Prácticas de Sistemas Operativos

Módulo II. Uso de los Servicios del SO mediante la API

Sesión 6: Control de archivos y archivos proyectados en memoria

1. Objetivos principales

Esta nueva sesión nos va a permitir:

- Estudiar las órdenes de la función **`fcntl`** que nos permiten cambiar o consultar las banderas de control de acceso de un archivo abierto.
- Manejar la orden de la función **`fcntl`** que nos permite duplicar un descriptor de archivo para implementar redireccionamiento de archivos.
- Manejar las órdenes de la función **`fcntl`** que nos permiten consultar o implementar el bloqueo de una región de un archivo.
- Interfaz de programación para crear y manipular proyecciones de archivos en memoria, y uso de **`mmap()`** como IPC (InterProcess Communication).

2. La función `fcntl()`

La llamada al sistema **`fcntl`** (*file control*) es una función multipropósito que, de forma general, permite consultar o ajustar las banderas de control de acceso de un descriptor, es decir, de un archivo abierto. Además, permite realizar la duplicación de descriptores de archivos y bloqueo de un archivo para acceso exclusivo. Tiene como prototipo:

```
#include <unistd.h>

#include <fcntl.h>

int fcntl(int fd, int orden, /* argumento_orden */);

                    Retorna: si Ok, depende de orden; -1 si error.
```

El argumento **`orden`** admite un rango muy diferente de operaciones a realizar sobre el descriptor de archivo que se especifica en **`fd`**. El tercer argumento, que es opcional, va a depender de la orden indicada. A continuación, mostramos las órdenes admitidas y que describiremos en los apartados siguientes:

F_GETFL	Retorna las banderas de control de acceso asociadas al descriptor de archivo.
F_SETFL	Ajusta o limpia las banderas de acceso que se especifican como tercer argumento.

F_GETFD Devuelve la bandera *close-on-exec*[1] del archivo indicado. Si devuelve un 0, la bandera está desactivada, en caso contrario devuelve un valor distinto de cero. La bandera *close-on-exec* de un archivo recién abierto esta desactivada por defecto.

F_SETFD Activa o desactiva la bandera *close-on-exec* del descriptor especificado. En este caso, el tercer argumento de la función es un valor entero que es 0 para limpiar la bandera, y 1 para activarlo.

F_DUPFD Duplica el descriptor de archivo especificado por **fd** en otro descriptor. El tercer argumento es un valor entero que especifica que el descriptor duplicado debe ser mayor o igual que dicho valor entero. En este caso, el valor devuelto por la llamada es el descriptor de archivo duplicado (**nuevoFD = fcntl(viejoFD, F_DUPFD, inicialFD**).

F_SETLK Establece un cerrojo sobre un archivo. No bloquea si no tiene éxito inmediatamente.

F_SETLKW Establece un cerrojo y bloquea al proceso llamador hasta que se adquiere el cerrojo.

F_GETLK Consulta si existe un bloqueo sobre una región del archivo.

2.1 Banderas de estado de un archivo abierto

Uno de los usos de la función permite recuperar o modificar el modo de acceso y las banderas de estado (las especificadas en **open**) de un archivo abierto. Para recuperar los valores, utilizamos la orden **F_GETFL**:

```
int banderas, ModoAcceso;
banderas=fcntl(fd, F_GETFL);
if (banderas == -1)
      perror("fcntl error");
```

Tras lo cual, podemos comprobar si el archivo fue abierto para escrituras sincronizadas[2] como se indica:

```
if (banderas & O_SYNC)
      printf ("Las escrituras son sincronizadas \n");
```

Comprobar el modo de acceso es algo más complicado ya que las constantes **O_RDONLY** (0), **O_WRONLY** (1) y **O_RDWR** (2) no se corresponden con un único bit de la bandera de estado. Por ello, utilizamos la máscara **O_ACCMODE** y comparamos la igualdad con una de las constantes:

```
ModoAcceso=banderas & O_ACCMODE;
if (ModoAcceso == O_WRONLY || ModoAcceso == O_RDONLY)
      printf ("El archivo permite la escritura \n);
```

Podemos utilizar la orden **F_SETFL** de **fcntl()** para modificar algunas de las banderas de estado del archivo abierto. Estas banderas son **O_APPEND**, **O_NONBLOCK**, **O_NOATIME**, **O_ASYNC**, y **O_DIRECT**. Se ignorará cualquier intento de modificar alguna otra bandera.

1 Si la bandera *close-on-exec* está activa en un descriptor, al ejecutar la llamada **exec()** el proceso hijo no heredará este descriptor.

2 Consulte la bandera **O_SYNC** de la llamada **open()**.

El uso de la función **fcntl** para modificar las banderas de estado es útil en los siguientes casos:

1. 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.
2. 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 las banderas, primero invocamos a **fcntl** para obtener una copia de la bandera existente. A continuación, modificamos el bit correspondiente, y finalmente hacemos una nueva invocación de **fcntl** para modificarla. Por ejemplo, para habilitar la bandera **O_APPEND** podemos escribir el siguiente código:

```
int bandera;
bandera = fcntl(fd, F_GETFL);
if (bandera == -1)
      perror("fcntl");
bandera |= O_APPEND;
if (fcntl(fd, F_SETFL, bandera) == -1)
      perror("fcntl");
```

2.2 La función `fcntl()` *utilizada para duplicar descriptores de archivos*

La orden **F_DUPFD** de **fcntl** permite duplicar un descriptor, es decir, que cuando tiene éxito, tendremos en nuestro proceso dos descriptores apuntando al mismo archivo abierto con el mismo modo de acceso y compartiendo el mismo puntero de lectura-escritura, es decir, compartiendo la misma sesión de trabajo, como vimos en una sesión anterior (ordenes **dup** y **dup2**).

Veamos un fragmento de código que nos permite redireccionar la salida estándar de un proceso hacia un archivo (tal como hacíamos con **dup**, **dup2** o **dup3**):

```
int fd = open ("temporal", O_WRONLY);
close (1);
if (fcntl(fd, F_DUPFD, 1) == -1 ) perror ("Fallo en fcntl");
char bufer[256];
int cont = write (1, bufer, 256);
```

La primera línea abre el archivo al que queremos redireccionar la salida estándar, en nuestro ejemplo, temporal. A continuación, cerramos la salida estándar asignada al proceso llamador. De esta forma, nos aseguramos que el descriptor donde se va a duplicar está libre. Ya podemos realizar la duplicación de **fd** en el descriptor 1. Recordad que el tercer argumento de **fcntl** especifica que el descriptor duplicado es mayor o igual que el valor especificado. En nuestro ejemplo, como hemos realizado un **close(1)**, el descriptor duplicado es el 1. Tras definir el búfer de escritura, realizamos una operación **write(1, …)** que escribe en el archivo temporal, ya que este descriptor apunta ahora al archivo abierto por **open**. Podéis esbozar cómo se redireccionaría la entrada estándar.

Actividad 2.2 Trabajo con la llamada al sistema `fcntl()`

Ejercicio 1. Implementa un programa que admita t argumentos. El primer argumento será una orden de Linux; el segundo, uno de los siguientes caracteres "<" o ">", y el tercero el nombre de un archivo (que puede existir o no). El programa ejecutará la orden que se especifica como argumento primero e implementará la redirección especificada por el

segundo argumento hacia el archivo indicado en el tercer argumento. Por ejemplo, si deseamos redireccionar la entrada estándar de **sort** desde un archivo **temporal**, ejecutaríamos:

```
$> ./mi_programa sort "<" temporal
```

Nota. El carácter redirección (**<**) aparece entre comillas dobles para que no los interprete el shell sino que sea aceptado como un argumento del programa **mi_programa**.

