READ.c

•*Autores:*
• *J. Adal Vega Pérez*
• *Mario Rodríguez Hernández*
•*© Universidad de Las Palmas de Gran Canaria*

INTRODUCCIÓN. 3

FICHEROS ESPECIALES 5

A) De Caracteres: 6
B) De Bloque: 6

PIPES (INTERCONEXIONES) 6

EL PROGRAMA READ.C 7

PROCEDIMIENTOS QUE UTILIZA READ.C: 7
VARIABLES EXTERNAS: 7
ESTRUCTURAS DE DATOS: 7
PROCEDIMIENTOS EXTERNOS UTILIZADOS 8
EL PROGRAMA FUENTE 8

DO_READ 9

PROCEDIMIENTO LLAMADO: 9

READ_WRITE 9

PARÁMETROS 9
PROCEDIMIENTOS LLAMADOS 9
VARIABLES IMPORTANTES 10
ALGORITMO 10

RW_CHUNK 16

PARÁMETROS 16
PROCEDIMIENTOS LLAMADOS 16
VARIABLES IMPORTANTES 16
ALGORITMO 16

READ_MAP 19

PARÁMETROS 19
PROCEDIMIENTOS LLAMADOS 19
VARIABLES IMPORTANTES 19
ALGORITMO 19

RD_INDIR 21

PARÁMETROS 21
PROCEDIMIENTOS LLAMADOS 21
VARIABLES IMPORTANTES 22

READ_AHEAD 22

PROCEDIMIENTOS LLAMADOS 23
VARIABLES IMPORTANTES 23
ALGORITMO: 23

RAHEAD 23

PARÁMETROS 23
PROCEDIMIENTOS LLAMADOS 24
VARIABLES IMPORTANTES 24
ALGORITMO: 24

CUESTIONES 27

FIGURA 1. 1

FIGURA 2 29

LIBRERÍA 30

INTRODUCCIÓN.

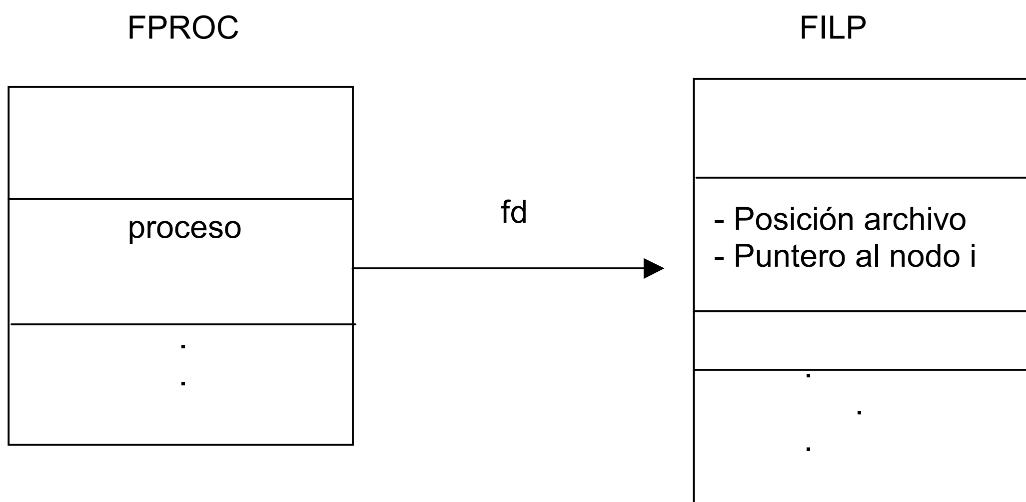
La mayor parte del código del sistema de archivo se dedica a las llamadas al sistema (OPEN, READ, WRITE, PIPE, etc.). El programa READ.C contiene los procedimientos que gestionan la llamada al sistema READ y la llamada al sistema WRITE. Tanto **do_read** como **do_write** llaman a un procedimiento común **read_write** que es el que realiza la mayor parte del trabajo. Para poder entender mejor *READ.C* comenzaremos mostrando un ejemplo que contempla el funcionamiento de la llamada al sistema READ realizada por un proceso de usuario :

1º.- Cuando se abre un archivo (llamada **open**), se especifica un código(0, 1 ó 2) para indicar si se va a abrir para lectura, escritura o ambas y la llamada devuelve un descriptor de archivo (fd) para utilizarlo en las llamadas al sistema para archivos.

