FILEDES.C

Lock

Germán Jerez Cárdenes
Miguel Ángel Mederos Sánchez

El fichero *filedes* contiene tres funciones que actúan sobre los descriptores de archivos. En el desarrollo del siguiente texto se hace mención a los problemas de almacenar la posición de cada proceso trabajando sobre un archivo común a varios. Entenderemos por proceso los programas que quieren leer o escribir de zonas de memoria que llamaremos archivos. Esta dificultad obliga a la introducción de unas tablas adicionales conocidas como **FILP.**

Cada vez que un archivo se abre se le asigna un descriptor de archivo. Este tiene un número que lo identifica. Este número debe ser pasado por el File System al proceso que abrió el archivo. Este identificador será usado en las siguientes llamadas write o read. del proceso sobre el archivo. El FS al igual que el Kernel y el MM conserva parte de la tabla de procesos en su espacio de direcciones. De esta tabla haremos particular referencia a tres campos. Los dos primeros son apuntadores a los **nodos i** del directorio raíz y del directorio de trabajo para poder acceder a los archivos de manera absoluta o relativa respectivamente (Los nodos i son records que contienen datos sobre subdirectorios o sobre archivos, además de indicarnos en qué bloque se encuentran).

El tercer campo es un array con el que se indexa el archivo que se está utilizando en base al descriptor de archivo que tenemos.

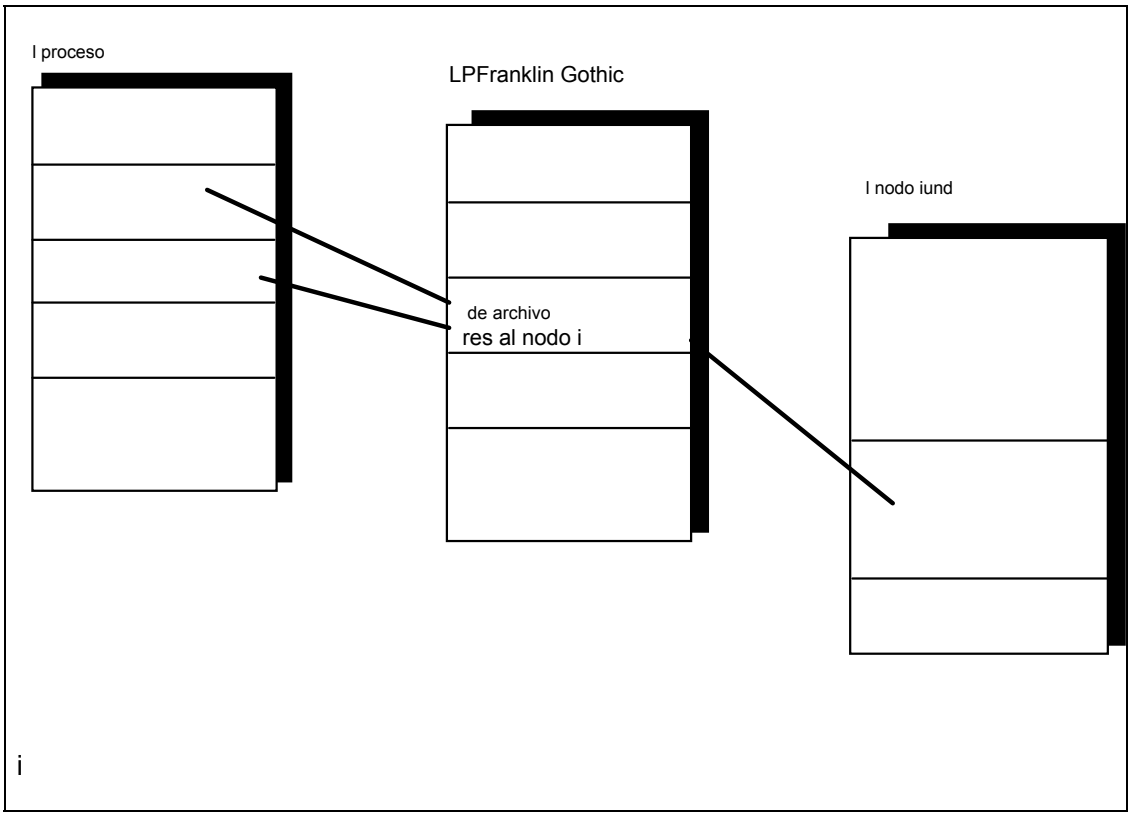
Cada proceso tiene una dirección de 32 bits llamada **posición del archivo** que indica la siguiente posición de lectura o escritura. Los problemas van a surgir cuando debamos almacenar este dato. Lo que parece más simple es disponer de una tabla compartida por todos los procesos donde la entrada K-ésima nos diga la posición del archivo con descriptor k.