Ejercicio 2. Reescribir el programa que implemente un encauzamiento de dos órdenes pero utilizando **fcntl**. Este programa admitirá tres argumentos. El primer argumento y el tercero serán dos órdenes de Linux. El segundo argumento será el carácter "|". El programa deberá ahora hacer la redirección de la salida de la orden indicada por el primer argumento hacia el cauce, y redireccionar la entrada estándar de la segunda orden desde el cauce. Por ejemplo, para simular el encauzamiento **ls|sort**, ejecutaríamos nuestro programa como:

```
$> ./mi_programa2  ls "|" sort
```

2.3 La función `fcntl()` *y el bloqueo de archivos*

Es evidente que el acceso de varios procesos a un archivo para leer/escribir puede producir condiciones de carrera. Para evitarlas debemos sincronizar las acciones de éstos. Si bien podríamos pensar en utilizar semáforos, el uso de cerrojos de archivos es más corriente debido a que el kernel asocia automáticamente los cerrojos con archivos. Tenemos dos APIs para manejar cerrojos de archivos:

- **flock()** que utiliza un cerrojo para bloquear el archivo completo.
- **fcntl()** que utiliza cerrojos para bloquear regiones de un archivo.

El método general para utilizarlas tiene los siguientes pasos:

1. Posicionar un cerrojo sobre el archivo.
2. Realizar las entradas/salidas sobre el archivo.
3. Desbloquear el archivo de forma que otro proceso pueda bloquearlo.

Si bien, el bloqueo de archivos se utiliza en conjunción con E/S de archivos, se puede usar como técnica general de sincronización. Por ejemplo, varios procesos cooperantes que bloquean un archivo para indicar que un proceso está accediendo a un recurso compartido, como una región de memoria compartida, que no tiene por qué ser el propio archivo.

Como consideración inicial y dado que la biblioteca **stdio** utiliza búfering en espacio de usuario, debemos tener cuidado cuando utilizamos funciones de *stdio* con las técnicas que vamos a describir a continuación. El problema proviene de que el búfer de entrada puede llenarse antes de situar un cerrojo, o un búfer de salida puede limpiarse después de eliminar un cerrojo. Algunos medios para evitar estos problemas podemos:

- Realizar las E/S utilizando **read()** y **write()** y llamadas relacionadas en lugar de utilizar la biblioteca **stdio**.
- Limpiar el flujo (stream) **stdio** inmediatamente después de situar un cerrojo sobre un archivo, y limpiarlo una vez más inmediatamente antes de liberar el cerrojo.
- Si bien a coste de eficiencia, deshabilitar el búfering de **stdio** con **setbuf()** o similar.

Debemos distinguir dos tipos de bloqueo de archivos: consultivo y obligatorio. Por defecto, el tipo de bloqueo es **bloqueo consultivo**, esto significa que un proceso puede ignorar un cerrojo situado por otro proceso. Para que el bloqueo consultivo funcione, cada proceso que accede a un archivo debe cooperar situando un cerrojo antes de realizar una operación de E/S. Por contra, en un **bloqueo obligatorio** se fuerza a que un proceso que realiza E/S respete el cerrojo impuesto por otro proceso.

2.3.1 Bloqueo de registros con `fcntl()`

Como hemos indicado, **`fcntl()`** puede situar un cerrojo en cualquier parte de un archivo, desde un único byte hasta el archivo completo. Esta forma de bloqueo se denomina normalmente bloqueo de registros[3].

Para utilizar **`fcntl`** en el bloqueo de archivos, debemos usar una estructura **`flock`** que define el cerrojo que deseamos adquirir o liberar. Se define como sigue:

```
struct flock {
      short l_type; /* Tipo de cerrojo: F_RDLCK, F_WRLCK, F_UNLCK */
      short l_whence; /*Interpretar l_start: SEEK_SET,SEEK_CURR,SEEK_END*/
      off_t l_start; /* Desplazamiento donde se inicia el bloqueo */
      off_t l_len; /* Numero bytes bloqueados: 0 significa "hasta EOF" */
      pid_t l_pid; /* Proceso que previene nuestro bloqueo(solo F_GETLK)*/
};
```

El elemento **`l_type`** indica el tipo de bloqueo que queremos utilizar y puede tomar uno de los siguientes valores: **`F_RDLCK`** para un cerrojo de lectura, **`F_WRLCK`** para un cerrojo de escritura, y **`F_UNLCK`** que elimina un cerrojo.

A la vista de lo indicado, si fd es el descriptor de un archivo previamente abierto donde queremos situar el bloqueo, la forma general de utilizar **`fcntl`** para el bloqueo tiene el aspecto siguiente:

```
struct flock mi_bloqueo;

. . . /* ajustar campos de mi_bloqueo para describir el cerrojo a usar */

fcntl(fd, orden, &mi_bloqueo);
```

La Figura 1 muestra cómo podemos utilizar el bloqueo de archivos para sincronizar el acceso de dos procesos a la misma región de un archivo. En esta figura, asumimos que todas las peticiones a los cerrojos son bloqueantes, de forma que la espera fallará si el cerrojo está cogido por otro proceso.

Si queremos situar un cerrojo de lectura en un archivo, el archivo debe abrirse en modo lectura. De forma similar, si el cerrojo es de escritura, el archivo se abrirá en modo escritura. Para ambos tipos de cerrojos, el archivo debe abrirse en modo lectura-escritura (**`O_RDWR`**). Si usamos un cerrojo que sea incompatible con el modo de apertura del archivo se producirá un error **`EBADF`**.

---

3 Si bien, el nombre es técnicamente incorrecto, ya que en sistemas tipo Unix los archivos son un flujo de bytes sin estructura de registros. Cualquier noción de registro dentro de un archivo es definida completamente por la aplicación que lo usa. En realidad, sería más correcto utilizar los términos rango de bytes, región de archivo o segmento de archivo.

El conjunto de campos **l_whence**, **l_start** y **l_len** especifica el rango de bytes que deseamos bloquear. Los primeros dos valores son similares a los campos **whence** y **offset** de **lseek()**, es decir, **l_start** especifica un desplazamiento dentro del archivo relativo a valor **l_whence** (**SEEK_SET** para el inicio del archivo, **SEEK_CUR** para la posición actual y **SEEK_END** para el final del archivo). El valor de **l_start** puede ser negativo siempre que la posición definida no caiga por delante del inicio del archivo.

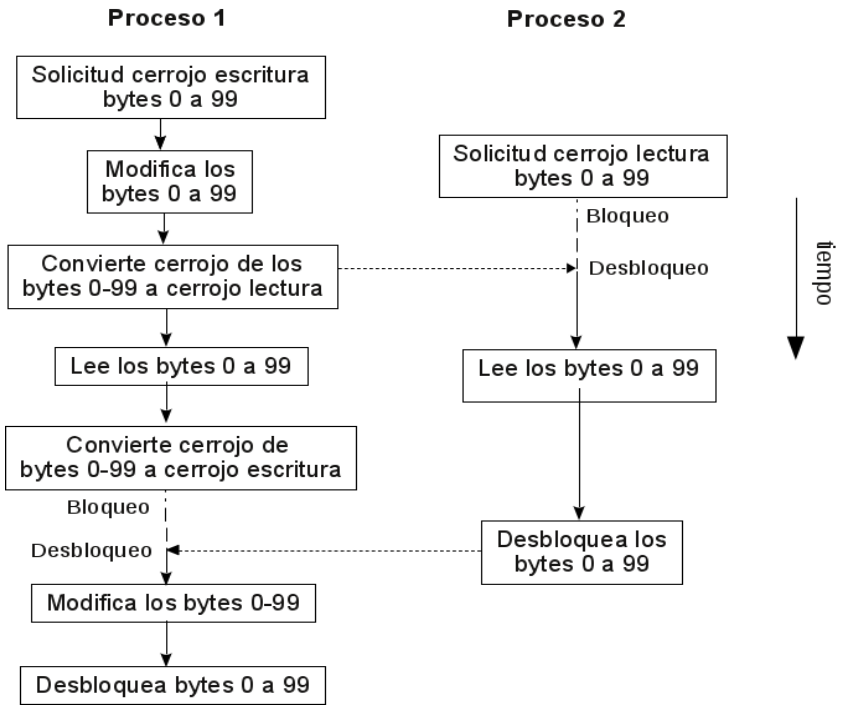


Figura 1.-Uso del bloqueo de registros para sincronizar el acceso a un archivo.