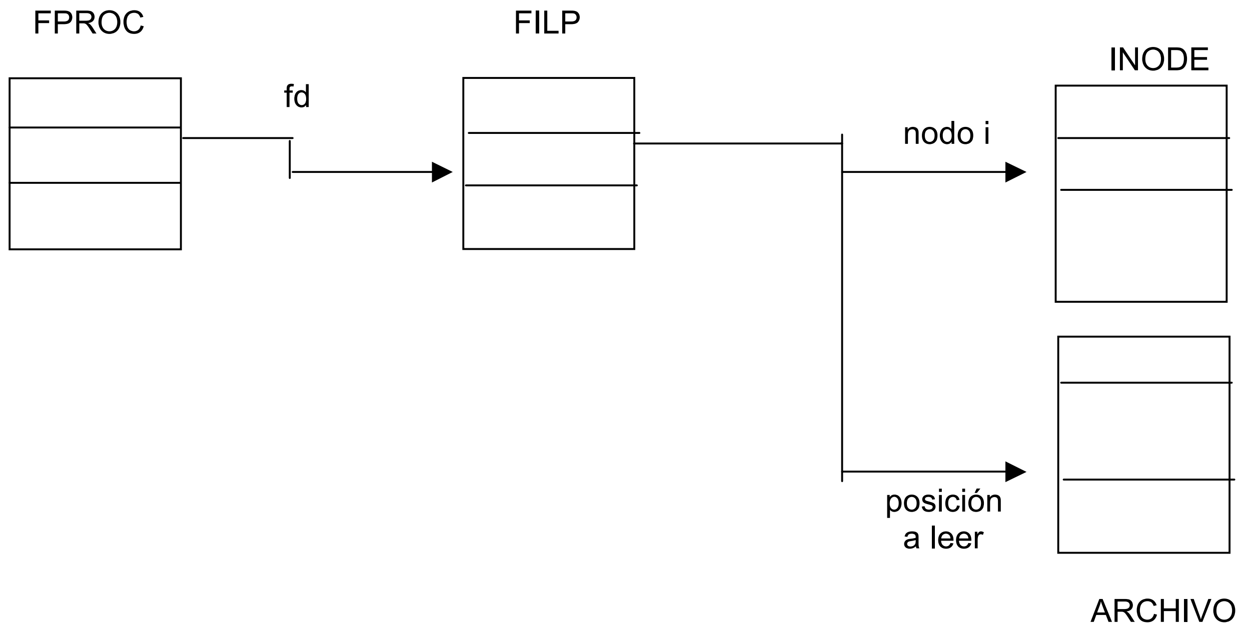
2º.- Cuando un programa de usuario ejecuta la llamada al sistema **read(fd,buffer,nbytes)** para leer un archivo ordinario, se llama a la librería **_read.c** que a su vez realiza un **_syscall** (que se encuentra en *PROC.C*) el cual se encarga de construir un mensaje de tipo READ con tres parámetros:
descriptor de fichero
dirección del buffer de usuario
número de bytes a transferir
que envía al FS. El proceso de usuario se bloquea esperando la respuesta del FS.

3º.- El **main** del FS usa el tipo de mensaje como índice en el vector **call_vec** para llamar el procedimiento que se encarga de la operación de lectura: **do_read**.

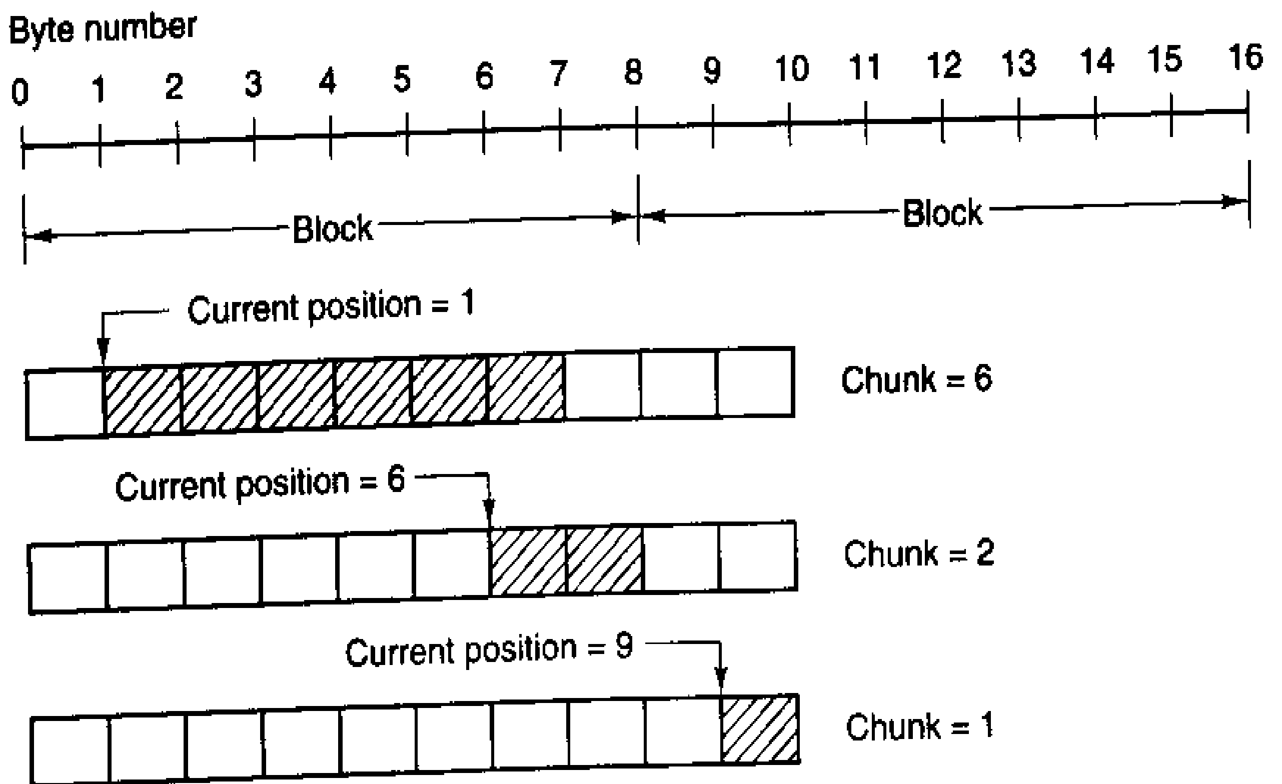
4º.- Después, este procedimiento extrae el descriptor de archivo (fd) del mensaje y lo usa para determinar cuál es la entrada que corresponde a dicho descriptor en la tabla filp (es el procedimiento **get_filp**, que se encuentra en *FILEDES.C*, el que determina la asociación entre descriptor de fichero y ranura de la tabla filp). La situación que tenemos hasta este momento se puede representar así:



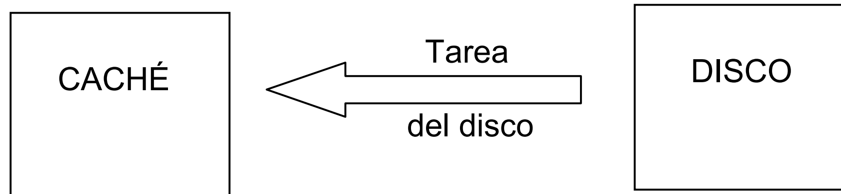
5º.- Una vez encontrada la ranura en la tabla filp se determinará la posición a leer dentro del archivo y el puntero al nodo i correspondiente. Gráficamente:



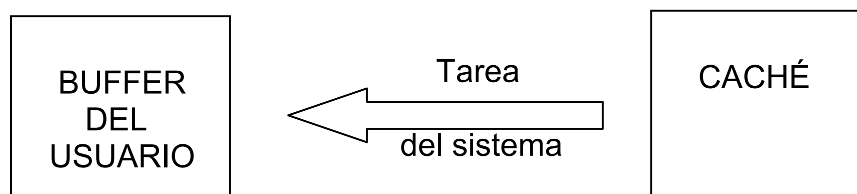
6º.- La solicitud se divide en partes (chunks) de manera que cada una de ellas quepa en un bloque.



7º.- Para cada una de estás partes se comprueba primero si el bloque está en la caché, si no lo está el sistema de archivo elige el buffer menos recientemente usado (LRU), lo reclama para su uso y envía un mensaje a la tarea del disco para reescribirlo si ha sido modificado ("sucio"). Por último, se pide a la tarea del disco (el driver del dispositivo en general) que ponga el bloque leído de disco en la caché.



8 º.- Una vez que el bloque (los trozos que lo forman) esté en la caché, el sistema de archivo emite un mensaje a la tarea del sistema para que copie los datos en el buffer de usuario, colocando los trozos en el orden correcto.



9º.- Al final, el sistema de archivo manda un mensaje de respuesta al usuario especificando cuantos bytes se han copiado. Cuando la contestación llega al usuario, la función de biblioteca read extrae el código de respuesta y lo devuelve, como valor de retorno de dicha función, al proceso solicitante.

Las llamadas más utilizadas son **read** y **write**, que leen o escriben, de forma secuencial, a partir de la posición actual almacenada en un puntero asociado al archivo que indica el siguiente byte a transferir. Cuando se desea acceder de forma aleatoria a una posición del archivo se usa la llamada al sistema **lseek** que cambia el valor de ese puntero a la nueva posición. Las llamadas read o write posteriores podrán entonces podrán acceder a la nueva posición que incluso puede estar más allá del final del archivo.

FICHEROS ESPECIALES

Se usan para acceder a los dispositivos de E/S (/dev/hd0, etc) como si fuesen ficheros de tal forma que para leer o escribir en ellos se usa su fichero especial asociado teniendo en cuenta los bits de modo rwx.

Todos los ficheros especiales tienen, en su **nodo i**, un número de dispositivo mayor (tipo de dispositivo) que serán atendidos por el mismo manejador y uno del menor(el dispositivo concreto) que se le pasa como parámetro al **driver de dispositivos** para que sepa cuál debe leer o escribir. Así, cuando se realiza una lectura de un fichero especial, el FS extrae dichos números del nodo i para saber a qué driver debe llamar y le envía a éste un mensaje con:
el número de dispositivo menor, la operación, buffer y número de bytes a transferir.

Se pueden distinguir dos tipos:

A) De Caracteres:

De flujo de caracteres (terminales, impresoras, etc). Los datos leídos de estos dispositivos no pasan por la Caché.

B) De Bloque:

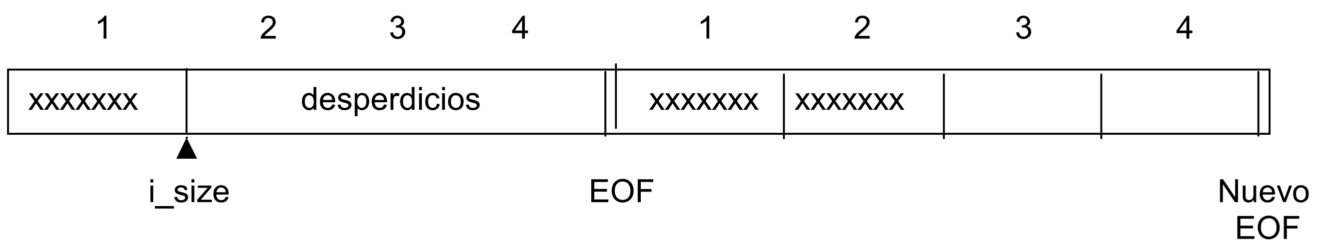
Formados por bloques direccionables de tamaño fijo y elegidos al azar(discos,etc) de forma que un programa puede leer el bloque n de un disco sin tener en cuenta el sistema de archivos almacenado en él. Los datos leídos se almacenan primero en la Caché.

