open.c
Minix 2.0

Francisco Javier Bordón Caballero

Domingo Alberto Suárez González

© Universidad de Las Palmas de Gran Canaria

Índice de contenidos.

0. El presente índice 1
1. Introducción 2
1.1. Sistemas de ficheros 2
1.1. Archivos y nodos-i 3
1.2. Protección y permisos de acceso 4
1.3. Tipos de archivo 5
1.4. Generalidades sobre la apertura de archivos 6
2. Algoritmo de creación y apertura de archivos 8
3. El fichero *open.c* 9
3.1. Preliminares 9
3.2. Las llamadas *creat* y *open* 10
3.2.1. El procedimiento *do_creat* 13
3.2.2. El procedimiento *do_open* 13
3.2.3. El procedimiento *common_open* 15
3.2.4. El procedimiento *pipe_open* 17
3.2.5. El procedimiento *new_node* 18
3.3. El procedimiento *do_mknod* 19
3.4. El procedimiento *do_close* 21
3.5. El procedimiento *do_lseek* 23
3.6. El procedimiento *do_mkdir* 25
3.7. Diagrama de llamadas 28
3.8. Código de librerías 29
4. Cuestiones 31

1. Introducción.

1.1. Sistemas de ficheros.

Los sistemas de ficheros son particiones lógicas del disco. El núcleo del sistema trabaja con el sistema de ficheros a un nivel lógico y no trata directamente con los discos a nivel físico. Cada disco es considerado como un dispositivo lógico que tiene asociados unos números de dispositivo (minor number y major number). Estos números se utilizan para indexar dentro de una tabla de funciones, cuál tenemos que emplear para acceder al manejador del disco. El manejador del disco se va a encargar de transformar las direcciones lógicas (núcleo) de nuestro sistema de ficheros a direcciones físicas del disco. El major number indica el tipo de dispositivo del que se trata y el minor number indica el número de unidad dentro del dispositivo.

Un sistema de ficheros se compone de una secuencia de bloques lógicos, cada uno de los cuales tiene un tamaño fijo. El tamaño de cada bloque es el mismo para todo el sistema de ficheros.

La estructura que tiene un sistema de ficheros es la siguiente:

Bloque de boot	Superbloque	Lista de nodos – i	Bloques de datos

El superbloque describe el estado de un sistema de ficheros. Contiene información acerca de su tamaño, total de ficheros que puede contener, qué espacio queda libre, etc. Como el acceso a disco suele degradar bastante el tiempo de ejecución de un programa, lo normal es que el núcleo realice la E/S con el disco a través de un buffer caché y que el sistema tenga en memoria una copia del superbloque y de las listas de nodos – i.

La lista de nodos – i tiene una entrada por cada fichero, donde se guarda una descripción del mismo: situación del fichero en el disco, propietario, permisos de acceso, fecha de actualización, etc. Cada fichero en el sistema tiene asociado un nodo – i. El nodo – i contiene la información necesaria para que un proceso pueda acceder al fichero. Esta información incluye: propietario, derechos de acceso, tamaño, localización en el sistema de ficheros, etc. Durante el proceso de arranque del sistema, el núcleo lee la lista de nodos – i del disco y carga una copia en memoria, conocida como tabla de nodos – i. La tabla de nodos – i está cargada siempre en memoria.

En la zona de bloques de datos es donde se encuentra situado el contenido de los ficheros a los que hace referencia la lista de nodos – i. Cada uno de los bloques destinados a datos sólo puede ser asignado a un fichero, tanto si lo ocupa totalmente como si no.

1.2. Archivos y nodos-i.

En Minix, al igual que en la gran mayoría de los sistemas operativos, los archivos pueden contener programas, datos o cualquier otro elemento que el usuario desee. El sistema operativo es el encargado de proporcionar operaciones especiales (en forma de llamadas al sistema) destinadas a la creación, destrucción, lectura y escritura de archivos.

Cada archivo lleva asociado un nodo-i o inode, estructura que ya debemos conocer sobradamente, pero que no estará de más recordar de nuevo, al menos en su estructura básica (figura 1). Una de las funciones principales del nodo-i es indicar dónde están los bloques de datos de cada archivo.

Tipo de archivo y modo de acceso
Identificador de usuario propietario
Tamaño del archivo (2 bytes)
Tiempo desde la última modificación (2 bytes)
Enlaces GID
Número 0 de zona
Número 1 de zona
Número 2 de zona
Número 3 de zona
Número 4 de zona
Número 5 de zona
Número 6 de zona
Indexación indirecta
Indexación indirecta doble

Figura 1. Estructura de un nodo-i.

Hay que notar que el nombre del fichero no queda especificado en su nodo – i. Es en los ficheros de tipo directorio donde a cada nombre de fichero se le asocia su nodo –i correspondiente.

La tabla de nodos –i contiene la misma información que la lista de nodos – i, además de la siguiente información:

- Estado del nodo –i
  - Si el nodo – i está bloqueado;
  - Si hay algún proceso esperando a que el nodo – i quede desbloqueado;
  - Si la copia del nodo – i que hay en memoria difiere de la que hay en el disco;
  - Si la copia de los datos del fichero que hay en memoria difiere de los datos que hay en el disco.
- El nº de dispositivo lógico del sistema de ficheros que contiene al fichero.
- El número de nodo –i .
- Punteros a otros nodos – i cargados en memoria. El núcleo enlaza los nodos – i sobre una hash y sobre una lista libre.
- Un contador que indica el número de copias del nodo – i que están activas (por ejemplo, porque el fichero está abierto por varios procesos).

1.3. Protección y permisos de acceso.

A cada archivo se le asocia también un código binario de protección de 9 bits. Dicho código consta de 3 campos de 3 bits cada uno. El primer campo establece los permisos de acceso para el propietario del fichero, el segundo los permisos para los restantes miembros del grupo del propietario, y el tercero se reserva para el resto de usuarios del sistema.

Dentro de cada campo, los 3 bits se conocen como **bits rwx**, y corresponden, respectivamente, a accesos de lectura, escritura y ejecución.

Por ejemplo, el código de protección: rwxr-x--x

establece que el propietario puede leer, escribir y ejecutar el archivo, mientras que otros miembros del grupo sólo pueden leer y ejecutar. Al resto de usuarios únicamente se les concede permiso de ejecución.