El campo **l_len** es un entero que especifica el número de bytes a bloquear a partir de la posición definida por los otros dos campos. Es posible bloquear bytes no existentes posteriores al fin de archivo, pero no es posible bloquearlos antes del inicio del archivo. A partir del kernel de Linux 2.4.21 es posible especificar un valor de **l_len** negativo. Esta solicitud se aplica al rango **(l_start – abs(l_len), l_start -1)**.

De forma general, lo adecuado sería bloquear el rango mínimo de bytes que necesitemos. Esto permite más concurrencia entre todos los procesos que desean bloquear diferentes regiones de un mismo archivo.

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 el fin de archivo sin importar cuanto crezca el archivo”. Esto es conveniente si no conocemos de antemano cuantos bytes vamos a añadir al archivo. Para bloquear el archivo completo, podemos especificar **l_whence** como **SEEK_SET** y **l_start** y **l_len** a 0.

Como vimos anteriormente, son tres las órdenes que admite **fcntl()** relativas al bloqueo de archivo. En este apartado, vamos a detallar cada una de ellas:

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 **fcntl** falla con el error **EAGAIN**.

`F_SETLKW` 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`**. Esto se puede utilizar para establecer un temporizador asociado a la solicitud de bloquea a través de **`alarm()`** o **`setitimer()`**.

`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 cual, 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`**).

Existe una condición de carrera potencial al combinar **`F_GETLK`** con un posterior **`F_SETLK`** o **`F_SETLKW`**, ya que cuando realicemos una de estas dos últimas operaciones, la información devuelta por **`F_GETLK`** puede estar obsoleta. Por tanto, **`F_GETLK`** es menos útil de lo que parece a primera vista. Incluso si nos indica que es posible bloquear un archivo, debemos estar preparados para que **`fcntl`** retorne con error al hacer un **`F_SETLK`** o **`F_SETLKW`**.

Tenemos que observar los siguientes puntos en la adquisición/liberación de un cerrojo:

- Desbloquear una región siempre tiene éxito. No se considera error desbloquear una región en la cual no tenemos actualmente un cerrojo.

- En cualquier instante, un proceso solo puede tener un tipo de cerrojo sobre una región concreta de un archivo. Situar un nuevo cerrojo en una región que tenemos bloqueada no modifica nada si el cerrojo es del mismo tipo del que teníamos, o se cambia automáticamente el cerrojo actual al nuevo modo. En este último caso, cuando convertimos un cerrojo de lectura en uno de escritura, debemos contemplar la posibilidad de que la llamada devuelva error (**`F_SETLK`**) o bloquee (**`F_SETLKW`**).

- Un proceso nunca puede bloquearse el mismo en una región de archivo, incluso aunque sitúe cerrojos mediante múltiples descriptores del mismo archivo.

- Situar un cerrojo de modo diferente en medio de un cerrojo que ya tenemos produce tres cerrojos: dos cerrojos más pequeños en el modo anterior, uno a cada lado del nuevo cerrojo (ver Figura 2). De la misma forma, adquirir un segundo cerrojo adyacente o que solape con un cerrojo existente del mismo modo produce un único cerrojo que cubre el área combinada de los dos cerrojos. Se permiten otras permutaciones. Por ejemplo, desbloquear una región situada dentro de otra región mayor existente deja dos regiones más pequeñas alrededor de la región desbloqueada. Si un nuevo cerrojo solapa un cerrojo existente con un modo diferente, el cerrojo existente es encogido, debido a que los bytes que se solapan se incorporan al nuevo cerrojo.

- Cerrar un descriptor de archivo tiene una semántica inusual respecto del bloqueo de archivos, que veremos en breve

Otro aspecto a tener en cuenta, es el interbloqueo entre procesos que se puede producir cuando utilizamos **`F_SETLKW`**. El escenario es el típico, dos procesos intentan bloquear una

región previamente bloqueada por el otro. Para evitar esta posibilidad, el kernel comprueba en cada nueva solicitud de bloqueo realizada con **`F_SETLKW`** si puede dar lugar a un interbloqueo. Si fuese así, el kernel selecciona uno de los procesos bloqueados y provoca que su llamada **`fcntl()`** falle con el error **`EDEADLK`**. En los kernel de Linux actuales, se selecciona al último proceso que hizo la llamada, pero no tiene porque ser así en futuras versiones, ni en otros sistemas Unix. Por tanto, cualquier proceso que utilice **`F_SETLKW`** debe estar preparado para manejar el error **`EDEADLK`**. Las situaciones de interbloqueo son detectadas incluso cuando bloqueamos múltiples archivos diferentes, como ocurre en un interbloqueo circular que involucra a varios procesos.

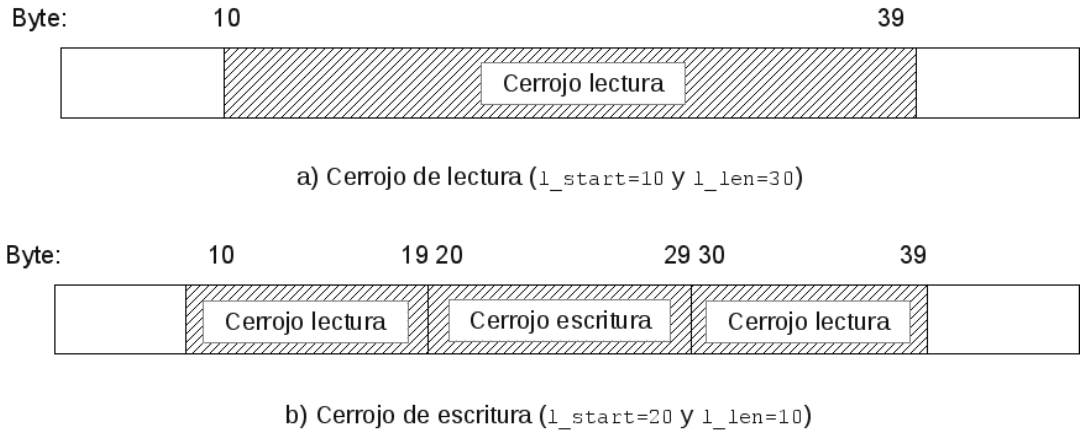


Figura 2.- Partiendo un cerrojo de lectura mediante un cerrojo de escritura.

Respecto a la inanición que puede producirse cuando un proceso intente realizar un cerrojo de escritura cuando otros procesos intentan establecer cerrojos de lectura, en Linux debemos indicar que se siguen las siguientes reglas (que no tienen que cumplirse en otros sistemas Unix):

- El orden en el que se sirven las solicitudes de cerrojos no esta determinado. Si varios procesos esperan para obtener un cerrojo, el orden en el que se satisfacen las peticiones depende de como se planifican los procesos.