Esta primera opción tiene el problema de que el minix permite la compartición de archivos por varios procesos. Así, sea un proceso p1 que acaba de trabajar en la posición 32 y pasa a otra tarea. Si después llega un proceso p2 a trabajar en el mismo archivo modificará la posición donde debe seguir trabajando p1 (cada proceso trabaja normalmente en una zona diferente del archivo).

La otra disponibilidad que en principio tenemos es que cada proceso disponga de su propia posición del archivo almacenada en su tabla de procesos. Esta solución presenta sólo un problema que la invalida. El problema proviene de la semántica de la llamada FORK ya que esta exige que cuando un proceso se ramifique, tanto el padre como el hijo compartan un solo apuntador que dé la posición actual de cada archivo abierto.

Veamos un ejemplo de lo dicho. Cuando el shell se ramifica produciendo el primer programa, su posición de archivo en la salida estandard es cero. Después esta posición es heredada por el hijo, el cual escribe, por ejemplo, 1 k de salida. Cuando el hijo termina, la posición del archivo debe ser 1k. Ahora el shell lee algo más de la escritura del shell y produce otro hijo. Es esencial que el segundo hijo comience a escribir en el sitio donde se quedó el primer programa. Si el shell no compartiera la posición del archivo con sus hijo,

el segundo programa escribiría sobre la salida del primero en lugar de anexarla a ésta. Como resultado no es posible colocar la posición del archivo en la tabla de procesos.



La solución final que adopta minix es el uso de una nueva tabla, la tabla FILP. Cada archivo abierto tiene tantas entradas ocupadas en la tabla FILP como procesos (que no sean hijos) estén leyendo o escribiendo en el. Cada entrada dispone de un contador que indica el número de procesos que la apuntan (= número de procesos utilizando el archivo), con el cual el FS puede saber cuando el archivo ya no está siendo utilizado por ningún proceso y puede reutilizar esa entrada para otro archivo.

En cada entrada está la posición del archivo donde está trabajando el proceso correspondiente. En cada tabla de proceso sólo se almacena la dirección de la entrada en la tabla FILP para cada archivo que esté utilizando ese proceso.

Clarifiquemos esto con un ejemplo. Si un proceso P1 está trabajando con tres archivos A1, A2, A3 el tercer campo de la tabla de proceso es un array con tres entradas ocupadas. Cada entrada es un apuntador a la tabla FILP donde guardamos la posición del archivo en el que estamos trabajando.

Para solucionar el problema planteado anteriormente por la ejecución de un FORK, tanto el padre como todos sus procesos hijos comparten la misma entrada de la tabla FILP. Aunque lo único que la tabla FILP debe contener realmente es la posición del archivo compartido, conviene colocar también ahí el apuntador del nodo i. De esta forma todo lo que contiene el array indexado por el descriptor del archivo en la tabla del proceso es un apuntador a la tabla FILP.

La estructura 'filp' viene definida en el fichero "file.h". Es un intermediario entre los descriptores de ficheros y los inodes. Se considera que un slot está libre cuando filp_count = 0.

```
EXTERN struct filp {
  mode_t filp_mode;    /* bits RW, que indican como se abre el archivo */
  int filp_flags;      /* flags de open y de fcntl */
  int filp_count;      /* Indica cuantos descriptores de archivo comparten el slot*/
  struct inode *filp_ino; /* puntero al inode */
  off_t filp_pos;      /* posición del fichero */
} filp[NR_FILPS];

#define FILP_CLOSED        0       /* filp_mode: asociado al dispositivo */

#define NIL_FILP (struct filp *) 0      /* indica la ausencia de un slot filp */
```

A continuación se describirán las tres funciones desarrolladas en filedes.