En el caso de la escritura en un fichero no especial de bloques, cuando el tamaño de una zona sea múltiplo del de bloque, hay que tener en cuenta además el siguiente problema:

Supongamos que tenemos zonas de 4k con bloques de 1k y un archivo de tamaño 1k. Entonces la zona 1 se asigna a dicho archivo, quedando el resto de la zona con "desperdicios" (residuos del propietario anterior). Esto no es ningún problema en la **lectura** porque en el nodo i se marca el tamaño de ese archivo como 1k y las lecturas que sobrepasen el final de un archivo siempre producen un conteo igual a cero y sin datos.

Si posteriormente, por ejemplo, **se escribe** un byte en una posición posterior al final del archivo, (perteneciente a la zona siguiente), se actualiza el tamaño de éste incluyendo los bloques con desperdicios. En consecuencia, puede existir una violación de seguridad cuando se lean los datos contenidos en los bloques anteriores pertenecientes a la zona 1.

Aunque este problema no es muy frecuente y no ocurre si el tamaño de una zona es igual al de bloque, este procedimiento lo soluciona verificando, cuando se hace una escritura, si se sobrepasa el final del archivo y en ese caso pone a cero explícitamente todos los bloques aún no asignados en la zona que antes era la última (**clear_zone)**.



PIPES (INTERCONEXIONES)

Son pseudoarchivos (tienen nodo i asociado) que se pueden usar para conectar 2 procesos A y B de forma que el A (**escritor**), para enviar datos al B los escribe en el pipe como si fuera un archivo de salida. Y el B (**lector**) puede leerlos del pipe como si fuera un archivo de entrada. Es decir, la comunicación entre los procesos se realiza de forma parecida a escrituras y lecturas de archivos, no igual, ya que por ejemplo, el proceso que intenta leer de un pipe debe esperar si no existe ningún escritor de datos sobre el mismo.

Por tanto la particularidad de la gestión de los pipes es que :

1º. El FS debe verificar el estado del pipe para ver si la operación se puede completar.

2º Si no es posible, guarda los parámetros de la llamada en la tabla de procesos y no envía la contestación al proceso de usuario para que éste se bloquee hasta que pueda servir su solicitud.

3º Cuando otro proceso modifique el estado del pipe, el FS lo detecta para servir la llamada pendiente.

EL PROGRAMA READ.C

Este fichero contiene la base del mecanismo que se usa para **leer y escribir** ficheros. Las peticiones de lectura y escritura se dividen en trozos (**chunks**) que no sobrepasen el tamaño de bloque y se procesa cada uno de ellos.

PROCEDIMIENTOS QUE UTILIZA READ.C:

do_read
read_write
rw_chunk
read_map
rd_indir
read_ahead
rahead

VARIABLES EXTERNAS:

who	Número del proceso solicitante.
fd	Descriptor del fichero.
nbytes	Número de bytes a transferir.
buffer	Dirección del buffer de usuario.
rdahed_inode	Puntero al nodo i para seguir la lectura (lectura anticipada).
rdahedpos	Posición desde la cual se seguirá la lectura anticipada.

ESTRUCTURAS DE DATOS:

Se indican sólo los campos utilizados en este programa.

fproc	Tabla de procesos del FS
filp	(FILE.H): filp_mode, filp_pos, filp_flags y filp_ino
inode	(INODE.H) (INODE.C): _size, i_mode, i_zone[0], i_dev, i_pipe, i_update, i_dirt, i_seek, i_ndzones, i_nindirs, i_sp
super_block	(SUPER.H) (SUPER.C): s_max_size, s_zones, s_firstdatazone, s_long_zone_size, s_version, s_native
buf	(BUF.H) (CACHÉ.C):

b_data, b_dirt, b_ind

PROCEDIMIENTOS EXTERNOS UTILIZADOS

DEVICE.C:

dev_io	Se llama cuando se necesita acceder al dispositivo de E/S real. Sólo es llamado por **read_write** para transferencia con archivos especiales de caracteres y **rw_chunk** para archivos especiales de bloque.

FILEDES.C :

get_filp	Comprueba el descriptor de archivo y devuelve un puntero a la tabla **filp**.
find_filp	Busca en dicha tabla una entrada correspondiente a un nodo i con una combinación de bits de modo rwx determinada (se usa para comprobar los pipes).

SUPER.C :

get_super	Devuelve el superbloque de un dispositivo.

WRITE.C :

clear_zone	Pone a cero los bloques de una zona a partir de una posición.
zero_block	Anula un bloque.
new_block	Adquiere un nuevo bloque .

PIPE.C :

pipe_check	Comprueba el estado de un pipe.
suspend:	Suspende al proceso llamador

CACHÉ.C :

get_block	Captura un bloque para leer o escribir en la Caché y llama a **rw_block** que lee o escribe un bloque de disco
put_block	Devuelve un bloque anteriormente solicitado con **get_block**.
rw_scattered	Lectura/escritura múltiple

EL PROGRAMA FUENTE

```
#include "fs.h"
#include <fcntl.h>
#include <minix/com.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

#define FD_MASK      077   /* El mayor nº de descriptor de fichero es 63 */

PRIVATE message umess;  /* Mensaje para pedir a SYSTASK la copia del usuario */

FORWARD _PROTOTYPE( int rw_chunk, (struct inode *rip, off_t position,
               unsigned off, int chunk, unsigned left, int rw_flag,
```

```
                char *buff, int seg, int usr) );
```

DO_READ

Este procedimiento ejecuta la llamada al sistema **READ** llamando a read_write indicando con el flag **READING** (valor 0) que la operación a realizar es de lectura.