En el caso particular de que un archivo sea de tipo directorio (se comentará de inmediato), el tercer bit de permisos (x) habilitado posibilita la búsqueda en dicho directorio.

1.4. Tipos de archivo.

En MINIX existen cuatro tipos de archivo:

a) Regulares:

Contienen bytes de datos organizados como un array lineal. Son los ficheros normales de uso común. Se subdividen a su vez en tipos distintos según su funcionalidad o uso. Podemos así encontrar archivos de datos, archivos de texto, programas fuente en cualquier lenguaje, programas ejecutables (binarios), etc.

b) Directorios:

Son los ficheros que nos permiten darle una estructura jerárquica a los sistemas de ficheros. Su función fundamental consiste en establecer la relación que existe entre el nombre de un fichero y su nodo – i correspondiente. Están organizados como una secuencia de entradas, cada una de las cuales contiene un número de nodo – i y el nombre de un fichero que pertenece al directorio. Las dos primeras entradas de un directorio reciben los nombres "." y "..".

Los procesos pueden leer el contenido de un directorio como si se tratase de un fichero de datos, sin embargo no pueden modificarlos. El derecho de escritura en un directorio está reservado al núcleo.

Los permisos de acceso a un directorio tiene los siguientes significados:

- Permiso de lectura. Permite que un proceso lea ese directorio.
- Permiso de escritura. Permite a un proceso crear una nueva entrada en el directorio o borrar alguna ya existente.
- Permiso de ejecución. Autoriza a un proceso para buscar el nombre de un fichero dentro del directorio.

Desde el punto de vista del usuario, vamos a referenciar los ficheros mediante su ruta de acceso. Llamadas como open, chdir o link reciben como parámetro de entrada la ruta de acceso de un fichero y no su nodo – i. El núcleo es quien se encarga de traducir la ruta de acceso de un fichero a su nodo – i correspondiente. El algoritmo se encarga de analizar los componentes de la ruta de acceso y de leer los nodos – i intermedios necesarios para verificar que se trata de una ruta de acceso correcta y que el fichero realmente existe.

c) Especiales:

Son necesarios para permitir que los dispositivos de E/S puedan utilizarse exactamente igual que los archivos, es decir, empleando las mismas llamadas al

sistema que con estos. Existen dos tipos de archivos especiales: los de bloque, como los discos, y los de caracter, como las terminales o las impresoras.

Todos los archivos especiales tienen un numero de dispositivo mayor y un numero de dispositivo menor. El primero indica el tipo de dispositivo, mientras que el segundo especifica cuál de los dispositivos de esa clase se está referenciando. El número de dispositivo mayor establece qué manejador de dispositivo ha de encargarse de manejar el archivo, mientras que el número menor se le pasa a dicho manejador como un parámetro más de la llamada.

d) Tuberías (pipes):

Es un fichero con una estructura similar a la de un fichero ordinario. La diferencia principal con éstos es que los datos de una tubería son transitorios. Esto quiere decir que una vez que un dato es leído de una tubería, desaparece de ella y nunca más podrá ser leído. Se utiliza para conectar dos procesos entre sí.

Cuando un determinado proceso A desee enviar datos a otro proceso B, aquél escribirá en la tubería como si se tratase de un archivo de salida convencional. El proceso B podrá posteriormente leer datos de la tubería análogamente a como lo haría de un archivo de entrada.

En contraposición a las tuberías con nombre tenemos las tuberías sin nombre que no tienen asociado ningún nombre de fichero y que sólo existen mientras algún proceso está unido a ellas. Las tuberías sin nombre actúan como un canal de comunicación entre dos procesos y tienen asociado un nodo – i mientras existen.

Las tuberías sin nombre son creadas a través de la llamada pipe (desde un proceso), mientras que las tuberías con nombre se crean con la orden mknod (desde la línea de órdenes del sistema) o con la llamada mknod (desde un proceso).

1.5. Generalidades sobre la apertura de archivos.

Antes de que un archivo pueda leerse o escribirse, debe ser abierto. Es éste el momento de comprobar los permisos de acceso. Caso de poder acceder al archivo, el sistema devuelve al usuario un entero corto llamado *descriptor de archivo*, que servirá para identificar el archivo en cualquier operación posterior. En caso de no superar el test de los permisos, se produce un código de error.

Cuando un archivo es abierto, se localiza su nodo-i y se trae a memoria, donde permanecerá hasta que se cierre el archivo.

Al igual que los manejadores del Kernel y de la memoria, el sistema de archivos conserva parte de la tabla de procesos dentro de su espacio de direcciones. En lo que

a nosotros nos compete, tres de los campos son de especial interés.

Los dos primeros son apuntadores a los nodos-i del directorio raíz y el directorio de trabajo, y son necesarios para facilitar los accesos a los archivos mediante rutas absolutas o relativas, respectivamente. El tercer campo es un array que se indexa mediante el descriptor anteriormente mencionado de archivo.

Minix introduce una tabla compartida llamada *filp*, que se sitúa virtualmente entre la tabla de procesos y la tabla de nodos-i. Dicha tabla se hace necesaria para solventar los problemas que se derivan del acceso de varios procesos a un archivo compartido, más concretamente de la necesidad de almacenar la posición del archivo en la que está trabajando cada uno de los procesos. Para más información sobre la tabla *filp*, nos remitimos al archivo *filedes.c*, en el cual se puede encontrar una descripción completa tanto de sus estructura como de su funcionamiento.

2. Algoritmo de creación y apertura de archivos.

Como paso previo a la explicación del código de *open.c*, y con el fin de centrar en su justa medida cada uno de los procedimientos que expondremos a continuación, incluimos en este punto el algoritmo genérico que se seguirá cada vez que se ejecute una llamada al sistema *open* o *creat*.

Conseguir el nombre del fichero a crear o abrir llamando a *fetch_name*. Si no es posible, retornar indicando el error.

Llamada a *common_open*. Éste es realmente el procedimiento común que realiza la creación y apertura de un fichero, y su funcionamiento es el siguiente:

Llamar a *get_fd* para asegurar que existen un descriptor de archivo y una entrada en la tabla filp disponibles. Si no es así, retornar indicando el error.