- Los escritores no tienen prioridad sobre los lectores, ni viceversa.

El programa **Tarea 13.3** toma uno o varios pathnames como argumento y, para cada uno de los archivos especificados, se intenta un cerrojo consultivo del archivo completo. Si el bloqueo falla, el programa escanea el archivo para mostrar la información sobre los cerrojos existentes: el nombre del archivo, el PID del proceso que tiene bloqueada la región, el inicio y longitud de la región bloqueada, y si es exclusivo (“w”) o compartido (“r”). El programa itera hasta obtener el cerrojo, momento en el cual, se procesaría el archivo (esto se ha omitido) y, finalmente, se libera el cerrojo.

Tarea 13.3.- Bloqueo de múltiples archivos.

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
       struct flock cerrojo;
       int fd;
       while (--argc >0 ) {
```

```
            if ((fd=open(*++argv, O_RDWR)) == -1) {
            perror("open fallo"); continue;
      }
      cerrojo.l_type =F_WRLCK;
      cerrojo.l_whence =SEEK_SET;
      cerrojo.l_start =0;
      cerrojo.l_len =0;
      /* intentamos un bloqueo de escritura del archivo completo */

      while (fcntl(fd, F_SETLK, &cerrojo) == -1) {
      /*si el cerrojo falla, vemos quien lo bloquea*/
            while(fcntl(fd, F_GETLK, &cerrojo) != -1 && cerrojo.l_type != F_UNLCK ) {
                  printf("%s bloqueado por %d desde %d hasta %d para %c", *argv,
                  cerrojo.l_pid, cerrojo.l_start, cerrojo.l_len,
                  cerrojo.l_type==F_WRLCK ? 'w':'r'));
                  if (!cerrojo.l_len) break;
            cerrojo.l_start +=cerrojo.l_len;
            cerrojo.l_len=0;
            } /*mientras existan cerrojos de otros procesos*/
      } /*mientras el bloqueo no tenga exito */

      /* Ahora el bloqueo tiene exito y podemos procesar el archivo */
      . . .
      /* Una vez finalizado el trabajo, desbloqueamos el archivo entero */

      cerrojo.l_type =F_UNLCK;
      cerrojo.l_whence =SEEK_SET;
      cerrojo.l_start =0;
      cerrojo.l_len =0;
      if (fcntl(fd, F_SETLKW, &cerrojo) == -1) perror("Desbloqueo");
      }
      return 0;
}
```

En sistemas Linux, no existe límite teórico al número de cerrojos de archivos. El limite viene impuesto por la cantidad de memoria disponible.

Respecto a la eficiencia de adquirir/liberar un cerrojo, indicar que no hay una respuesta fija ya que depende de las estructuras de datos del kernel utilizadas para mantener el registro de cerrojos y la localización de un cerrojo particular dentro de la estructura de datos. Si suponemos un gran número de cerrojos distribuidos aleatoriamente entre muchos procesos, podemos decir que el tiempo necesario para añadir o eliminar un cerrojo crece aproximadamente de forma lineal con el numero de cerrojos ya utilizados sobre un archivo.

Al manejar el bloqueo de registros con **`fcntl()`** debemos tener en cuenta las siguientes consideraciones relativas a la semántica de herencia y liberación:

- Los cerrojos de registros no son heredados con **`fork()`** por el hijo.

- Los bloqueos se mantienen a través de **`exec()`**.

- Todos los hilos de una tarea comparten los mismos bloqueos.

- Los cerrojos de registros están asociados tanto a procesos como a inodos. Una consecuencia esperada de esta asociación es que cuando un proceso termina, todos los cerrojos que poseía son liberados. Menos esperado es que cuando un proceso cierra un descriptor, se liberan todos los cerrojos que ese proceso tuviese sobre ese archivo, sin importar el descriptor sobre el que se obtuvo el cerrojo ni como se obtuvo el descriptor (**`dup`**, o **`fcntl`**).

Los elementos vistos hasta ahora sobre cerrojos se refieren a bloqueos consultivos. Como comentamos, esto significa que un proceso es libre de ignorar el uso de **`fcntl()`** y realizar

directamente la operación de E/S sobre el archivo. El kernel no lo evitará. Cuando usamos bloqueo consultivo se deja al diseñador de la aplicación que:

- Ajuste la propiedad y permisos adecuados sobre el archivo, para evitar que los procesos no cooperantes realicen operaciones de E/S sobre él.

- Se asegure que los procesos que componen la aplicación cooperan para obtener el cerrojo apropiado sobre el archivo antes de realizar las E/S.

Linux, como otros Unix, permite que el bloqueo obligatorio de archivos con **`fcntl()`**. Es decir, cada operación de E/S sobre el archivo se comprueba para ver si es compatible con cualquier cerrojo que posean otros procesos sobre la región del archivo que deseamos manipular.

Para usar bloqueo obligatorio, debemos habilitarlo en el sistema de archivos que contiene a los archivos que deseamos bloquear y en cada archivo que vaya a ser bloqueado. En Linux, habilitamos el bloqueo obligatorio sobre un sistema de archivos montándolo con la opción **`-o mand`**:

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

Desde un programa, podemos obtener el mismo resultado especificando la bandera **`MS_MANDLOCK`** cuando invocamos la llamada **`mount(2)`**.

El bloqueo obligatorio sobre un archivo se habilita combinado la activación del bit *setgroupid* y desactivando el bit *group-execute*. Esta combinación de bits de permisos no tiene otro uso y por eso se le asigna este significado, lo que permite no cambiar los programas ni añadir nuevas llamadas. Desde el shell, podemos habilitar el bloqueo obligatorio como sigue:

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

Desde un programa, podemos habilitarlo ajustando los permisos adecuadamente con **`chmod()`** o **`fchmod()`**. Así, cuando mostramos los permisos de un archivo que permite el bloqueo obligatorio, veremos una S en el permiso ejecución-de-grupo:

```
%> ls -l /pruebafs/archivo
-rw-r-Sr-- 1 jagomez jagomes 0 12 Dec 14:00 /pruebafs/archivo
```

El bloqueo obligatorio es soportado por todos los sistemas de archivos nativos Linux y Unix, pero puede no ser soportado en sistemas de archivos de red o sistemas de archivos no-Unix. Por ejemplo, VFAT no soporta el bloqueo obligatorio al no disponer de bit de permiso set-group-ID.

La cuestión que se plantea ahora es qué ocurre cuando tenemos activado el bloqueo obligatorio y al realizar una operación de E/S encontramos un conflicto de cerrojos como, por ejemplo, intentar escribir en una región que tiene actualmente un cerrojo de lectura o escritura, o intentar leer de una región que tiene un cerrojo de lectura. La respuesta depende de si el archivo esta abierto de forma bloqueante o no bloqueante. Si el archivo se abrió en modo bloqueo, la llamada al sistema bloqueará al proceso. Si el archivo se abrió con la bandera **`O_NONBLOCK`**, la llamada fallará inmediatamente con el error **`EAGAIN`**. Reglas similares se aplican para **`truncate()`** y **`ftruncate()`**, si intentamos añadir o eliminar en una región que solapa con la región bloqueada. Si hemos abierto el archivo en modo bloqueo, no hemos especificado **`O_NONBLOCK`**, las llamadas de lectura o escritura pueden provocar una situación de interbloqueo. En una situación como la descrita antes, el kernel resuelve la situación aplicando el mismo criterio, selecciona a uno de los procesos involucrados en el interbloqueo y provoca que la llamada al sistema **`write()`** falle con el error **`EDEADLK`**.

Los intentos de abrir un archivo con la bandera **O_TRUNC** fallan siempre de forma inmediata, con error **EAGAIN**, si los otros procesos tienen un cerrojo de lectura o escritura sobre cualquier parte del archivo.

Los cerrojos obligatorios hacen menos por nosotros de lo que cabría esperar, y tienen algunas deficiencias:

- Mantener un cerrojo sobre un archivo no evita que otro proceso pueda borrarlo, ya que solo necesita los permisos necesarios para eliminar el enlace del directorio.

- Para habilitar cerrojos obligatorios en un archivo públicamente accesible debemos tener cierto cuidado ya que incluso procesos privilegiados no podrían sobrepasar el cerrojo. Un usuario malintencionado podría obtener continuamente un cerrojo sobre el archivo de forma que provoque un ataque de denegación de servicio.

- Existe un coste de rendimiento asociado a los cerrojos obligatorios debido a que el kernel debe comprobar en cada acceso los posibles conflictos. Si el archivo tiene numerosos cerrojos la penalización puede ser significativa.

- Los cerrojos obligatorios también incurren en un coste en el diseño de la aplicación debido a la necesidad de comprobar si cada llamada de E/S produce un error **EAGAIN** para operaciones no bloqueantes, o **EDEADLK** para operaciones bloqueantes.

En resumen, debemos evitar cerrojos obligatorios salvo que sean estrictamente necesarios.

2.3.2 El archivo /proc/locks

En Linux, podemos ver los cerrojos actualmente en uso en el sistema examinando el archivo específico de /proc/locks. Un ejemplo de su contenido se puede ver a continuación:

```
jose@linux:~> cat /proc/locks
1: POSIX ADVISORY READ 8581 08:08:2100091 128 128
2: POSIX ADVISORY READ 8581 08:08:2097547 1073741826 1073742335
3: POSIX ADVISORY READ 8581 08:08:2100089 128 128
4: POSIX ADVISORY READ 8581 08:08:2097538 1073741826 1073742335
5: POSIX ADVISORY WRITE 8581 08:08:2097524 0 EOF
6: POSIX ADVISORY WRITE 3937 08:08:2359475 0 EOF
. . .
```

Este archivo contiene información de los cerrojos creados tanto por **flock()** como por **fcntl()**. Cada línea muestra información de un cerrojo y tiene ocho campos que indican:

1. Número ordinal del cerrojo dentro del conjunto de todos los cerrojo del archivo.

2. Tipo de cerrojo, donde POSIX indica que el cerrojo se creó con **fcntl()** y **FLOCK** el que se creó con **flock()**.

3. Modo del cerrojo, bien consultivo (**ADVISORY**) bien obligatorio (**MANDATORY**).

4. El tipo de cerrojo, ya sea **WRITE** o **READ** (correspondiente a cerrojos compartidos o exclusivos por **fcntl()**).

5. El **PID** del proceso que mantiene el cerrojo.

6. Tres números separados por ":" que identifican el archivo sobre el que se mantiene el cerrojo: el número principal y secundario del dispositivo donde reside el sistema de archivos que contiene el archivo, seguido del número de inodo del archivo.

7. El byte de inicio del bloqueo. Para **`flock()`**, siempre es 0.

8. El byte final del bloqueo. Donde **`EOF`** indica que el cerrojo se extiende hasta el final del archivo. Para **`flock()`** esta columna siempre es **`EOF`**.

2.3.3 Ejecutar una única instancia de un programa

Algunos programas y, en especial, algunos demonios, deben asegurarse de que solo se ejecuta una instancia del mismo en un instante dado. El método común es que el programa cree un archivo en un directorio estándar y establezca un cerrojo de escritura sobre él.

El programa tiene el bloqueo del archivo durante toda su ejecución y lo libera justo antes de terminar. Si se inicia otra instancia del programa, fallará al obtener el cerrojo de escritura. Por tanto, supone que existe ya otra instancia del programa ejecutándose y termina. Una ubicación usual para tales archivos de bloqueos es /var/run. Alternativamente, la ubicación del archivo puede especificarse en una línea en el archivo de configuración del programa.

Por convención, los demonios escriben su propio identificador en el archivo de bloqueo, y en ocasiones el archivo se nombra con la extensión "**`.pid`**". Por ejemplo, **`syslogd`** crea un archivo **`/var/run/syslogd.pid`**. Esto es útil en algunas aplicaciones para poder encontrar el PID del demonio.

También permite hacer comprobaciones extras de validez, ya que podemos comprobar que el proceso con ese indicador existe utilizando **`kill(pid,0)`**.

Actividad 2.3 Bloqueo de archivos con la llamada al sistema `fcntl`

Ejercicio 3. Construir un programa que verifique que, efectivamente, el kernel comprueba que puede darse una situación de interbloqueo en el bloqueo de archivos.

Ejercicio 4. Construir un programa que se asegure que solo hay una instancia de él en ejecución en un momento dado. El programa, una vez que ha establecido el mecanismo para asegurar que solo una instancia se ejecuta, entrará en un bucle infinito que nos permitirá comprobar que no podemos lanzar más ejecuciones del mismo. En la construcción del mismo, deberemos asegurarnos de que el archivo a bloquear no contiene inicialmente nada escrito en una ejecución anterior que pudo quedar por una caída del sistema.

3. Archivos proyectados en memoria con `mmap()`

Un archivo proyectado en memoria es una técnica que utilizan los sistemas operativos actuales para acceder a archivos. En lugar de una lectura "tradicional" lo que se hace es crear una nueva región de memoria en el espacio de direcciones del proceso del tamaño de la zona a acceder del archivo (una parte o todo el archivo) y cargar en ella el contenido de esa parte del archivo. Las páginas de la proyección son (automáticamente) cargadas del archivo cuando sean necesarias.

La función **`mmap()`** proyecta bien un archivo bien un objeto memoria compartida en el espacio de direcciones del proceso. Podemos usarla para tres propósitos:

- Con un archivo regular para suministrar E/S proyectadas en memoria (este Apartado).
- Con archivo especiales para suministrar proyecciones anónimas (Apartado 3.2).
- Con **shm_open** para compartir memoria entre procesos no relacionados (no lo veremos dado que no hemos visto segmentos de memoria compartida, que es uno de los mecanismos denominados *System V* IPC).

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

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

      Retorna: dirección inicial de la proyección,si OK; MAP_FAILED, si error.
```

El primer argumento de la función, **address**, especifica la dirección de inicio dentro del proceso donde debe proyectarse (mapearse) el descriptor. Normalmente, especificaremos un nulo que le indica al kernel que elija la dirección de inicio. En este caso, la función retorna como valor la dirección de inicio asignada. El argumento **len** es el número de bytes a proyectar, empezando con un desplazamiento desde el inicio del archivo dado por **offset**. Normalmente, **offset** es 0. La Figura 3 muestra la proyección. El argumento **fd** indica el descriptor del archivo a proyectar, y que una vez creada la proyección, podemos cerrar.

Dado que la página es la unidad mínima de gestión de memoria, la llamada **mmap()** opera sobre páginas: el área proyectada es múltiplo del tamaño de página. Por tanto, **address** y **offset** deberían estar alineados a limite de páginas, es decir, deberían ser enteros múltiples del tamaño de página. Si el parámetro **len** no está alineado a página (porque, por ejemplo, el archivo subyacente no tiene un tamaño múltiplo de página), la proyección se redondea por exceso hasta completar última página. Los bytes añadidos para completar la página se rellenan a cero: cualquier lectura de los mismos retornará ceros, y cualquier escritura de ésta memoria no afectará al almacén subyacente, incluso si es **MAP_SHARED** (volveremos sobre esto n el apartado 4).