GET_FD

Esta función se encarga de encontrar un descriptor de archivo y una entrada en la tabla FILP libres, cuando un proceso ha hecho una peticiín de abrir un fichero. Si no se encuentra alguna de las dos devuelve un error. En casos de que la resolución resulte exitosa, devuelve un puntero al número de descriptor del archivo y un puntero a la entrad de la tabla FILP libre.

Parámetros que utiliza:

start : Entero que indica en qué posición de la tabla de procesos empezar a buscar un descriptor de fichero libre.

bits : Indica el modo de apertura del fichero.

k : Puntero a la posición donde se devuelve el descriptor de ficheros.

fpt : Puntero a la posición donde se devuelve la dirección de la entrada libre en la tabla FILP.

PROGRAMA COMENTADO

```
#include "fs.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
/*-----------------------------------------------------------------------------------------*/
/*                                    get_fd                                               */
/*-----------------------------------------------------------------------------------------*/

PUBLIC int get_fd(start, bits, k, fpt)
int start;
mode_t bits;
int *k;
struct filp **fpt;
{
/* Buscan un descriptor de archivo lobre y una entrad libre en la tabla FILP. Al final del
programa se rellena la palabra de modo */
    register struct filp *f;
    register int i;

    *k = -1         /* se pone negativo para detectar  posteriormente si se ha encontrado el
                    descriptor de archivo libre */

    /* Busca en la tabla de procesos un descriptor sde archivos libre empezando desde la
    posición start. */
    for (i=start; i < OPEN_MAX; i++ ) {
        if (fp->fp_filp[i] == NIL_FILP) {
            /* Se ha localizado un descriptor de archivo libre . */
            *k = i;
            break;
        }
    }

    /* Chequea si se ha encontrado un descriptor de archivo. */
    if (*k < 0 ) return(EMFILE); /* debido a que inicializamos k a -1 */

    /* El descriptor de archivo ha sido encontrado, buscamos ahora una entrada libre en la
tabla FILP. */
    for (f = &filp[0] ; f < &filp[NR_FILPS]; f++) {
        if (f->filp.count == 0 ) {
            f->filp_mode = bits;
            f->filp_pos = 0L;
            f-> filp_flags = 0;
            *fpt = f;
            return(OK);
        }
    }
```

```
    /* Si el programa llega hasta aquí significa que la tabla filp está llena y retorna este
error. */
    return(ENFILE);
}
```

GET_FILP

Nos devuelve la entrada en la tabla filp para un descriptor de fichero dado.

Parámetros de entrada:

fild : entero usado como descriptor del fichero.

PROGRAMA COMENTADO

```
/*-------------------------------------------------------------------------------------*/
/*                                    get_filp                                         */
/*-------------------------------------------------------------------------------------*/

Public struct filp *get_filp(fild)
int fild;
{/* Mira si el parametro fild que nos pasan es válido. En caso contrario devuelve su
puntero a la entrada correspondiente en la tabla filp. */

    err_code = EBADF;
    if (fild < 0 || fild >= OPEN_MAX ) return (NIL_FILP);
    return(fp->fp_filp[fild]);   /* puede ser también NIL_FILP */
}
```

FIND_FILD

Esta función consiste en encontrar una entrada en la tabla filp de modo que se refiera al nodo i indicado por rip y en el modo que se describe en la variable bits. Esta función se usa para determinar si todavía hay alguien interesado en el final de un pipe. Al igual que el get_fd realiza su trabajo con búsqueda lineal a través de la tabla filp.

Parámetros de entrada:

rip: Nodo i referido por el filp que se obtendrá.
bits: modo del filp que se obtendrá (RWX).