Si estamos CREANDO un fichero entonces

Crea un nuevo nodo llamando al procedimiento *new_node*. Si no es posible, retorna indicando el error.

Si no es así (estamos ABRIENDO un fichero) entonces

Llama a *eat_path* para traer el nodo-i a memoria. Si no es posible, retorna indicando el error.

Si el fichero a crear ya existía o bien estamos en un intento de apertura, entonces

Chequear protecciones llamando a *forbidden* para ver si el fichero puede ser abierto.

Si tenemos acceso, hay que actuar de acuerdo con el tipo de fichero que estemos manejando, ya que cada uno precisa un procesamiento diferente.

Caso de fichero regular:

Si es crear un fichero que ya existe entonces

Trunca el fichero regular llamando a *truncate*, es decir, elimina todas las zonas del NODO-i y lo marca como sucio.
Elimina algunos campos del nodo-i llamando a *wipe_inode*.

Caso de directorio:
Los directorios pueden ser solo leídos y no escritos.

Caso de fichero especial:
Se llama a la tarea del dispositivo en cuestión, pues los archivos especiales pueden necesitar procesamiento específico al abrirse. Esto se hace llamando a *dev*

Caso de fichero tubería (pipe):
Se llama a *pipe_open*, para comprobar que haya al menos un par lector/escritor para la tubería. Si no lo hay, suspende al llamador; de lo contrario revive a los otros procedimientos bloqueados esperando en la tubería.

Si en el paso anterior hubo algún error entonces
Liberar al nodo-i con *put_inode* y retornar el tipo de error.

Llenar los campos en la tabla filp y devolver el descriptor de archivo.

3. El fichero *open.c*.

3.1. Preliminares.

Empezamos ya con el código de *open.c*; estas son las declaraciones de apertura:

```
/* Este fichero contiene los procedimientos crear, abrir, cerrar y buscar en ficheros.
    Los puntos de entrada de este archivo son los siguientes:
        do_creat:   realiza la llamada al sistema CREAT
        do_mknod: realiza la llamada al sistema MKNOD
        do_open:   realiza la llamada al sistema OPEN
        do_close:   realiza la llamada al sistema CLOSE
        do_lseek:   realiza la llamada al sistema LSEEK
        do_mkdir:   realiza la llamada al sistema MKDIR      */
```

```
#include "fs.h"
#include <sys/stat.h>
#include <fcntl.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "buf.h"
#include "dev.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "lock.h"
#include "param.h"

PRIVATE message dev_mess;
PRIVATE char mode_map[] = {R_BIT, W_BIT, R_BIT|W_BIT, 0};

FORWARD _PROTOTYPE( int common_open, (int oflags, Mode_t omode)         );
FORWARD _PROTOTYPE( int pipe_open, (struct inode *rip,Mode_t bits,int
oflags));
FORWARD _PROTOTYPE( struct inode *new_node, (char *path, Mode_t bits,
                                        zone_t z0)  );
```

3.2. Las llamadas *creat* y *open*.

Las llamadas al sistema *creat(nombre,modo)* y *open(nombre,modo)* se implementan mediante los procedimientos *do_creat* y *do_open*, respectivamente. Hay que tener en cuenta que las siguientes llamadas tienen la misma funcionalidad:

fd = creat (“mifichero”, 0666);

fd = open (“mifichero”, O_WRONLY | O_CREAT | O_TRUNC, 0666);

Procedimientos externos utilizados:

fetch_name(name, name_length, flag) ***(UTILITY.C)***

Toma la trayectoria especificada y la coloca en la variable global *user_path*. Si *flag* es igual a M3 y la longitud es menor o igual a 14 bytes, la trayectoria está presente en el mensaje; en caso contrario, hay que buscar la cadena en el espacio de usuario.

get_fd(0, bits, &fd, &fil_ptr) ***(FILEDES.C)***

Busca en la tabla de procesos del sistema de fichero (fproc) un descriptor de archivo (fd) libre, así como un hueco libre en la tabla filp.

find_filp(rip, bits) ***(FILEDES.C)***

Barre la tabla filp buscando una ranura que se refiera al nodo-i *rip*. Se usa para determinar si todavía hay alguien interesado en el extremo final de una tubería. Si no encuentra a nadie, retorna NIL_FILP.

eat_path(user_path) ***(PATH.C)***

Analiza gramaticalmente la trayectoria *user_path* y coloca su nodo-i en la tabla de nodos-i. Si no es posible, devuelve NIL_INODE como valor de función y un código de error en *err_code*.

advance(dirp,string) ***(PATH.C)***

Dada la dirección al nodo-i de un directorio y una componente de una trayectoria, busca el nodo-i de esta componente en el directorio y lo recuperara. Si no lo encuentra devuelve NIL_INODE.

forbidden(rip, bits, 0) ***(PROTECT.C)***

Dado un apuntador a un nodo-i (rip) y el acceso deseado (bits), determina si está permitido el acceso. De ser así, se devuelve OK; en otro caso, el resultado es EACCESS.

truncate(rip) ***(LINK.C)***

Elimina todas las zonas del nodo-i apuntado por *rip*.

wipe_inode(rip) ***(INODE.C)***

Elimina (limpia) algunos campos del nodo-i.

put_inode(rip) ***(INODE.C)***

El solicitante ya no está usando este nodo-i. Si nadie más lo esta usando, se reescribirá en el disco inmediatamente. Si no tiene enlaces (links), hay que truncarlo y devolverlo al pozo (pool) de nodos-i disponibles.

rw_inode**(rip,WRITING)** **(INODE.C)**

Escribe en disco un nodo-i modificado.

alloc_inode(dev,bits) ***(INODE.C)***

En la creación de un nuevo archivo, este procedimiento es el encargado de conseguir un nodo-i libre.

dev_open(rip, bits, oflags) ***(DEVICE.C)***

Los archivos especiales pueden necesitar procesamiento especial al abrirse. Por tanto, es necesario buscar qué tarea da servicio al dispositivo en cuestión.

suspend(XOPEN) ***(PIPE.C)***

Suspende al proceso llamador.

release(rip, call_nr, susp_count) ***(PIPE.C)***

Revive a un proceso bloqueado.

