero a una estructura que se define de la forma:

```
struct FTW {
    int base; // desplazamiento (offset) de la parte base del pathname
    int level;  // profundidad del archivo en el recorrido del árbol
}
```

    - El campo base es un desplazamiento entero de la parte del nombre del archivo (el componente después del último `/`) del `pathname` pasado a `func`.

Cada vez que es invocada, `func` debe retornar un valor entero que es interpretado por `nftw()`. Si retorna 0, indica a `nftw()` que continúe el recorrido del árbol. Si todas las invocaciones a `func` retornan cero, `nftw()` retornará 0. Si retorna distinto de cero, `nftw()` para inmediatamente, en cuyo caso `nftw()` retorna este valor como su valor de retorno.