PROCEDIMIENTO LLAMADO:

read_write : Ejecuta la operación de lectura o escritura.

```
/*===================================================================*
 *                   do_read                                         *
 *===================================================================*/

PUBLIC int do_read()
{
  return(read_write(READING));
}
```

READ_WRITE

Ejecuta las operaciones correspondientes a las llamadas **read**(fd, buffer, nbytes) o **write**(fd, buffer, nbytes). Para realizar dichas operaciones, se dividen los bytes a transferir en **chunks** y se llama al procedimiento **rw_chunk** que será el encargado de leer o escribir cada trozo. Un chunk comienza en la posición actual y se extiende hasta que:

a) se han leído todos los bytes o
b) se llega al límite de un bloque o
c) se activa el EOF.
De está forma un chunk nunca usa 2 bloques de disco.

Se tiene en cuenta también las particularidades de la gestión de los diferentes tipos de fichero: ordinario(regular), directorio, pipe, especial de caracteres y especial de bloque.

PARÁMETROS

rw_flag : Indica el tipo de operación (lectura o escritura)

PROCEDIMIENTOS LLAMADOS

get_filp: Mira si el **fd** es válido, si no lo es devuelve **NIL_FILP** y si lo es, devuelve la entrada en la tabla filp para ese descriptor de fichero.

dev_io: Lee o escribe caracteres en un dispositivo.
clear_zone: Elimina bloques que, en un momento dado, están en la mitad del archivo cuando se hace una localización a una posición más allá del fin de fichero, seguida de una escritura de datos en esa posición.
pipe_check: Verifica si es posible realizar la operación sobre un pipe.
find_filp: Encuentra una entrada en la tabla filp de modo que se refiera al nodo i indicado en el parámetro **rip** y que tenga definido el modo que indica **bits**.
rw_chunk: Lee o escribe un chunk.
suspend: Suspende al proceso llamador

VARIABLES IMPORTANTES

position: nº de 32 bits que indica el siguiente byte a leer o escribir
oflags: bits de modo(rwx)
rip: puntero al nodo i
f_size: tamaño del fichero
cum_io: acumulador del nº de bytes transferidos
r: devuelve el resultado de la operación
off: offset dentro de un bloque
op: variable que indica si se realizará una lectura o escritura
mode_word:palabra de modo de acceso al fichero (tipo, protección,..)
regular: valor 1 si es un fichero regular o pipe
char_espec: valor 1 si el fichero es especial de caracteres
block_espec : valor 1 si el fichero es espacial de bloques
rdwt_err: error de lectura o escritura en el dispositivo
chunk: trozo de tamaño que no sobrepasa un bloque
bytes_left: bytes restantes hasta el EOF
wf: puntero a entradas en filp para escritores en pipe
partial_pipe con valor 1 si el fichero a tratar es un pipe
partial_cnt nº de bytes que se encuentran disponibles en el pipe para transferir

ALGORITMO

1º Determina los bits de usuario(**usr)** y de segmento(**seg**) del proceso solicitante.

COMPROBACIONES INICIALES:

2º SI número de bytes a transferir < 0 ENTONCES Error
3º SI fd valido (**get_filp**) ENTONCES
f = puntero a la entrada en la tabla filp (nodo i)
SINO Error

4º SI Operación es lectura y fichero abierto para escritura solo (bits de modo) ENTONCES
Error.

INICIALIZACIÓN DE VARIABLES:

5º Obtener de la tabla filp la posición en el fichero y el puntero al nodo i, y obtener del nodo i el tamaño del fichero y el tipo de fichero (regular, especial).

SI posición no está dentro de los límites (tamaño máximo de fichero) ENTONCES Error

COMPROBACIONES SEGÚN EL TIPO DE FICHERO

6º SI es archivo especial de caracteres (no se usa la Caché) ENTONCES
Llama a **dev_io** para leer o escribir caracteres en un dispositivo
SINO
TEST PARA ESCRITURAS

SI operación de **escritura** (no en dispositivo de bloques) ENTONCES
SI posición > tamaño máximo de fichero - nbytes (el fichero se hará demasiado grande) ENTONCES Error

SI Escritura es más allá del EOF (Append) ENTONCES
Limpia la zona que contiene el EOF si se va a crear un hueco para que todos los bloques no escritos antes de EOF se lean como 0. El EOF está al final de la zona asignada.(**clear_zone)**
SI es un **pipe** (campo pipe del nodo i a 1) y no es posible realizar operación sobre él (**pipe_check**) ENTONCES
Retorna resultado del test

LA TRANSFERENCIA (es el núcleo central para ficheros ordinarios)

MIENTRAS nbytes <> 0 HACER
Calcular el offset dentro de1 bloque
Elegir el tamaño del chunk
SI es **lectura** ENTONCES
Calcular los bytes pendientes.
SI posición > tamaño del fichero (más allá del EOF) ENTONCES
Salir del bucle
SI chunk > bytes pendientes ENTONCES
chunk = bytes pendientes

Se transfiere un chunk (**rw_chunk**)
SI se ha alcanzado EOF ENTONCES
Salir del bucle
SI existe Error de lectura o escritura en el dispositivo ENTONCES
Salir del bucle
Actualizar punteros y contadores (de buffer de usuario, número de bytes y posición)
FIN MIENTRAS
FIN SI

TEST POSTERIOR A LA TRANSFERENCIA

7º SI es **escritura** ENTONCES
SI es fichero ordinario o Directorio ENTONCES
Actualizar posición dentro del fichero
SINO (es **lectura**)
SI es un pipe y se ha llegado al final (posición > tamaño) ENTONCES
Reinicializar a 0 los punteros del pipe (tamaño y posición)