Procedimientos internos:

common_open(oflags, omode)

Cuerpo común de los procedimientos de creación y apertura. Es llamado tanto por *do_creat* como por *do_open*. Recibe como parámetros los permisos de acceso (oflags) y el tipo de fichero (omode).

pipe_open(rip, bits, oflags)

Chequea si hay al menos un lector y un escritor para la tubería. De no haberlos, se suspende el proceso llamador; en otro caso, revive a todos los procesos que estaban bloqueados en el pipe.

new_node(path, bits, z0)
Crea un nuevo nodo-i.

Parámetros de entrada:

name: nombre del fichero a crear o abrir
mode: modo del archivo

Parámetros de salida:

fd: descriptor del archivo creado o abierto
err_code / r: código de error

3.2.1. El procedimiento *do_creat*.

Este procedimiento simplemente obtiene el nombre del archivo a crear, llamando a *fetch_name*, y pasa el control a *common_open*.

```
/*===========================================================================*
 *                      do_creat                                             *

*===========================================================================*/
PUBLIC int do_creat()
{
/* Perform the creat(name, mode) system call. */
  int r;

  if (fetch_name(name, name_length, M3) != OK) return(err_code);
  r = common_open(O_WRONLY | O_CREAT | O_TRUNC, (mode_t) mode);
  return(r);
}
```

3.2.2. El procedimiento *do_open*.

Este procedimiento es esencialmente similar al anterior, con la salvedad de la llamada a *fetch_name*, que difiere dependiendo de si el modo de apertura incluye el flag O_CREAT o no.

```
/*===========================================================================*
 *                      do_open                                              *

*===========================================================================*/
PUBLIC int do_open()
{
/* Perform the open(name, flags,...) system call. */
```

```
  int create_mode = 0;        /* is really mode_t but this gives problems
*/
  int r;

  /* If O_CREAT is set, open has three parameters, otherwise two. */
  if (mode & O_CREAT) {
      create_mode = c_mode;
      r = fetch_name(c_name, name1_length, M1);
  } else {
      r = fetch_name(name, name_length, M3);
  }

  if (r != OK) return(err_code); /* name was bad */
  r = common_open(mode, create_mode);
  return(r);
}
```

3.2.3. El procedimiento *common_open*.

Implementa el algoritmo descrito en el apartado 2 (código común para las llamadas de creación y apertura de ficheros).

```
/*===========================================================================*
 *                      common_open                                          *

*===========================================================================*/

PRIVATE int common_open(oflags, omode)
register int oflags;
mode_t omode;
{
/* Common code from do_creat and do_open. */

  register struct inode *rip;
  int r, b, major, task, exist = TRUE;
  dev_t dev;
  mode_t bits;
  off_t pos;
  struct filp *fil_ptr, *filp2;

  /* Remap the bottom two bits of oflags. */
  bits = (mode_t) mode_map[oflags & O_ACCMODE];

  /* See if file descriptor and filp slots are available. */
  if ( (r = get_fd(0, bits, &fd, &fil_ptr)) != OK) return(r);

  /* If O_CREATE is set, try to make the file. */
  if (oflags & O_CREAT) {
      /* Create a new inode by calling new_node(). */
        omode = I_REGULAR | (omode & ALL_MODES & fp->fp_umask);
      rip = new_node(user_path, omode, NO_ZONE);
      r = err_code;
      if (r == OK) exist = FALSE;      /* we just created the file */
      else if (r != EEXIST) return(r); /* other error */
      else exist = !(oflags & O_EXCL); /* file exists, if the O_EXCL
                                  flag is set this is an error */
  } else {
       /* Scan path name. */
      if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
  }

  /* Claim the file descriptor and filp slot and fill them in. */
  fp->fp_filp[fd] = fil_ptr;
  fil_ptr->filp_count = 1;
  fil_ptr->filp_ino = rip;
  fil_ptr->filp_flags = oflags;

  /* Only do the normal open code if we didn't just create the file. */
  if (exist) {
      /* Check protections. */
      if ((r = forbidden(rip, bits)) == OK) {
            /* Opening reg. files directories and special files differ. */
            switch (rip->i_mode & I_TYPE) {
               case I_REGULAR:
                  /* Truncate regular file if O_TRUNC. */
                  if (oflags & O_TRUNC) {
```

```
                if ((r = forbidden(rip, W_BIT)) !=OK) break;
                truncate(rip);
                wipe_inode(rip);
                /* Send the inode from the inode cache to the
                 * block cache, so it gets written on the next
                 * cache flush.
                 */
                rw_inode(rip, WRITING);
        }
        break;

    case I_DIRECTORY:
        /* Directories may be read but not written. */
        r = (bits & W_BIT ? EISDIR : OK);
        break;

    case I_CHAR_SPECIAL:
    case I_BLOCK_SPECIAL:
        /* Invoke the driver for special processing. */
        dev_mess.m_type = DEV_OPEN;
        dev = (dev_t) rip->i_zone[0];
        dev_mess.DEVICE = dev;
        dev_mess.COUNT = bits | (oflags & ~O_ACCMODE);
        major = (dev >> MAJOR) & BYTE;      /* major device nr */
        if (major <= 0 || major >= max_major) {
                r = ENODEV;
                break;
        }
        task = dmap[major].dmap_task; /* device task nr */
        (*dmap[major].dmap_open)(task, &dev_mess);
        r = dev_mess.REP_STATUS;
        break;

    case I_NAMED_PIPE:
        oflags |= O_APPEND;     /* force append mode */
        fil_ptr->filp_flags = oflags;
        r = pipe_open(rip, bits, oflags);
        if (r == OK) {
                /* See if someone else is doing a rd or wt on
                 * the FIFO.  If so, use its filp entry so the
                 * file position will be automatically shared.
                 */
                b = (bits & R_BIT ? R_BIT : W_BIT);
                fil_ptr->filp_count = 0; /* don't find self */
                if ((filp2 = find_filp(rip, b)) != NIL_FILP) {
                        /* Co-reader or writer found. Use it.*/
                        fp->fp_filp[fd] = filp2;
                        filp2->filp_count++;
                        filp2->filp_ino = rip;
                        filp2->filp_flags = oflags;

                        /* i_count was incremented incorrectly
                         * by eatpath above, not knowing that
                         * we were going to use an existing
                         * filp entry.  Correct this error.
                         */
                        rip->i_count--;
                } else {
                        /* Nobody else found.  Restore filp. */
                        fil_ptr->filp_count = 1;
                        if (b == R_BIT)
```

```
                                    pos = rip->i_zone[V2_NR_DZONES+1];
                               else
                                    pos = rip->i_zone[V2_NR_DZONES+2];
                               fil_ptr->filp_pos = pos;
                          }
                     }
                     break;
               }
          }
     }

     /* If error, release inode. */
     if (r != OK) {
          fp->fp_filp[fd] = NIL_FILP;
          fil_ptr->filp_count= 0;
          put_inode(rip);
          return(r);
     }

     return(fd);
}
```