PROGRAMA COMENTADO

```
/*----------------------------------------------------------------------------------------*/
/*                                    find_filp                                           */
/*----------------------------------------------------------------------------------------*/

PUBLIC struct filp *find_filp(rip, bits)
register struct inode *rip;
int bits;
{

    register struct filp *f;

    for ( f = &filp[0]; f < &filp[NR_FILPS]; f++) {
        if (f->filp_count != 0 && f->filp_ino == rip && ( f->filp_mode & bits)){
            return(f);
        }
    }

    /* Si el programa llega a esta punto el filp buscado no se encuentra en la tabla y devuelve
    este hecho. */

    return( NIL_FILP);
}
```

LOCK.C

Las funciones de bloqueo de POSIX son:

Operación	Significado
F_SETLK	Bloquea la región para lectura y escritura
F_SETLKW	Bloquea la región para escritura
F_GETLK	Indica si la región está bloqueada

Se puede bloquear una parte de un fichero para la lectura y la escritura, o sólo para la escritura con una llamada de FCNTL indicando F_SETLK o F_SETLKW. Sólo existen dos funciones en el fichero "lock.c". Se llama a la función Lock_op con dos parámetros; el primero es un puntero a la estructura filp del proceso y el segundo parámetro indicará la operación a realizar (cuadro anterior). Cuando se bloquea una región, esta no debe tener conflicto con ninguna de las regiones bloqueadas actualmente, y cuando se libera el bloqueo, el bloqueo existente no debe dividirse en dos. Cuando se libera cualquier bloqueo, se llama a la otra función del fichero "lock_revive". Esta despierta a todos los procesos que están esperando para ser bloqueados. Esta estrategia es un compromiso, ya que genera código extra para indicar que procesos exáctamente están esperando para liberar un bloqueo en particular. Esos procesos que siguen esperando por un fichero bloqueado, se bloquearán otra vez cuando se ejecuten. Esta estrategia se basa en la presunsión de que el bloqueo se usará poco. Si se construyera una base de datos multiusuario más grande en MINIX, sería recomendable reimplementar esto. A la función "lock_revive" también se la llama cuando se cierra un fichero bloqueado, como podría ocurrir, por ejemplo, cuando se mata un proceso antes de que este termine de usar un fichero bloqueado.

La estructura file_lock, que se encuentra en el fichero "lock.h". Al igual que la estructura 'filp', apunta a la tabla de inodes, aunque en este caso, para almacenar el indicador de bloqueo.

```
EXTERN struct file_lock {
  short lock_type;              /* F_RDLOCK o F_WRLOCK; 0 indica slot libre */
  pid_t lock_pid;               /* pid del proceso que tiene el bloqueo */
  struct inode *lock_inode;     /* puntero al inode bloqueado */
  off_t lock_first;             /* posición del primer byte bloqueado */
  off_t lock_last;              /* posición del último byte bloqueado */
} file_lock[NR_LOCKS];
```

Otra de las estructuras utilizadas es la estructura flock, que se encuentra en "fcntl.h"

```
struct flock {
  short l_type;                 /* tipos: F_RDLCK, F_WRLCK, o F_UNLCK */
  short l_whence;               /* flag para el desplazamiento inicial */
  off_t l_start;          /* desplazamiento relativo en bytes */
  off_t l_len;                  /* tamaño; si es 0, entonces hasta EOF */
  pid_t l_pid;                  /* id del proceso del poseedor del bloqueo */
};
```