SI hay algún escritor pendiente al final del pipe (**find_filp**) ENTONCES
Poner a 0 la posición del fichero para esa entrada en filp
FIN SI

8º Actualizar la Posición en la tabla filp

9º SI se solicitó operación de **lectura anticipada** (en acceso secuencial sólo: i_seek=**NO_SEEK**) en ficheros ordinarios o directorios ENTONCES
Almacenar en variables globales el nodo i y la posición del fichero de la cual se leerá para que, después de mandar mensaje de respuesta al usuario, el sistema pueda empezar a trabajar para obtener el siguiente bloque. Si el FS se bloquea esperando el siguiente bloque el usuario puede trabajar mientras con los datos ya leídos.

11º Se chequea si error en disco

10º SI operación bien realizada ENTONCES
Actualizar fecha de acceso (poner marca **mtime** para actualizarlo más tarde por medio de **clock_time**(UTILITY.C)) en el nodo i
Marca bloque como "sucio" en el nodo i, para posteriormente rescribirlo en disco.
SI es un pipe ENTONCES
fp->fp_cum_io_partial = cum_io
se suspende al llamador
retorna cero
FIN SI
fp->fp_cum_io_partial = cum_io
retorna el nº de bytes transferidos
FIN SI

```
/*===========================================================================*
 *                         read_write                                        *

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

PUBLIC int read_write(rw_flag)
int rw_flag;                    /* LECTURA o ESCRITURA */
{

  register struct inode *rip;
  register struct filp *f;
  off_t bytes_left, f_size, position;
  unsigned int off, cum_io;
  int op, oflags, r, chunk, usr, seg, block_spec, char_spec;
  int regular, partial_pipe = 0, partial_cnt = 0;
  dev_t dev;
  mode_t mode_word;
  struct filp *wf;

  /* Si el MM es el solicitante (llamada EXEC) usa este código para hacer que el FS cargue por él los
segmentos completos en el espacio de usuario. El MM coloca disparates en los diez bits más significativos del fd
(entero de 16 bits). */

  if (who == MM_PROC_NR && (fd & (~BYTE)) ) {
     usr = (fd >> 8) & BYTE; /*  bits de usuario */
```

```
      seg = (fd >> 6) & 03; /* bits de segmento */
fd &= FD_MASK;       /* se libera de los bits de usuario y de segmento (FD_MASK es el nº máximo de
fd */
  } else {   /* caso normal */
      usr = who;          /* nº de proceso solicitante */
      seg = D;
  }

  /* COMPROBACIONES INICIALES */

  /* Si el descriptor de fichero y la operación a realizar son válidos, determina el nodo i, el tamaño y el
modo. */
  if (nbytes < 0) return(EINVAL);
  if ((f = get_filp(fd)) == NIL_FILP) return(err_code);

  /* Bits de acceso : R_BIT = Rwx y W_BIT = rWx */
  if (((f->filp_mode) & (rw_flag == READING ? R_BIT : W_BIT)) == 0) {
     return(f->filp_mode == FILP_CLOSED ? EIO : EBADF);
  }
  if (nbytes == 0) return(0); /* Así los archivos especiales de caracteres no necesitan
chequear el  0 */

  /* INICIALIZACIÓN DE VARIABLES */

  position = f->filp_pos;
  if (position > MAX_FILE_POS) return(EINVAL);
  if (position + nbytes < position) return(EINVAL); /* unsigned overflow
*/
  oflags = f->filp_flags;
  rip = f->filp_ino;
  f_size = rip->i_size;
  r = OK;

  if (rip->i_pipe == I_PIPE) {
     /* fp->fp_cum_io_partial is only nonzero when doing partial writes */
     cum_io = fp->fp_cum_io_partial;
  } else {
     cum_io = 0; /* nº de bytes transferidos */
  }
  op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
  mode_word = rip->i_mode & I_TYPE; /*determina el tipo usando los bits de señal del nodo
i*/

  regular = mode_word == I_REGULAR || mode_word == I_NAMED_PIPE;

  char_spec = (mode_word == I_CHAR_SPECIAL ? 1 : 0);
  block_spec = (mode_word == I_BLOCK_SPECIAL ? 1 : 0);
  if (block_spec) f_size = LONG_MAX;
  rdwt_err = OK;        /* Error de E/S si ocurre fallo de disco*/

  /* COMPROBACIONES SEGÚN EL TIPO DE FICHERO */

  /*Verificación para los ficheros especiales de caracteres. */
  if (char_spec) {
```

```
      dev = (dev_t) rip->i_zone[0];
      r = dev_io(op, oflags & O_NONBLOCK, dev, position,nbytes,
who,buffer);
      if (r >= 0) {
            cum_io = r;       /* en r está el nº de bytes transferidos */
            position += r;
            r = OK;
      }
   } else {

   /* TEST PARA ESCRITURAS */

      /* si se escribe en archivo que no sea especial de bloque*/
      if (rw_flag == WRITING && block_spec == 0) {