3.2.4. El procedimiento *pipe_open*.

Esta función es llamada por *do_creat* y *do_open* (o, más concretamente, por *common_open*). Su cometido es chequear si existe al menos un lector y un escritor para el pipe. Si no existen ambos, se suspende al llamador; caso de haber al menos uno de cada tipo, se reviven todos los procesos bloqueados esperando en la tubería.

```
/*===========================================================================*
 *                      pipe_open                                            *
*===========================================================================*/

PRIVATE int pipe_open(rip, bits, oflags)
register struct inode *rip;
register mode_t bits;
register int oflags;
{
/*  This function is called from common_open. It checks if
 *  there is at least one reader/writer pair for the pipe, if not
 *  it suspends the caller, otherwise it revives all other blocked
 *  processes hanging on the pipe.
 */

  if (find_filp(rip, bits & W_BIT ? R_BIT : W_BIT) == NIL_FILP) {
      if (oflags & O_NONBLOCK) {
            if (bits & W_BIT) return(ENXIO);
      } else
            suspend(XPOPEN);  /* suspend caller */
  } else if (susp_count > 0) {/* revive blocked processes */
      release(rip, OPEN, susp_count);
      release(rip, CREAT, susp_count);
  }
  rip->i_pipe = I_PIPE;

  return(OK);}}
```

3.2.5. El procedimiento *new_node*.

Esta función asigna un nuevo nodo-i, crea una entrada para él en el directorio especificado (path) y lo inicializa. Como resultado dará la dirección en la tabla INODE del nuevo nodo-i conseguido.

Básicamente, lo que se hace es comprobar que no exista ya la entrada a la que se piensa asignar el nuevo nodo conseguido. Una vez averiguado esto, y si existen nodos libres en el sistema de archivos correspondiente, se crea la entrada y se actualiza el nodo.

Parámetros de entrada:

path: apuntador al nombre de la trayectoria
bits: modo de acceso del nuevo nodo-i
z0: número de la zona 0 asignada a este nodo-i

Parámetros de salida:

Retornará un puntero al nodo-i conseguido, si se logra, o un código NIL_NODE como indicativo de que no pudo ser.

```
/*===========================================================================
*
 *                      new_node                                 *

*===========================================================================*/

PRIVATE struct inode *new_node(path, bits, z0)
char *path;             /* pointer to path name */
mode_t bits;                  /* mode of the new inode */
zone_t z0;              /* zone number 0 for new inode */
{
/* New_node() is called by common_open(), do_mknod(), and do_mkdir().
 * In all cases it allocates a new inode, makes a directory entry for it on
 * the path 'path', and initializes it.  It returns a pointer to the inode
if
 * it can do this; otherwise it returns NIL_INODE.  It always sets
'err_code'
 * to an appropriate value (OK or an error code).
 */

  register struct inode *rlast_dir_ptr, *rip;
  register int r;
  char string[NAME_MAX];

  /* See if the path can be opened down to the last directory. */
  if ((rlast_dir_ptr = last_dir(path, string)) == NIL_INODE)
return(NIL_INODE);

  /* The final directory is accessible. Get final component of the path. */
```

```
  rip = advance(rlast_dir_ptr, string);
  if ( rip == NIL_INODE && err_code == ENOENT) {
      /* Last path component does not exist.  Make new directory entry. */
      if ( (rip = alloc_inode(rlast_dir_ptr->i_dev, bits)) == NIL_INODE) {
            /* Can't creat new inode: out of inodes. */
            put_inode(rlast_dir_ptr);
            return(NIL_INODE);
      }

      /* Force inode to the disk before making directory entry to make
       * the system more robust in the face of a crash: an inode with
       * no directory entry is much better than the opposite.
       */
      rip->i_nlinks++;
      rip->i_zone[0] = z0;          /* major/minor device numbers */
      rw_inode(rip, WRITING);       /* force inode to disk now */

      /* New inode acquired.  Try to make directory entry. */
      if ((r = search_dir(rlast_dir_ptr, string, &rip->i_num,ENTER)) != OK)
{
            put_inode(rlast_dir_ptr);
            rip->i_nlinks--;  /* pity, have to free disk inode */
            rip->i_dirt = DIRTY;    /* dirty inodes are written out */
            put_inode(rip);   /* this call frees the inode */
            err_code = r;
            return(NIL_INODE);
      }

  } else {
      /* Either last component exists, or there is some problem. */
      if (rip != NIL_INODE)
            r = EEXIST;
      else
            r = err_code;
  }

  /* Return the directory inode and exit. */
  put_inode(rlast_dir_ptr);
  err_code = r;
  return(rip);
}
```

3.3. El procedimiento *do_mknod*.

El procedimiento *do_mknod* implementa la llamada al sistema *mknod*(nombre,modo,dirección). Esta llamada asigna un nodo-i a la trayectoria parcial dada por *nombre*, con una *dirección* de zona en el disco también dada, y con un *modo* determinado. La única restricción que se impone es que realmente exista un nodo-i disponible y que la trayectoria sea válida.

Dado que, al contrario de lo que sucede en *do_creat*, no se hacen comprobaciones previas antes de asignar un nodo-i, esta llamada sólo se permite en modo de superusuario.