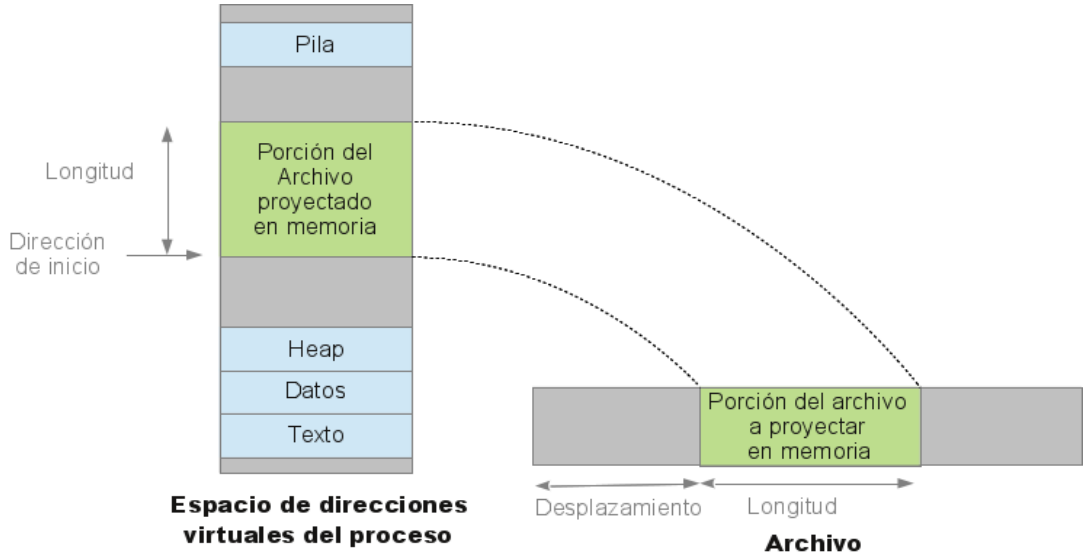


Figura 3.- Visión lógica de una proyección en memoria de un archivo.

El tipo de protección de la proyección viene dado por la máscara de bits **`prot`** y para ello usamos las constantes indicadas en la Tabla 1. Los valores comunes para la protección son **`PROT_READ | PROT_WRITE`**.

Tabla 1.- Valores de protección de una proyección.

Valor `prot`	Descripción
`PROT_READ`	Los datos se pueden leer.
`PROT_WRITE`	Los datos se pueden escribir.
`PROT_EXEC`	Podemos ejecutar los datos
`PROT_NONE`	No podemos acceder a los datos

Los valores del argumento **`flags`** se indican en la Tabla 2. Debemos especificar el indicador **`MAP_SHARED`** o **`MAP_PRIVATE`**. Opcionalmente se puede hacer un **`OR`** con **`MAP_FIXED`**. El significado detallado de algunos de estos indicadores es:

`MAP_PRIVATE` Las modificaciones de los datos proyectados por el proceso son visibles solo para ese proceso y no modifican el objeto subyacente de la proyección (sea un archivo o memoria compartida). El uso principal de este tipo de proyección es que múltiples procesos compartan el código/datos de un ejecutable o biblioteca[4], de forma que las modificaciones que realicen no se guarden en el archivo.

`MAP_SHARED` Las modificaciones de los datos de la proyección son visibles a todos los procesos que la comparten y estos cambios modifican el objeto subyacente. Los dos usos principales son bien realizar entradas/salidas proyectadas en memoria, bien compartir memoria entre procesos.

`MAP_FIXED` Instruye a **`mmap()`** que la dirección **`address`** es un requisito, no un consejo. La llamada fallará si el kernel es incapaz de situar la proyección en la dirección indicada. Si la dirección solapa una proyección existente, las páginas solapadas se descartan y se sustituyen por la nueva proyección. No debería especificarse por razones de portabilidad ya que requiere un conocimiento interno del espacio de direcciones del proceso.

Tabla 2.- Indicadores de una proyección.

Valor `flags`	Descripción
`MAP_SHARED`	Los cambios son compartidos
`MAP_PRIVATE`	Los cambios son privados
`MAP_FIXED`	Interpreta exactamente el argumento **`address`**
`MAP_ANONYMOUS`	Crea un mapeo anónimo (Apartado 3.2)
`MAP_LOCKED`	Bloquea las páginas en memoria (al estilo **`mlock`**)
`MAP_NORESERVE`	Controla la reserva de espacio de intercambio

4 Los sistemas operativos actuales construyen los espacios de direcciones de los procesos proyectando en memoria las regiones de código y datos del archivo ejecutable, así como de las bibliotecas.

MAP_POPULATE	Realiza una lectura adelantada del contenido del archivo
MAP_UNITIALIZED	No limpia(poner a cero) las proyecciones anónimas

A continuación, veremos un ejemplo de cómo crear una proyección. El programa **Tarea14.c** crea un archivo denominado Archivo y los rellena con nulos. Tras lo cual, crea una proyección compartida del archivo para que los cambios se mantengan. Una vez establecida la proyección, copia en la memoria asignada a la misma el mensaje "**Hola Mundo\n**". Tras finalizar el programa, podemos visualizar el archivo para ver cual es el contenido: la cadena "**Hola Mundo\n**".

Tarea 14.c.- Ejemplo de creación de una proyección.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

const int MMAPSIZE=1024;
int main()
{
  int fd, num;
  char ch='\0';
  char *memoria;
  fd = open("Archivo", O_RDWR|O_CREAT|O_EXCL, S_IRWXU);
  if (fd == -1) {
    perror("El archivo existe");
    exit(1);
  }
 for (int i=0; i < MMAPSIZE; i++){
      num=write(fd, &ch, sizeof(char));
      if (num!=1) printf("Error escritura\n");
  }
  memoria = (char *)mmap(0, MMAPSIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
  if (memoria == MAP_FAILED) {
      perror("Fallo la proyeccion");
      exit(2);
  }
  close(fd);  /* no es necesario el descriptor*/
  strcpy(memoria, "Hola Mundo\n"); /* copia la cadena en la proyección */
  exit(0);

}
```

Como hemos podido ver en el ejemplo anterior, los dos pasos básicos para realizar la proyección son:

1. Obtener el descriptor del archivo con los permisos apropiados dependientes del tipo de proyección a realizar, normalmente vía **open()**.

2. Pasar este descriptor de archivo a la llamada **mmap()**.

En el programa **Tarea15.c**, ilustramos otro ejemplo de la función **mmap()** que utilizamos en este caso tanto para visualizar un archivo completo, que pasamos como primer argumento (similar a la orden **cat**), como mostrar el contenido del byte cuyo desplazamiento se pasa

como segundo argumento. Mostramos estos dos aspectos para ilustrar cómo, una vez establecida la proyección, podemos acceder a los datos que contiene con cualquier mecanismo para leer de memoria.

Tarea15.c.- Visualizar un archivo con mmap().

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
int main (int argc, char *argv[])
{
       struct stat sb;
       off_t len;
       char *p;
       int fd;
       if (argc < 2) {
              printf("Uso: %s archivo\n”, argv[0]);
              return 1;    }
       fd = open (argv[1], O_RDONLY);
       if (fd == -1) {
              printf("Error al abrir archivo\n”);
              return 1;    }
       if (fstat (fd, &sb) == -1) {
              printf("Error al hacer stat\n");
              return 1;    }
       if (!S_ISREG (sb.st_mode)) {
              printf(“%s no es un archivo regular\n",argv[1]);
              return 1;    }
       p = (char *) mmap (0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
       if (p == MAP_FAILED) {
              printf("Fallo el mapeo\n");
              return 1; }
       if (close (fd) == -1) {
              printf("Error al cerrar el archivo\n");
              return 1;
       }
              /* Mostramos el archivo completo */
              printf(%s\n”, p);
              /* Mostramos en byte con desplazamiento argv[2]*/
              printf(”Byte con desplamiento %s es %d: “, argv[2], p[atoi(argv[2])]);
       if (munmap (p, sb.st_size) == -1) {
              printf("Error al cerrar la proyeccion\n");
              return 1;    }
       return 0;
}
```