      /* Se verifica por adelantado si el fichero se va a hacer más grande de lo que admite el dispositivo.
      */
            if (position > rip->i_sp->s_max_size - nbytes) return(EFBIG);
            /* chequea el flag O_APPEND (añadir al final del fichero) */
            if (oflags & O_APPEND) position = f_size;

            /* La escritura va a ser más allá del EOF y se creará un hueco en dicha zona. Entonces se
            limpia la zona con el EOF actual ya que todos los bloques no escritos anteriores a ese EOF
            deben leerse como ceros en lecturas posteriores.*/
            if (position > f_size) clear_zone(rip, f_size, 0);
      }

      /* Los Pipes son un poco diferentes, en ellos se verifica si se puede realizar la operación */
      if (rip->i_pipe == I_PIPE) {
             r =
pipe_check(rip,rw_flag,oflags,nbytes,position,&partial_cnt);
             if (r <= 0) return(r);
      }

      if (partial_cnt > 0) partial_pipe = 1;

   /* LA TRANSFERENCIA */

      /* Dividir la transferencia en trozos (chunks) que no abarquen más allá de un bloque de disco */
      while (nbytes != 0) {
            off=(unsigned int) (position % BLOCK_SIZE); /*offset dentro de un
bloque*/
            if (partial_pipe) {   /* sólo en caso de pipes */
                  chunk = MIN(partial_cnt, BLOCK_SIZE - off);
            } else
                  chunk = MIN(nbytes, BLOCK_SIZE - off);
            if (chunk < 0) chunk = BLOCK_SIZE - off;

            if (rw_flag == READING) {
                  bytes_left = f_size - position;
                  if (position >= f_size) break; /* Estamos más allá del EOF */
                  if (chunk > bytes_left) chunk = (int) bytes_left;
            }

            /* Lee o escribe los bytes de un trozo (chunk), es decir, parte de un bloque. */
```

```
          r = rw_chunk(rip, position, off, chunk, (unsigned) nbytes,
                   rw_flag, buffer, seg, usr);
          if (r != OK) break;  /* Se alcanzó el EOF */
          if (rdwt_err < 0) break;

          /* actualizar contadores y punteros */
          buffer += chunk;       /* dirección del buffer de usuario */
          nbytes -= chunk;       /* bytes por leer */
          cum_io += chunk;       /* bytes leídos */
          position += chunk;     /* posición en el fichero */

          if (partial_pipe) {
               partial_cnt -= chunk;
               if (partial_cnt <= 0)  break;
          }
     } /* fin mientras */
  }

  /* TEST POSTERIOR A LA TRANSFERENCIA */

  /* Si es Escritura, actualiza el tamaño del fichero (la posición) */
  if (rw_flag == WRITING) {
     if (regular || mode_word == I_DIRECTORY) {
          if (position > f_size) rip->i_size = position;
     }
  } else { /* Si es Lectura en pipe, resetea o recoloca los punteros al pipe */
     if (rip->i_pipe == I_PIPE && position >= rip->i_size) {
          rip->i_size = 0;       /* no quedan datos */
          position = 0;          /* resetea lector(es) */
          /* Si existe algún escritor para ese pipe encuentra su entrada en filp */
          if ( (wf = find_filp(rip, W_BIT)) != NIL_FILP) wf->filp_pos =0;
     }
  }
  f->filp_pos = position;

  /*   Verifica si se ha solicitado una lectura anticipada y si es así la establece. */
  if (rw_flag == READING && rip->i_seek == NO_SEEK && position %
BLOCK_SIZE== 0 && (regular || mode_word == I_DIRECTORY)) {
     rdahed_inode = rip;      /* nodo i del que se leerá */
     rdahedpos = position;      /* posición de la que se leerá */
  }
  rip->i_seek = NO_SEEK;

  if (rdwt_err != OK) r = rdwt_err; /* chequea error de disco */
  if (rdwt_err == END_OF_FILE) r = OK;
  if (r == OK) {
     if (rw_flag == READING) rip->i_update |= ATIME;
     if (rw_flag == WRITING) rip->i_update |= CTIME | MTIME;
     rip->i_dirt = DIRTY;       /* se marca al inode como modificado */
     if (partial_pipe) {
          partial_pipe = 0;
               /* partial write on pipe with */
               /* O_NONBLOCK, return write count */
          if (!(oflags & O_NONBLOCK)) {
```

```
                fp->fp_cum_io_partial = cum_io;
                suspend(XPIPE); /* partial write on pipe with */
                return(0);/* nbyte > PIPE_SIZE - non-atomic */
            }
        }
        fp->fp_cum_io_partial = 0;
        return(cum_io);
    } else {
        return(r);
    }
}
```