El código comentado del fichero sería:

```
/* Este fichero es el encargado del manejo del bloqueo requerido por POSIX
 *
 * Los puntos de entrada a este fichero son:
 *   lock_op:    realiza las operaciones de bloqueo ante una llamada del sistema FCNTL
 *   lock_revive: revive los procesos cuando se desbloquea
 */
#include "fs.h"
#include <fcntl.h>
#include <unistd.h>
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "lock.h"
#include "param.h"


/*===========================================================================*
 *                                lock_op                                    *
 *===========================================================================*/
PUBLIC int lock_op(f, req)
struct filp *f;
int req;                    /* puede ser F_SETLK o F_SETLKW  o GETLK*/
{
/* Encargado del manejo del bloqueo requerido por POSIX. */

  int r, ltype, i, conflict = 0, unlocking = 0;
  mode_t mo;
  off_t first, last;
  struct flock flock;
  vir_bytes user_flock;
  struct file_lock *flp, *flp2, *empty;

  /* Busca la estructura flock del espacio del usuario*/
  user_flock = (vir_bytes) name1;
  r = sys_copy(who, D, (phys_bytes) user_flock,
          FS_PROC_NR, D, (phys_bytes) &flock, (phys_bytes) sizeof(flock));
  if (r != OK) return(EINVAL);

  /* Se chequean los errores */
  ltype = flock.l_type; /* puede ser F_RDLCK, F_WRLCK o F_UNLCK */
  mo = f->filp_mode;
  if (ltype != F_UNLCK && ltype != F_RDLCK && ltype != F_WRLCK)
return(EINVAL);
  if (req == F_GETLK && ltype == F_UNLCK) return(EINVAL);
  if ( (f->filp_ino->i_mode & I_TYPE) != I_REGULAR) return(EINVAL);
```

```
if (req != F_GETLK && ltype == F_RDLCK && (mo & R_BIT) == 0) return(EBADF);
if (req != F_GETLK && ltype == F_WRLCK && (mo & W_BIT) == 0) return(EBADF);
/* Se calculan el primer y el último byte en la region bloqueada */
switch (flock.l_whence) {
        case SEEK_SET:      first = 0; break;
        case SEEK_CUR:      first = f->filp_pos; break;
        case SEEK_END:      first = f->filp_ino->i_size; break;
        default:            return(EINVAL);
}
/* Se comprueba el overflow. */
if (((long)flock.l_start > 0) && ((first + flock.l_start) < first))
        return(EINVAL);
if (((long)flock.l_start < 0) && ((first + flock.l_start) > first))
        return(EINVAL);
first = first + flock.l_start;
last = first + flock.l_len - 1;
if (flock.l_len == 0) last = MAX_FILE_POS;
if (last < first) return(EINVAL);

/* Se comprueba si la región coincide con cualquier región de bloqueo existente */
empty = (struct file_lock *) 0;
for (flp = &file_lock[0]; flp < & file_lock[NR_LOCKS]; flp++) {
        if (flp->lock_type == 0) {
                if (empty == (struct file_lock *) 0) empty = flp;
                continue;       /* 0 significa slot sin usar */
        }
        if (flp->lock_inode != f->filp_ino) continue; /* fichero diferente */
        if (last < flp->lock_first) continue;     /* el nuevo está por delante */
        if (first > flp->lock_last) continue;     /* el nuevo está por detrás */
        if (ltype == F_RDLCK && flp->lock_type == F_RDLCK) continue;
        if (ltype != F_UNLCK && flp->lock_pid == fp->fp_pid) continue;

        /* Podría haber un conflicto. Procesarlo */
        conflict = 1;
        if (req == F_GETLK) break;

        /* Si tratamos de hacer un bloqueo, y falla*/
        if (ltype == F_RDLCK || ltype == F_WRLCK) {
                if (req == F_SETLK) {
                        /* Para F_SETLK, solo devuelve el fallo */
                        return(EAGAIN);
                } else {
                        /* Para F_SETLKW, suspende el proceso */
                        suspend(XLOCK);
                        return(0);
                }
        }
```

```
        /* Estamos quitando un bloqueo y encontramos algo que se solapa*/
        unlocking = 1;
        if (first <= flp->lock_first && last >= flp->lock_last) {
                flp->lock_type = 0;     /* marca el slot como no usado */
                nr_locks--;             /* decrementamos el número de bloqueos */
                continue;
        }

        /* Una parte de una región bloqueada se ha desbloqueada */
        if (first <= flp->lock_first) {
                flp->lock_first = last + 1;
                continue;
        }

        if (last >= flp->lock_last) {
                flp->lock_last = first - 1;
                continue;
        }

        /* Mala suerte. Un bloque se ha dividido desbloqueando la mitad*/
        if (nr_locks == NR_LOCKS) return(ENOLCK);
        for (i = 0; i < NR_LOCKS; i++)
                if (file_lock[i].lock_type == 0) break;
        flp2 = &file_lock[i];
        flp2->lock_type = flp->lock_type;
        flp2->lock_pid = flp->lock_pid;
        flp2->lock_inode = flp->lock_inode;
        flp2->lock_first = last + 1;
        flp2->lock_last = flp->lock_last;
        flp->lock_last = first - 1;
        nr_locks++;
}
if (unlocking) lock_revive();

if (req == F_GETLK) {
        if (conflict) {
                /* GETLK y conflicto. Informa del bloqueo conflictivo*/
                flock.l_type = flp->lock_type;
                flock.l_whence = SEEK_SET;
                flock.l_start = flp->lock_first;
                flock.l_len = flp->lock_last - flp->lock_first + 1;
                flock.l_pid = flp->lock_pid;

        } else {
                /* Es GETLK y no hay conflicto */
                flock.l_type = F_UNLCK;
```

```
        }

        /* Copia la estructura flock al que llamo */
        r = sys_copy(FS_PROC_NR, D, (phys_bytes) &flock,
                who, D, (phys_bytes) user_flock, (phys_bytes) sizeof(flock));
        return(r);
 }

 if (ltype == F_UNLCK) return(OK);          /* Desbloquea una región desbloqueada */

 /* No hay conflicto. Si hay espacio, almacena el nuevo bloqueo en la tabla*/
 if (empty == (struct file_lock *) 0) return(ENOLCK);       /* tabla llena */
 empty->lock_type = ltype;
 empty->lock_pid = fp->fp_pid;
 empty->lock_inode = f->filp_ino;
 empty->lock_first = first;
 empty->lock_last = last;
 nr_locks++;
 return(OK);
}


/*==========================================================*
 *                          lock_revive                     *
 *==========================================================*/
PUBLIC void lock_revive()
{
/* Recorre todos los procesos que están esperando por cualquie clase de bloqueo y los
revive. Los que siguen bloqueados se bloquearan de nuevo. Los otros terminarán. Esta
estrategia  es una space-time-tradeoff. Indicando exactamente cuales son desbloqueados
obtendrán un código extra, y la única cosa que conseguiría sería alguna mejora  en casos
extremos. */

 int task;
 struct fproc *fptr;

 for (fptr = &fproc[INIT_PROC_NR + 1]; fptr < &fproc[NR_PROCS]; fptr++){
        task = -fptr->fp_task;
        if (fptr->fp_suspended == SUSPENDED && task == XLOCK) {
                revive( (int) (fptr - fproc), 0);  /*proceso a revivir, y si está colgado en una
                                                tty, es el proceso utilizado.*/
        }
 }
}
```

