6️⃣ Control de archivos y archivos proyectados a memoria

La función `fcntl()`

La función `fcntl()` realiza una gran variedad de operaciones de control sobre un descriptor de archivo abierto.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ...);
```

- **Valor de retorno:** en éxito depende de `cmd`, -1 en error.

Banderas de estado de archivo abierto

Consulta de banderas de estado: `cmd = F_GETFL`

Un uso de `fcntl()` es recopilar o modificar las banderas de estado de un archivo (los valores del argumento `flags` de la llamada a open, ver sesión 1). Para ello, especificamos `cmd` como `F_GETFL`.

```
int banderas;
banderas = fcntl(fd, F_GETFL);
```

Para comprobar el **modo de acceso** utilizamos la máscara `O_ACCMODE`, pues las constantes `O_RDONLY`, `O_WRONLY` y `O_RDWR` no se corresponden a un único bit de la bandera de estado.

```
int modo_acceso = banderas & O_ACCMODE;
if ( modo_acceso == O_WRONLY || modo_acceso == O_RDWR )
    printf("El archivo permite escritura\n");
```

Modificación de banderas de estado: `cmd = F_SETFL`

Podemos modificar los flags `O_APPEND`, `O_NONBLOCK`, `O_NOATIME`, `O_ASYNC` y `O_DIRECT`. Los intentos de modificar otras banderas serán ignorados. Esto es particularmente útil en los siguientes casos:

- El archivo no fue abierto por el programa llamador, de forma que no tiene control sobre las banderas utilizadas por `open()`. Por ejemplo, la entrada, salida y error estándares se abren antes de invocar al programa.
- Se obtuvo el descriptor del archivo a través de una llamada al sistema que no es open. Por ejemplo, se obtuvo con `pipe()` o `socket()`.

Para modificar los flags del archivo abierto, usamos `fcntl()` **dos veces**:

- Usamos `fcntl()` con `cmd = F_GETFL` primero para captar las banderas existentes.
- Modificamos las banderas captadas.
- Usamos `fcntl()` con `cmd = F_SETFL` para modificar las banderas.

Un ejemplo:

```
int banderas;

// captamos las banderas existentes
banderas = fcntl(fd, F_GETFL);

// comprobamos que no ha habido error
if ( banderas == -1 )
    perror("fcntl");

// modificamos las banderas captadas
bandera |= O_APPEND;  // le añadimos flag O_APPEND

// modificamos las banderas en fd
if ( fcntl(fd, F_SETFL, banderas) == -1 )
    perror("fcntl");
```

Duplicar descriptores de archivos: `cmd = F_DUPFD`

Al igual que con las funciones `dup()` (`dup()`, `dup2()` y `dup3()`), podemos duplicar descriptores, es decir, tener tras el éxito de la orden dos descriptores apuntando al mismo archivo abierto con el mismo modo de acceso y compartiendo el mismo puntero de lectura-escritura.

```
newfd = fcntl(oldfd, F_DUPFD, startfd);
```

Esta llamada duplica `oldfd` usando el menor descriptor de archivo no usado mayor o igual a `startfd`. Es aconsejable por tanto hacer `close(startfd)`, para asegurarnos que `oldfd` duplique a `startfd`, y no a un descriptor mayor a él.

Bloqueo de archivos

Pueden ocurir problemas si múltiples procesos intentan actualizar un archivo de forma simultánea, originando condiciones de carrera.

Existen las funciones `flock()` y `fcntl()` para realizar bloqueos sobre archivos, la principal diferencia entre ellas es que:

- `flock()` usa un cerrojo para bloquear el archivo completo.
- `fcntl()` usa cerrojos para bloquear regiones de archivos.

En general, el uso de bloqueos es como sigue:

1. Posicionar un cerrojo sobre un archivo.
2. Realizar E/S.
3. Desbloquear el archivo para que otro proceso pueda bloquearlo.

Podemos usarlo no sólo para E/S, sino para otras técnicas de sincronización.

Mezclando bloqueo y funciones de `stdio`

Debido al búfering que usa `stdio` en espacio de usuario, hemos de tener cuidado al usar funciones de `stdio` con las técnicas de sincronización mencionadas. El problema es que un búfer de entrada puede ser llenado antes de situar un cerrojo, o que una salida puede limpiarse antes de que se elimine un cerrojo. Podemos evitar estos problemas:

- Usando `read()` y `write()` en lugar de funciones de `stdio`.
- Limpiando el flujo de `stdio` inmediatamente después de situar el cerrojo sobre el archivo, y de nuevo tras liberar el cerrojo.
- Aunque sea algo menos eficiente, podemos deshabilitar el búfering de `stdio` con `setbuf()` o similar.

Bloqueo de registros (zonas de archivo) con `fcntl()`

En general, la forma de usar `fcntl()` para esto es:

```
struct flock flockstr;

// ajustar campos de flockstr para describir el lock

fcntl(fd, cmd, &flockstr);  // insertar lock definido por flockstr
```

La estructura `flock`

```
struct flock {
    short l_type;  // tipo de bloqueo:
             // F_RDLCK, F_WRLCK, F_UNLCK
    short l_whence;  // cómo interpretar l_start:
             // SEEK_SET, SEEK_CUR, SEEK_END
    off_t l_start;   // offset desde el que empieza el bloqueo
    off_t l_len;   // no. bytes a bloquear; 0 significa hasta EOF
    off_t l_pid;   // proceso que previene bloqueo (sólo F_GETLK)
}
```

- `l_type`: indica el tipo de bloqueo que queremos utilizar:

Tipo de cerrojo	Descripción
`F_RDLCK`	Colocar cerrojo de lectura
`F_WRLCK`	Colocar cerrojo de escritura
`F_UNLCK`	Eliminar cerrojo existente

- `l_whence`, `l_start`, `l_len`: especifican el rango de bytes que serán bloqueados. Los dos primeros funcionan como los arumentos `whence` y `offset` de `lseek()` (ver sesión 1).
    - `l_start`: especifica un desplazamiento que es interpretado respecto a:
        - el principio del archivo, si `l_whence` es `SEEK_SET`.
        - el desplazamiento actual del archivo, si `l_whence` es `SEEK_CUR`.
        - el final del archivo, si `l_whence` es `SEEK_END`.

      En los dos últimos casos, `l_start` puede ser un número negativo, siempre que la posición resultante no sea anterior al principio del archivo (byte 0).
    - `l_len`: entero que especifica el número de bytes a bloquear a partir de la posición definida por los otros campos.
        - El valor 0 de `l_len` tiene un significado especial: bloquear todos los bytes del archivo desde la posición especificada por `l_whence` y `l_start` hasta fin de archivo, sin importar cuánto crezca éste.

          Ejemplo: para bloquear un archivo completo especificamos `l_whence` a `SEEK_SET` y `l_start` y `l_len` a 0.
- `cmd`: especifica cómo se realizará el bloqueo.

Valor de `cmd`	Descripción
`F_SETLK`	Adquiere (`l_type` es `F_RDLCK` o `F_WRLCK`) o libera (`l_type` es `F_UNLCK`) un cerrojo sobre los bytes especificados por `flockstr`. Si hay un proceso que tiene un cerrojo incompatible sobre alguna parte de la región a bloquear, la llamada falla con el error `EAGAIN`.
`F_SETLW`	Igual que la anterior, excepto que si otro proceso mantiene un cerrojo incompatible sobre una parte de la región a bloquear, el llamador se bloqueará hasta que su bloqueo sobre el cerrojo tenga éxito. Si estamos manejando señales y no hemos especificado `SA_RESTART`, una operación `F_SETLKW` puede verse interrumpida, es decir, falla con error `EINTR`.
`F_GETLK`	Comprueba si es posible adquirir un cerrojo especificado en `flockstr`, pero realmente no lo adquiere. El campo `l_type` debe ser `F_RDLCK` o `F_WRLCK`. En este caso la estructura `flockstr` se trata como valor-resultado. Al retornar de la llamada, la estructura contiene información sobre si podemos establecer o no el bloqueo. Si el bloqueo se permite, el campo `l_type` contiene `F_UNLCK`, y los restantes campos no se tocan. Si hay uno o más bloqueos incompatibles sobre la región, la estructura retorna información sobre uno de los bloqueos, sin determinar cuál, incluyendo su tipo (`l_type`), y rango de bytes (`l_start`, `l_len`; `l_whence` siempre se retorna como `SEEK_SET`) y el identificador del proceso que lo tiene (`l_pid`).

Hay que tener mucho cuidado con `F_GETLK`, ya que puede que al usar la información que retorna, ésta esté obsoleta.

Cosas importantes a tener en cuenta en bloqueos:

- Dos procesos no pueden bloquear una misma zona: si se producen dos o más llamadas para bloquear zonas ya bloqueadas, retornan error. Si se producen dos o más llamadas para bloquear una zona no bloqueada, una de ellas es satisfactoria y el resto devolverán error.
- La consulta de bloqueos siempre es satisfactoria.
- No existe límite de bloqueos.
- Los cerrojos de registros no son heredados por `fork()` por el hijo.
- Los bloqueos se mantienen a través de `exec()`.
- Todos los hijos de una tarea comparten los mismos bloqueos.
- Los cerrojos de registros están asociados tanto a procesos como inodos. En consecuencia, cuando un proceso termina todos los cerrojos que poseía son liberados. Además, cuando un proceso cierra un descriptor se liberan todos los cerrojos que ese proceso tuviese sobre el archivo, sin importar el descriptor sobre el que se obtuvo el cerrojo o cómo se obtuvo.

Bloqueo consultivo vs. bloqueo obligatorio

Los bloqueos son **consultivos**, es decir, son usados por los programadores para asegurarse de que las operaciones de E/S u otras que hagan concurrencia sean satisfactorias. El kernel ignorará los locks al hacer sus operaciones.

Podemos habilitar el **bloqueo obligatorio** en Linux, habilitándolo en el sistema de archivos que contiene a los archivos que deseamos bloquear y en cada archivo que vaya a ser bloqueado, mediante la opción `-o mand`:

```
$> mount -o mand /dev/sda1 /archivo
```

Podemos obtener el mismo resultado especificando la bandera `MS_MANDLOCK` cuando invocamos a `mount(2)`.

Sobre un archivo, se habilita combinando la activación del bit *set-group-ID* y desactivando el bit *group-execute* (esta combinación en concreto porque no tiene otro uso):

```
$> chmod g+s,g-x archivo
```

Desde un programa podemos habilitarlo ajustando los permisos adecuadamente con `chmod()` o `fchmod()`.

Sin embargo, los cerrojos obligatorios deben ser evitados porque tienen diversas desventajas. Más información en el apartado 55.4 "Mandatory locking" de *The Linux Programming Interface*, M. Kerrisk.

El archivo `/proc/locks`

El archivo `/proc/locks` en Linux contiene información de los cerrojos creados por `flock()` y por `fcntl()`. Cada línea muestra información de un cerrojo y tiene ocho campos que indican:

1. Número ordinal del cerrojo en el set de todos los cerrojos del archivo.
2. El tipo de cerrojo:
    - `FLOCK` indica que fue creado por `flock().`
    - `POSIX` indica que fue creado por `fcntl()`.
3. El modo de bloqueo, `ADVISORY` o `MANDATORY`.
4. El tipo de bloqueo, `READ` o `WRITE`.
5. El ID del proceso que mantiene el bloqueo.
6. Los números separados por `:` que identifican el archivo sobre el que se establece el bloqueo. Estos números son números de dispositivo mayores y menores del sistema de archivos donde se encuentra el archivo, seguido de su número de inodo.
7. El byte donde comienza el cerrojo.
8. El byte donde termina el cerrojo. `EOF` indica fin de archivo.

Ejecutar una única instancia de un programa

Referencio al apartado 55.6 "Running just one instance of a program" del libro *The Linux Programming Interface*, de M. Kerrisk.

Archivos proyectados en memoria con `mmap()`

La llamada `mmap()` crea una nueva **proyección de memoria** en el espacio de memoria virtual del proceso que la llamada. Pude ser de dos tipos:

- ***File mapping***: proyecta una región de un archivo directamente a la memoria virtual del proceso que la llama. Una vez que ha sido proyectado un archivo, podemos acceder a su contenido mediante operaciones en los bytes de la región de memoria correspondiente. Las páginas de esta proyección son (automáticamente) cargadas desde el archivo como es requerido.
- ***Anonymous mapping***: una proyección anónima no tiene un archivo correspondiente. En su lugar, las páginas de la proyección son inicializadas a 0.

La memoria en la proyección de un proceso puede ser compartida con las proyecciones de otro proceso. Esto puede ocurrir de dos formas:

- Cuando dos procesos proyectan la misma región de un archivo, comparten las mismas páginas de la memoria física.
- Un proceso hijo creado con `fork()` hereda una copia de las proyecciones de su padre, y estas proyecciones se refieren a las mismas páginas de la memoria física que las del padre.

Cuando dos o más proceson comparten las mismas páginas, cada uno de ellos puede ver los cambios que hace el otro a los contenidos de la página, dependiendo de si la proyección es *privada* o *compartida*.

- ***Private mapping*** (`MAP_PRIVATE`): modificaciones a los contenidos de la proyección no son visibles a los otros procesos, y para la proyección de archivos, no son efectivas en el archivo subyacente. Aunque las páginas de proyecciones privadas son inicialmente compartidas, los cambios a los contenidos son privados a cada proceso (**copy-on-write**).
- ***Shared mapping*** (`MAP_SHARED`): las modificaciones a los contenidos de la proyección son visibles al resto de procesos que la comparten, y para una proyección de archivos, los cambios son efectivos en el archivo subyacente.

Resumimos el uso en la siguiente tabla:

Visibilidad de modificaciones	Proyección de archivo	Proyección anónima
Privada	Inicializar la memoria desde los contenidos del archivo	Asignación de memoria
Compartida	E/S proyectada a memoria; compartir memoria entre procesos (IPC)	Compartir memoria entre procesos (IPC)

Las proyecciones son perdidas cuando un proceso realiza un `exec()` (las funciones de tipo `exec...()`), pero son heredadas por el hijo de un `fork()`. También se hereda el tipo de proyección (privada o compartida).

Información de todas las proyecciones de un proceso es visible en el archivo específico de Linux `/proc/PID/maps`.

Creando una proyección: `mmap()`

> Crea una nueva proyección a memoria en el espacio de direcciones de la memoria virtual del proceso que la invoca.

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

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t
offset);
```