Procedimientos externos utilizados:

fetch_name

put_inode

Procedimientos internos:

new_node

Parámetros de entrada:

name: determina la trayectoria parcial a la que se quiere asignar un nodo-i
mode: el modo que se desea asignar a esa entrada
addr: dirección de la zona en disco en que se almacenará la información perteneciente a la entrada

Parámetros de salida:

Se retornará un OK o un valor de error. Se actualizan algunos campos del mensaje que dio lugar a la llamada.

```
/*===========================================================================
*
 *                    do_mknod                                 *

*===========================================================================*/
PUBLIC int do_mknod()
{
/* Perform the mknod(name, mode, addr) system call. */

  register mode_t bits, mode_bits;
  struct inode *ip;

  /* Only the super_user may make nodes other than fifos. */
  mode_bits = (mode_t) m.m1_i2;     /* mode of the inode */
  if (!super_user && ((mode_bits & I_TYPE) != I_NAMED_PIPE)) return(EPERM);
  if (fetch_name(m.m1_p1, m.m1_i1, M1) != OK) return(err_code);
  bits = (mode_bits & I_TYPE) | (mode_bits & ALL_MODES & fp->fp_umask);
  ip = new_node(user_path, bits, (zone_t) m.m1_i3);
  put_inode(ip);
  return(err_code);
}
```

3.4. El procedimiento *do_close*.

El procedimiento *do_close* implementa la llamada al sistema *close(fd)* de cierre de un archivo. Para ello lo único que hace es disminuir el contador filp correspondiente al fichero indicado. Si el contador disminuido resulta ser cero, se libera el nodo-i correspondiente (mediante *put_inode*). Si el nodo está modificado (sucio), se escribe en disco. En el caso de que el archivo que se pretenda cerrar sea un pipe o un archivo especial, se harán algunas operaciones más.

Procedimientos externos utilizados:

get_filp(fd)
Mira si el descriptor de archivo que se le pasa es válido y, si es así, retorna un apuntador a la entrada correspondiente en la tabla filp.

do_sync()
Salva la información contenida en la caché.

invalidate(rip->i_zone[0])
Borra de la caché los bloques de un dispositivo.

dev_close(rip)
Cierra un dispositivo.

mounted(rip)
Indica si el dispositivo *rip* está montado o no.

release

put_inode

Parámetros de entrada:
fd: descriptor del fichero

Parámetros de salida:
La función devolverá un OK o un código de error.

```
/*===========================================================================
*
 *                      do_close                                 *

*===========================================================================*
/

PUBLIC int do_close()
{
/* Perform the close(fd) system call. */

  register struct filp *rfilp;
  register struct inode *rip;
  struct file_lock *flp;
  int rw, mode_word, major, task, lock_count;
  dev_t dev;

  /* First locate the inode that belongs to the file descriptor. */
  if ( (rfilp = get_filp(fd)) == NIL_FILP) return(err_code);
  rip = rfilp->filp_ino;      /* 'rip' points to the inode */

  if (rfilp->filp_count - 1 == 0 && rfilp->filp_mode != FILP_CLOSED) {
      /* Check to see if the file is special. */
      mode_word = rip->i_mode & I_TYPE;
      if (mode_word == I_CHAR_SPECIAL || mode_word == I_BLOCK_SPECIAL) {
            dev = (dev_t) rip->i_zone[0];
            if (mode_word == I_BLOCK_SPECIAL)  {
                  /* Invalidate cache entries unless special is mounted
                   * or ROOT
                   */
                  if (!mounted(rip)) {
                          (void) do_sync();     /* purge cache */
                        invalidate(dev);
                  }
            }
            /* Use the dmap_close entry to do any special processing
             * required.
             */
            dev_mess.m_type = DEV_CLOSE;
            dev_mess.DEVICE = dev;
            major = (dev >> MAJOR) & BYTE;      /* major device nr */
            task = dmap[major].dmap_task; /* device task nr */
            (*dmap[major].dmap_close)(task, &dev_mess);
      }
  }

  /* If the inode being closed is a pipe, release everyone hanging on it.
*/
  if (rip->i_pipe == I_PIPE) {
      rw = (rfilp->filp_mode & R_BIT ? WRITE : READ);
      release(rip, rw, NR_PROCS);
  }

  /* If a write has been done, the inode is already marked as DIRTY. */
  if (--rfilp->filp_count == 0) {
      if (rip->i_pipe == I_PIPE && rip->i_count > 1) {
            /* Save the file position in the i-node in case needed later.
             * The read and write positions are saved separately.  The
             * last 3 zones in the i-node are not used for (named) pipes.
             */
            if (rfilp->filp_mode == R_BIT)
```

```
                rip->i_zone[V2_NR_DZONES+1] = (zone_t) rfilp->filp_pos;
            else
                rip->i_zone[V2_NR_DZONES+2] = (zone_t) rfilp->filp_pos;
        }
        put_inode(rip);
    }

    fp->fp_cloexec &= ~(1L << fd);    /* turn off close-on-exec bit */
    fp->fp_filp[fd] = NIL_FILP;

    /* Check to see if the file is locked.  If so, release all locks. */
    if (nr_locks == 0) return(OK);
    lock_count = nr_locks;      /* save count of locks */
    for (flp = &file_lock[0]; flp < &file_lock[NR_LOCKS]; flp++) {
        if (flp->lock_type == 0) continue;  /* slot not in use */
        if (flp->lock_inode == rip && flp->lock_pid == fp->fp_pid) {
            flp->lock_type = 0;
            nr_locks--;
        }
    }
    if (nr_locks < lock_count) lock_revive();     /* lock released */
    return(OK);
}
```

3.5. El procedimiento *do_lseek*.

El procedimiento *do_lseek* implementa la llamada al sistema *lseek*. Para ello, se colocará a un valor determinado el campo de la tabla filp que indica la posición en el archivo.

Hay tres posibilidades a la hora de cambiar este valor, llevándose a cabo la elección entre ellas mediante un parámetro. Las posibilidades son las siguientes:

- Colocar el valor de la posición del archivo a un valor absoluto (OFFSET).

- Hacer que la nueva posición del archivo sea igual a la ultima más un desplazamiento.

- Y por último, actualizar la posición del archivo a un valor que viene dado por el tamaño actual del fichero más el desplazamiento dado.

Además, *lseek* inhibirá la lectura anticipada de bloques que se lleva a cabo cuando el sistema detecta una lectura secuencial de un archivo, ya que suponemos que *lseek* va a servir para el acceso directo al mismo.

La llamada *lseek* no funciona con tuberías, dándose el correspondiente código de error si se intenta acceder a alguna.