Preguntas del Filedes.c y del Lock.c

1 – Alternativas para almacenar la "posición del archivo" y problemas que puede presentar su utilización en Minix

La posición del archivo es una dirección de 32 bits que indica la siguiente posición de lectura o escritura. Existen tres alternativas para almacenar este dato:

- Una tabla compartida por todos los procesos donde la entrada k-esima nos diga la posición del archivo con descriptor k.
  Problema: Minix permite la compartición de archivos por varios procesos, por tanto el último proceso que trabaje en un archivo modificará la posición donde debe trabajar el proceso anterior.

- Una tabla de procesos por cada proceso donde almacena la posición del archivo.
  Problema: La llamada FORK exige que tanto el proceso padre como el hijo compartan un solo apuntador que de la posición actual de cada archivo abierto.

- Tabla FILP: Cada archivo abierto tiene tantas entradas ocupa como procesos (que no sean hijos) estén leyendo o escribiendo en él. En cada entrada está la posición del archivo donde está leyendo o escribiendo el proceso correspondiente.

2 - Cuando un proceso quiere abrir un fichero, el Filedes.c debe realizar una serie de comprobaciones. Describirlas.

- Encontrar un descriptor de archivo y una entrada en la tabla FILP libres. Lo realiza la función GET_FD. A partir de una determinada posición de la tabla de procesos empieza a buscar un descrtptor de fichero libre.

- Comprobar que la tabla FILP no esté llena.

3 - Indicar las estructuras más importantes del fichero lock.c

- Estructura file_lock. Ver página 7.
- Estructura flock. Ver página 7.
- Estructura filp. Ver página 3.