- **Valor de retorno:** dirección de comienzo de la proyección en éxito, `MAP_FAILED` en error.
- `addr`: indica la dirección de inicio dentro del proceso donde debe proyectarse el descriptor.
  - `NULL`: el kernel escoge la dirección más adecuada para la proyección (preferible).
- `length`: especifica el tamaño de la proyección, en bytes. Será aproximado por el kernel al múltiplo inmediatamente superior del tamaño de página.
- `prot`: máscara de bits especificando la protección que será establecida en la proyección. Puede ser `PROT_NONE` o una combinación mediante ORs (`|`) de las siguientes flags:

Valor	Descripción
`PROT_NONE`	La región no puede accederse
`PROT_READ`	Los contenidos de la región pueden ser leídos
`PROT_WRITE`	Los contenidos de la región pueden ser modificados
`PROT_EXEC`	Los contenidos de la región pueden ser ejecutados

- `flags`: máscara de bits especificando opciones que controlan varios aspectos de la operación de proyección. Sólo uno de estos valores puede ser incluido en la máscara (opcionalmente puede hacerse un OR (`|`) con `MAP_FIXED`):

Flag	Descripción
`MAP_PRIVATE`	Los cambios son privados
`MAP_SHARED`	Los cambios son compartidos
`MAP_FIXED`	Interpreta exactamente el argumento `address`
`MAP_ANONYMOUS`	Crea un mapeo anónimo
`MAP_LOCKED`	Bloquea las páginas en memoria (al estilo `mlock`)
`MAP_NORESERVE`	Controla la reserva de espacio de intercambio
`MAP_POPULATE`	Realiza una lectura adelantada del contenido del archivo
`MAP_UNITIALIZED`	No limpia (poner a cero) las proyecciones anónimas

- `fd`: descriptor del archivo a proyectar, y que una vez creada la proyección, podemos cerrar.
- `offset`: indica el inicio de archivo. El argumento `len` es, por tanto, el número de bytes a proyectar, empezando con un desplazamiento desde el inicio del archivo dado por `offset`.

Esquema básico para realizar una proyección a memoria

En general seguimos los siguientes pasos:

1. Obtenemos el descriptor del archivo con los permisos apropiados dependientes del tipo de proyección a realizar, normalmente con `open()`.
2. Pasar este descriptor de archivo a `mmap()`.

Eliminar una proyección: `munmap()`

> Realiza la operación contraria a `mmap()`, eliminando una proyección de memoria del espacio de direcciones virutales del proceso.

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

int munmap(void *addr, size_t length);
```

- `addr`: dirección de comienzo del rango de direcciones a ser desproyectadas.
- `length`: entero no negativo especificando el tamaño (en bytes) de la región para ser

desproyectada. Será desproyectada hasta el siguiente múltimo del tamaño de página.

Normalmente, desproyectamos una proyección completa. Por eso, normalmente utilizaremos en `addr` la dirección que retornó `mmap()` para la proyección, y en `length` el valor que fue usado para la llamada a `mmap()`.

Una vez desmapeada una proyección, cualquier referencia a ella generará la señal `SIGSEGV`. Si una región se declaró `MAP_PRIVATE`, los cambios que se realizaron en ella son descartados.