Para eliminar una proyección del espacio de un proceso, invocaremos a la función:

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

int munmap(void *address, size_t length);

                                Retorna: 0,si OK; -1, si error.
```

El argumento **address** es la dirección que retornó la llamada **mmap()** para esa proyección, y **len** es el tamaño de la región mapeada. Una vez desmapeada, cualquier referencia a la proyección generará la señal **SIGSEGV**. Si una región se declaró **MAP_PRIVATE**, los cambios que se realizaron en ella son descartados.

Durante la proyección, el kernel mantiene sincronizada la región mapeada con el archivo (normalmente en disco) suponiendo que se declaró **`MAP_SHARED`**. Es decir, si modificamos un dato de la región mapeada, el kernel actualizará en algún instante posterior el archivo. Pero si deseamos que la sincronización sea inmediata debemos utilizar **`msync()`**.

Pero ¿por qué utilizar **`mmap()`**? La utilización del mecanismo de proyección de archivos tiene algunas ventajas:

- Desde el punto de vista del programador, simplificamos el código, ya que tras abrir el archivo y establecer el mapeo, no necesitamos realizar operaciones **`read()`** o **`write()`**.

- Es más eficiente, especialmente con archivos grandes, ya que no necesitamos copias extras de la información desde del kernel al espacio de usuario y a la inversa. Además, una vez creada la proyección no incurrimos en el coste de llamadas al sistema extras, ya que solo accedemos a memoria.

- Cuando varios procesos proyectan un mismo objeto en memoria, solo hay una copia de él compartida por todos ellos. Las proyecciones de solo-lectura o escritura-compartida son compartidas en cada proceso, las proyecciones de escritura comparten las páginas mediante el *mecanismo COW* (*copy-on-write*).

- La búsqueda de datos en la proyección involucra la manipulación de punteros, por lo que no hay necesidad de utilizar **`lseek()`**.

No obstante, hay que mantener presentes algunas consideraciones al utilizar la llamada:

- El mapeo de un archivo debe encajar en el espacio de direcciones de un proceso. Con un espacio de direcciones de 32 bits, si hacemos numerosos mapeos de diferentes tamaños, fragmentamos el espacio y podemos hacer que sea difícil encontrar una región libre continua.

- La atomicidad de las operaciones es diferente. La atomicidad de las operaciones en una proyección viene determinada por la atomicidad de la memoria, es decir, la celda de memoria. Con las operaciones **`read()`** o **`write()`** la atomicidad en la modificación de datos viene determinada por el tamaño del búfer que especifiquemos en la operación.

- La visibilidad de los cambios es diferente. Si dos procesos comparten una proyección, la modificación de datos de la proyección por parte de un proceso es instantánemente vista por el otro proceso. Cuando utilizamos la interfaz clásica **`read()`**/**`write()`**, si un proceso lee un dato de un archivo y posteriormente otro proceso hace una escritura para modificarlo, el primer proceso solo verá la modificación si vuelve a realizar otra lectura.

- La diferencia entre el tamaño del archivo proyectado y el número de páginas utilizadas en la proyección es espacio “desperdiciado”. Para archivos pequeños la porción de espacio desperdiciado es más significativa que para archivos grandes. En muchos casos, este posible desperdicio de espacio se compensa no debiendo tener múltiples copias de la información en memoria.

- No todos los archivos pueden ser proyectados. Si intentamos proyectar un descriptor que referencia a un terminal o a un socket, la llamada genera error. Estos descriptores deben accederse mediante **`read()`** o **`write()`** o variantes.

Otras funciones relacionadas con la protección de archivos son:

`mremap()`: se utiliza para extender una proyección existente.

`mprotect()`: cambia la protección de una proyección.

`madvise()`: establece consejos sobre el uso de memoria, es decir, como manejar las entradas/salidas de páginas de una proyección.

`remap_file_pages()`: permite crear mapeos no-lineales, es decir, mapeos donde las páginas del archivo aparecen en un orden diferente dentro de la memoria contigua.

`mlock()`: permite bloquear (anclar) páginas en memoria.

`mincore()`: informa sobre las páginas que están actualmente en RAM

3.1 Compartición de memoria

Una forma de compartir memoria entre un proceso padre y un hijo es invocar **`mmap()`** con **`MAP_SHARED`** en el padre antes de invocar a **`fork()`**. El estándar POSIX.1 garantiza que la proyección creada por el padre se mantiene en el hijo. Es más, los cambios realizados por un proceso son visibles en el otro.

Para mostrarlo, en el programa **`Tarea16.c`** construimos un ejemplo donde un proceso padre crea una proyección que se utiliza para almacenar un valor. El padre asignará valor a **`cnt`** y el hijo solo leerá el valor asignado por el padre (lo hacemos así -solo lectura en el hijo- para evitar condiciones de carrera y así evitarnos tener que introducir un mecanismo de sincronización para acceder al contador). También podemos ver cómo el archivo contiene el valor modificado del padre.

Tarea16.c. Compartición de memoria entre procesos padre-hijo con **`mmap()`**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

#define MMAPSIZE 1

int main (int argc, char *argv[])
{
    off_t len;
    char bufer='a';
    char *cnt;
    int fd;

    fd = open("Archivo", O_RDWR|O_CREAT|O_TRUNC, S_IRWXU);
    if (fd == -1) {
    perror("El archivo existe");
    exit(1);
    }
    write(fd, &bufer, sizeof(char));

    cnt = (char *) mmap (0, MMAPSIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (cnt == MAP_FAILED) {
```

```
       printf("Fallo el mapeo\n");
       return 1;
    }
    if (close (fd) == -1) {
      printf("Error al cerrar el archivo\n");
      return 1;   }

if (fork() == 0) { /* hijo */
  sleep(2);
  printf("El valor de cnt es: %d", cnt);
  exit(0);
  }
  /* padre */
  strcpy(cnt, "b");
  exit(0);
}
```

La compartición de memoria no está restringida a procesos emparentados, cualesquiera procesos que mapeen la misma región de un archivo, comparten las mismas páginas de memoria física. El que estos procesos vean o no las modificaciones realizadas por otros dependerá de que el mapeo sea compartido o privado.

3.2 Proyecciones anónimas

Una *proyección anónima* es similar una proyección de archivo salvo que no existe el correspondiente archivo de respaldo. En su lugar, las páginas de la proyección son inicializadas a cero. La Tabla 4 resumen los propósitos de los diferentes tipos de proyecciones.

Tabla 4.- Propósitos de los diferentes tipos de proyecciones en memoria.