Procedimientos externos utilizados:

get_filp

Parámetros de entrada:

No estan especificados explícitamente, sino que se sacan del mensaje pasado. Serán los siguientes:

ls_fd: da el identificador de archivo sobre el que se quiere trabajar y a través del cual podremos obtener la entrada de la tabla filp correspondiente

offset: es el desplazamiento que va a determinar el nuevo valor del campo posición del archivo (ffilp->filp_pos)

whence: es el parámetro que dicta qué tipo de desplazamiento se va a realizar

Parámetros de salida:

Se devolverá OK o un código identificativo de error.

```
/*===========================================================================
*
 *                    do_lseek                               *

*===========================================================================*
/
PUBLIC int do_lseek()
{
/* Perform the lseek(ls_fd, offset, whence) system call. */

  register struct filp *rfilp;
  register off_t pos;

  /* Check to see if the file descriptor is valid. */
  if ( (rfilp = get_filp(ls_fd)) == NIL_FILP) return(err_code);

  /* No lseek on pipes. */
  if (rfilp->filp_ino->i_pipe == I_PIPE) return(ESPIPE);

  /* The value of 'whence' determines the start position to use. */
  switch(whence) {
      case 0:     pos = 0;    break;
      case 1: pos = rfilp->filp_pos;      break;
      case 2: pos = rfilp->filp_ino->i_size;    break;
      default: return(EINVAL);
  }
```

```
  /* Check for overflow. */
  if (((long)offset > 0) && ((long)(pos + offset) < (long)pos))
return(EINVAL);
  if (((long)offset < 0) && ((long)(pos + offset) > (long)pos))
return(EINVAL);
  pos = pos + offset;

  if (pos != rfilp->filp_pos)
      rfilp->filp_ino->i_seek = ISEEK;    /* inhibit read ahead */
  rfilp->filp_pos = pos;
  reply_l1 = pos;       /* insert the long into the output message */
  return(OK);
}
```

3.6. El procedimiento *do_mkdir*.

El procedimiento *do_mkdir* implementa la llamada al sistema que permite la creacion de un subdirectorio, *mkdir*(nombre,modo). Dado el nuevo nombre de directorio, expande este nombre al de toda la trayectoria, de una forma similar a como se hizo en *mknod*, mediante el procedimiento *fetch_name*. Una vez conocida toda la trayectoria y siempre que se pueda, o sea, si hay espacio en disco, el nombre es válido y no existe uno igual, se creará el subdirectorio.

El procedimiento para crear un subdirecorio es sencillo. Una vez obtenido un nodo-i libre, y contando con que no se haya producido ningún error, se crea el archivo del directorio al que este nodo-i apunta. Las dos primeras entradas en este archivo tendran como nombre "." y ".." , y darán el numero del nodo-i del directorio actual y de su directorio padre, respectivamente. Igualmente, la entrada generada en el directorio padre se actualiza con el nombre del directorio creado y un enlace a su nodo-i.

Procedimientos externos utilizados:

fetch_name

put_inode

last_dir(user_path, string)

Dada una trayectoria, trae a memoria el nodo-i de la ultima componente y devuelve un puntero a él.

search_dir(rip, dot1, &dot, ENTER)

Mete en el directorio apuntado por ldirp (dirección del nodo-i del directorio) la cadena *string* con el número de nodo-i, si flag es igual a ENTER.

Procedimientos internos:

new_node

Parámetros de entrada:

Vienen definidos en el mensaje de entrada, y son los siguientes:

name: nombre del subdirectorio que se pretende crear
mode: modo de protección que se desea asignar

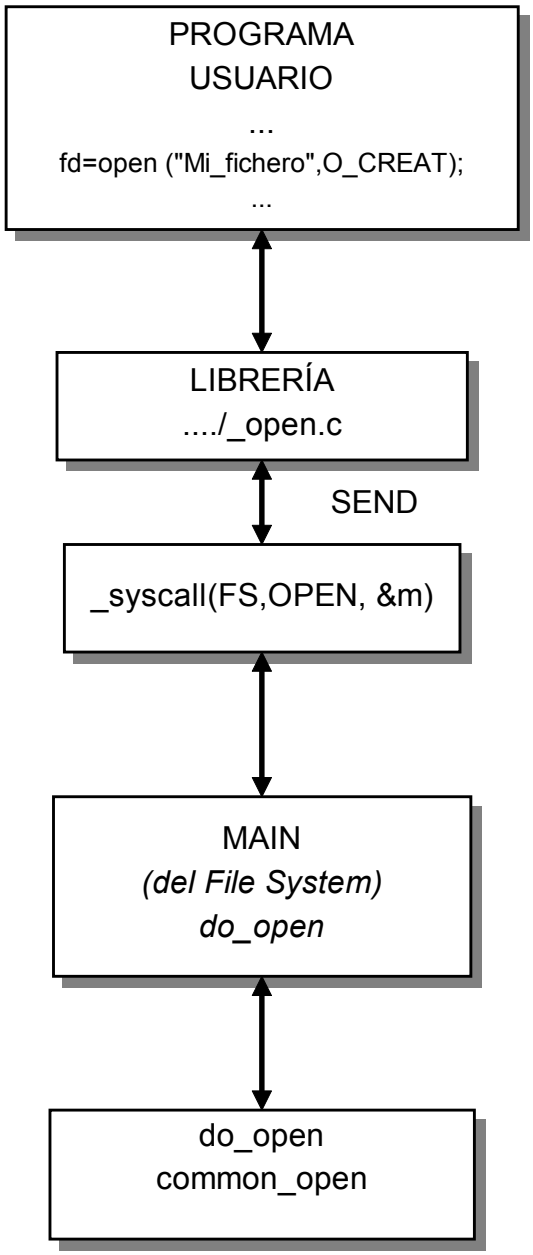
Parámetros de salida:

Una vez más, se devolverá un indicativo de error o una confirmación de éxito.

```
/*===========================================================================
*
 *                      do_mkdir                              *

*============================================================================*
/

PUBLIC int do_mkdir()
{
/* Perform the mkdir(name, mode) system call. */

  int r1, r2;                 /* status codes */
  ino_t dot, dotdot;          /* inode numbers for . and .. */
  mode_t bits;                /* mode bits for the new inode */
  char string[NAME_MAX];      /* last component of the new dir's path name
*/
  register struct inode *rip, *ldirp;

  /* Check to see if it is possible to make another link in the parent dir.
*/
  if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
  ldirp = last_dir(user_path, string);    /* pointer to new dir's parent */
  if (ldirp == NIL_INODE) return(err_code);
  if ( (ldirp->i_nlinks & BYTE) >= LINK_MAX) {
      put_inode(ldirp); /* return parent */
      return(EMLINK);
  }
```

```
  /* Next make the inode. If that fails, return error code. */
  bits = I_DIRECTORY | (mode & RWX_MODES & fp->fp_umask);
  rip = new_node(user_path, bits, (zone_t) 0);
  if (rip == NIL_INODE || err_code == EEXIST) {
      put_inode(rip);         /* can't make dir: it already exists */
      put_inode(ldirp); /* return parent too */
      return(err_code);
  }

  /* Get the inode numbers for . and .. to enter in the directory. */
  dotdot = ldirp->i_num;      /* parent's inode number */
  dot = rip->i_num;           /* inode number of the new dir itself */

  /* Now make dir entries for . and .. unless the disk is completely full.
*/
  /* Use dot1 and dot2, so the mode of the directory isn't important. */
  rip->i_mode = bits;   /* set mode */
  r1 = search_dir(rip, dot1, &dot, ENTER);      /* enter . in the new dir
*/
  r2 = search_dir(rip, dot2, &dotdot, ENTER);   /* enter .. in the new dir
*/

  /* If both . and .. were successfully entered, increment the link counts.
*/
  if (r1 == OK && r2 == OK) {
      /* Normal case.  It was possible to enter . and .. in the new dir. */
      rip->i_nlinks++;  /* this accounts for . */
      ldirp->i_nlinks++;      /* this accounts for .. */
      ldirp->i_dirt = DIRTY;  /* mark parent's inode as dirty */
  } else {
      /* It was not possible to enter . or .. probably disk was full. */
      (void) search_dir(ldirp, string, (ino_t *) 0, DELETE);
      rip->i_nlinks--;  /* undo the increment done in new_node() */
  }
  rip->i_dirt = DIRTY;        /* either way, i_nlinks has changed */

  put_inode(ldirp);           /* return the inode of the parent dir */
  put_inode(rip);       /* return the inode of the newly made dir */
  return(err_code);           /* new_node() always sets 'err_code' */
}
```

3.7 Diagrama de llamadas



3.8. Código de las librerías

En este apartado vamos a mostrar el código de las librerías, el cual se encarga de construir los mensajes apropiados que se pasan al sistema de ficheros para que realice la tarea en concreto que se ha solicitado. Como ejemplo mostramos el código de las librerías para las llamadas 'open', 'creat' y 'close'. El resto son muy similares.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

                  src/lib/posix/_open.c

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
42800 #include <lib.h>
42801 #define open    _open
42802 #include <fcntl.h>
42803 #include <stdarg.h>
42804 #include <string.h>
42805
42806 #if _ANSI
42807 PUBLIC int open(const char *name, int flags, ...)
42808 #else
42809 PUBLIC int open(name, flags)
42810 _CONST char *name;
42811 int flags;
42812 #endif
42813 {
42814   va_list argp;
42815   message m;
42816
42817   va_start(argp, flags);
42818   if (flags & O_CREAT) {
42819         m.m1_i1 = strlen(name) + 1;
42820         m.m1_i2 = flags;
42821         m.m1_i3 = va_arg(argp, Mode_t);
42822         m.m1_p1 = (char *) name;
42823   } else {
42824         _loadname(name, &m);
42825         m.m3_i2 = flags;
42826   }
42827   va_end(argp);
42828   return (_syscall(FS, OPEN, &m));
42829 }
```

++

src/lib/posix/_creat.c

++

```
39800 #include <lib.h>
39801 #define creat   _creat
39802 #include <fcntl.h>
39803
39804 PUBLIC int creat(name, mode)
39805 _CONST char *name;
39806 Mode_t mode;
39807 {
39808   message m;
39809
39810   m.m3_i2 = mode;
39811   _loadname(name, &m);
39812   return(_syscall(FS, CREAT, &m));
39813 }
```

++

src/lib/posix/_close.c

++

```
39600 #include <lib.h>
39601 #define close   _close
39602 #include <unistd.h>
39603
39604 PUBLIC int close(fd)
39605 int fd;
39606 {
39607   message m;
39608
39609   m.m1_i1 = fd;
39610   return(_syscall(FS, CLOSE, &m));
39611 }
```

4. Cuestiones.

1. ¿Por qué no se puede hacer un *lseek* con una tubería?

Las tuberías se emplean para la comunicación entre procesos. Un proceso sólo puede escribir en una tubería cuando ésta está vacía, al igual que sólo puede leer cuando hay algo escrito. La utilidad *lseek* fue pensada para el acceso directo a ficheros, algo que se contrapone a la filosofía de las tuberías.

2. ¿Por qué sólo el usuario puede utilizar la llamada al sistema *mknod*?

Esta restricción se impone ya que se requiere que exista un nodo-i libre y que la trayectoria que se pasa como parámetro sea correcta, y dichas condiciones no se comprueban durante la ejecución de *mknod*.

3. ¿Qué sucede al intentar crear un directorio que ya existe?

Esta circunstancia se detecta cuando, en el procedimiento *mkdir*, se intenta obtener un nuevo nodo-i libre para el directorio (a través de la función *new_node*) y se comprueba que ya existe. Si tal cosa acontece, se devolverá un código de error.

4. ¿Qué peculiaridades existen al abrir o crear un fichero especial y un directorio?

Los ficheros especiales simulan un dispositivo de E/S, de modo que en la apertura se llama al manejador de dispositivo correspondiente (a través de *dev_open*). Los directorios se usan para llevar el control de los archivos y sólo pueden ser leídos, nunca escritos.

5. A la hora de crear un nuevo fichero, hay que actualizar la entrada del nodo-i conseguido en disco y crear la entrada en el directorio correspondiente. ¿Cuál de las dos operaciones es más conveniente realizar en primer lugar?

En previsión de fallos, siempre es mejor tener un nodo-i marcado como ocupado en disco pero sin entrada de directorio que una entrada de directorio que no tiene nodo-i al que dirigirse. Esto es una forma sutil de decir que en primer lugar deberíamos marcar el nodo-i como ocupado.