Visibilidad de las modificaciones	Tipo de proyección	
	Archivo	Anónimo
Privado	Inicializa memoria con el contenido del archivo	Asignación de memoria
Compartido	E/S proyectadas en memoria Compartición de memoria (IPC)	Compartición de memoria (IPC)

El ejemplo mostrado en el Programa 3 funciona correctamente pero nos ha obligado a crear un archivo en el sistema de archivos (si no existía ya) y a inicializarlo. Cuando utilizamos una proyección para ser compartida entre procesos padre e hijo, podemos simplificar el escenario de varias formas, dependiendo del sistema:

- Algunos sistemas suministran las proyecciones anónimas que nos evitan tener que crear y abrir un archivo. En su lugar, podemos utilizar el indicador **`MAP_ANON`** (o **`MAP_ANONYMOUS`**) en **`mmap()`** el valor -1 para **`fd`** (el **`offset`** se ignora). La memoria se inicializa a cero. Esta forma está en desuso (en Linux se implementa a través de */dev/zero*) y la podemos ver en el programa **`Tarea17.c`**.

- La mayoría de los sistemas suministran el seudo-dispositivo **`/dev/zero`** que tras abrirlo y leerlo suministra tantos ceros como le indiquemos, y cualquier cosa que escribamos en él es descartada. Su uso se ilustra en el programa **`Tarea18.c`**.

`Tarea17.c`.- Proyección anónima con **`MAP_ANON`**.

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main (int argc, char *argv[])
{
    char *p;

    p = (char *)mmap (0, sizeof(char), PROT_READ , MAP_SHARED |MAP_ANON, -1, 0);
    if (p == MAP_FAILED) {
             printf("Fallo el mapeo\n");
             return 1;
       }
       close(fd);
      return 0;
}
```

Tarea18.c.- Proyección anónima con **`/dev/zero`**.

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main (int argc, char *argv[])
{
  int fd;
  char *p;

  fd = open("/dev/zero", O_RDONLY);

  p = (char *) mmap (0, sizeof(int), PROT_READ , MAP_SHARED, fd, 0);
  if (p == MAP_FAILED) {
       printf("Fallo el mapeo\n");
       return 1;
  }
  close(fd);
  return 0;
}
```

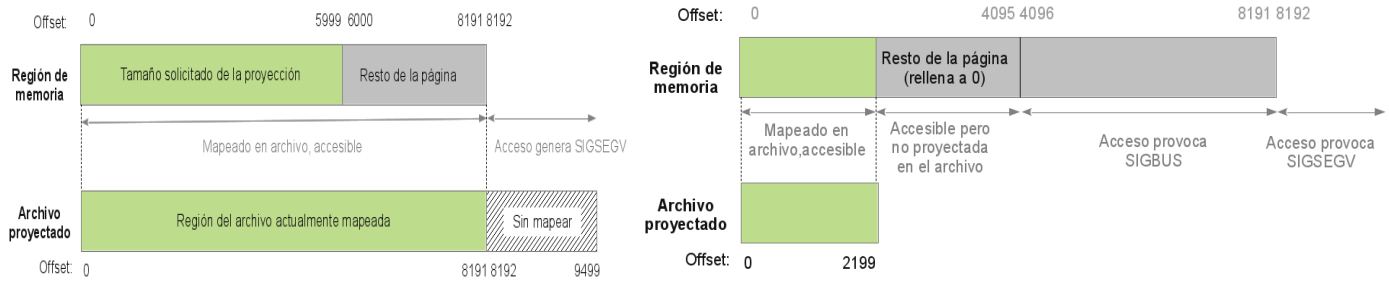
Si unimos este hecho a los vistos en el Apartado 2, podemos deducir que una de las utilidades de las proyecciones anónimas es que varios procesos emparentados compartan memoria. También se puede utilizar como mecanismo de reserva y asignación de memoria en un proceso.

3.3 Tamaño de la proyección y del archivo proyectado

En muchos casos, el tamaño del mapeo es múltiplo del tamaño de página, y cae completamente dentro de los límites del archivo proyectado. Sin embargo, no es necesariamente así, por ello vamos a ver que ocurre cuando no se dan esas condiciones.

La Figura 4a describe el caso en el que la proyección está dentro de los límites del archivo proyectado pero el tamaño de la región no es múltiplo del tamaño de página del sistema. Suponemos en el sistema tiene páginas de 4KiB y que realizamos una proyección de 5999 B. En este caso, la región proyectada ocupará 2 páginas de memoria si bien la proyección del

archivo es menor. Cualquier intento de acceder a la zona redondeada hasta el límite de página provocará la generación de la señal `SIGSEGV`.



(a) (b)

Figura 4.- Proyección con tamaño (a) no múltiplo del tamaño de página, (b) mayor que el archivo de respaldo.

Cuando la proyección se extiende más allá del fin de archivo (Figura 4b), la situación es algo más compleja. Como antes, si el tamaño de la proyección no es múltiplo del tamaño de página, ésta se redondea por exceso. En la Figura 4, creamos una proyección de 8192 B de un archivo que tiene 2199 B. En éste los bytes de redondeo de la proyección hasta completar una página (bytes 2199 al 4095, en el ejemplo), son accesibles pero no se mapean en el archivo de respaldo, y se inicializan a 0. Estos bytes nunca son compartidos con la proyección del archivo con otro proceso. Su modificación no se almacena en el archivo.

Si la proyección incluye páginas por encima de la página redondeada por exceso, cualquier intento de acceder esta zona generará la señal **`SIGBUS`**. En nuestro ejemplo, los bytes comprendidos entre la dirección 4096 y la 8192. Cualquier intento de acceder por encima de la dirección 8191, generará la señal **`SIGSEGV`**.

El programa **`Tarea19.c`** muestra la forma habitual de manejar un archivo que esta creciendo: especifica una proyección mayor que el archivo, tiene en cuenta del tamaño actual del archivo (asegurándonos no hacer referencias a posiciones posteriores al fin de archivo), y deja que que se incremente el tamaño del archivo conforme se escribe en él.

`Tarea19.c`.- Proyección que deja crecer un archivo.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

#define FILE "datos"
#define SIZE 32768

int main(int argc, char **argv)
{
       int    fd, i;
       char   *ptr;

       fd = open(FILE, O_RDWR|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
       ptr = (char*)mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

       for (i = 4096; i<= SIZE; i += 4096) {
         printf("Ajustando tamaño archivo a %d, i);
         ftruncate(fd, i);
         printf("ptr["<< i-1 <<"] = ", <<ptr[i - 1])endl;
```

```
        }
        exit(0);
}
```

En el Programa, hemos utilizado la función **ftruncate()** que trunca el archivo indicado por **fd** al tamaño indicado como segundo argumento. Si el tamaño indicado es mayor que el tamaño del archivo, su contenido se rellena con nulos.

La ejecución del programa se muestra a continuación. Además, podemos ver como el tamaño del archivo efectivamente se ha modificado.

```
~/tmp> ./m7
Ajustando tamaño archivo a 4096
ptr[4095] =
Ajustando tamaño archivo a 8192
ptr[8191] =
. . .
Ajustando tamaño archivo a 32768
ptr[32767] =
~/tmp> ls -l datos
-rw-r--r-- 1 jose users 32768 ene 14 18:31 datos
```

4 Actividades

Ejercicio 5: Escribir un programa, similar a la orden **cp**, que utilice para su implementación la llamada al sistema **mmap()** y una función de C que nos permite copiar memoria, como por ejemplo **memcpy()**. Para conocer el tamaño del archivo origen podemos utilizar **stat()** y para establecer el tamaño del archivo destino se puede usar **ftruncate()**.