RW_CHUNK

Lee o escribe un chunk. Para ello, toma el nodo i y una posición del fichero, los convierte (**read_map**) en un nº de bloque de disco físico, lo solicita a la Caché y por último solicita la transferencia de ese bloque (o una parte de él) a o desde el espacio de usuario.

PARÁMETROS

rip, position off, chunk, left, rw_flag, buff, seg y usr

PROCEDIMIENTOS LLAMADOS

read_map: Convierte una posición de archivo lógica al número de bloque físico.
zero_block: Anula un bloque, poniéndolo todo a cero.
new_block: Adquiere un nuevo bloque y devuelve su puntero.
rahead: Realiza lectura anticipada de bloque.
get_block: Verifica si el bloque solicitado está en la Caché. Si es así, devuelve un puntero a éste y si no expulsa algún otro bloque y captura el solicitado.
sys_copy: Es una llamada a la tarea del sistema para realizar la copia física.
put_block: Devuelve un bloque a la lista de bloques disponibles.

VARIABLES IMPORTANTES

b: bloque físico
dev: dispositivo
bp: puntero al buffer de Caché
r: nº de bytes procesados
n: normal o no_read

ALGORITMO

1º. Calcula número de bloque físico (**b**) y número de dispositivo.
2º. SI se trata de un bloque que no existe ENTONCES
    SI es **lectura** ENTONCES
        Lectura de bloque inexistente
        Pide al manejador de la caché que le halle el bloque (**get_block**).
        Pone el bloque a cero (**zero_block**).
    SINO
        Escritura de bloque inexistente
        Adquiere un bloque nuevo y en el caso de que no existan bloques disponibles devuelve Error. (**new_block**).
    FIN SI

SINO
SI es **lectura** ENTONCES
Realiza lectura y lectura adelantada (**rahead**)
SINO (es escritura)
Si el bloque existente va a ser parcialmente reescrito, se lee primero y en el caso de que vaya a ser reescrito totalmente no necesita leerse.
Mira si el bloque ya está en la caché y si no lo busca en disco y devuelve el puntero al bloque bp (**get_block**).
FIN SI
FIN SI

3º. Ahora **bp** apunta al buffer válido. Si es **escritura**, estamos más allá del tamaño en bytes del fichero y no es dispositivo especial de bloques, el bloque al que apunta "bp" se pone a ceros (**zero_block**).

4º. SI es **lectura** ENTONCES
copiar el chunk desde el buffer de caché al espacio de usuario **(sys_copy)**
SINO (es escritura)
copiar el chunk desde el espacio de usuario al buffer y marcar el buffer como modificado.

5º. Devuelve bloque a la lista de bloques disponibles (**put_block**) para que pueda ser "expulsado" por la caché.

6º. Devuelve número de bytes transferidos.

```
/*===========================================================================*
 *                        rw_chunk                                           *
*===========================================================================*
/

PRIVATE int rw_chunk(rip,position, off,chunk, left,rw_flag, buff, seg, usr)

register struct inode *rip; /* puntero al nodo i del fichero que se va a leer o escribir */
off_t position;             /* posición dentro del fichero a ser leída o escrita */
unsigned off;               /* offset dentro del bloque actual */
int chunk;                  /* número de bytes a leer o escribir*/
unsigned left;              /* máximo número de bytes necesitados después de una posición */
int rw_flag;                /* READING o WRITING */
char *buff;                 /* dirección virtual del buffer de usuario */
int seg;                    /* segmentos T o D en el espacio de usuario */
int usr;                    /* el proceso de usuario */
{
/* Lee o escribe (parte de) un bloque */

  register struct buf *bp;
  register int r;
  int n, block_spec;
  block_t b;
  dev_t dev;
```

```
  block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
 /* calcula nº de bloque y dispositivo */
  if (block_spec) {
     b = position/BLOCK_SIZE;
     dev = (dev_t) rip->i_zone[0];
  } else {
     b = read_map(rip, position);
     dev = rip->i_dev;
  }

  if (!block_spec && b == NO_BLOCK) {
     if (rw_flag == READING) {
          /* Lectura de un bloque que no existe (NO_BLOCK): debe leer todos ceros */
          bp = get_block(NO_DEV, NO_BLOCK, NORMAL);   /* obtiene el buffer */
          zero_block(bp);
     } else {
        /* Escritura a un bloque inexistente: crea bloque y lo introduce en el nodo i. Si no existe Error
*/
          if ((bp= new_block(rip, position)) == NIL_BUF)return(err_code);
     }
  } else if (rw_flag == READING) {
     /* Lectura y lectura adelantada si corresponde.*/
     bp = rahead(rip, b, position, left);
  } else {  /* escritura */

     /* Normalmente, un bloque existente que va a ser parcialmente sobreescrito se debe leer primero
(NORMAL), salvo si se va a reescribir el bloque entero, en cuyo caso no necesita leerse (NO_READ). Si
ya está en la Caché, obtenerlo; si no conseguir un buffer libre para traer el bloque en cuestión desde el
dispositivo.*/
     n = (chunk == BLOCK_SIZE ? NO_READ : NORMAL);
     if (!block_spec && off == 0 && position >= rip->i_size) n = NO_READ;

     /* Se pide al manejador de la Caché que encuentre el bloque, leyéndolo de disco si es necesario y
almacenando en bp su dirección. Una vez leído el contador de encabezado del bloque indica que está en
uso para que no pueda ser expulsado de la Caché */
     bp = get_block(dev, b, n);
  }

  /* En cualquier caso, bp apunta ahora a un buffer válido. */
  if (rw_flag == WRITING && chunk != BLOCK_SIZE && !block_spec &&
               position >= rip->i_size && off == 0) {
     zero_block(bp);
  }

  if (rw_flag == READING) {
     /* Copia un chunk desde el buffer de bloques (FS) al espacio de usuario */
     r = sys_copy(FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
               usr, seg, (phys_bytes) buff,(phys_bytes) chunk);
  } else {   /*escritura*/
     /* Copia un chunk desde el espacio de usuario al buffer de  bloques*/
     r = sys_copy(usr, seg, (phys_bytes) buff,     FS_PROC_NR, D,
                 (phys_bytes) (bp->b_data+off),(phys_bytes) chunk);
     bp->b_dirt = DIRTY;
```

```
  }

  n = (off + chunk == BLOCK_SIZE ? FULL_DATA_BLOCK : PARTIAL_DATA_BLOCK);
```
/* Libera el bloque para que se pueda ser expulsado de la Caché cuando sea el momento, decrementando el contador de uso del mismo. */

```
  put_block(bp, n);
  return(r);
}
```

READ_MAP

Dado un nodo i y una posición dentro del fichero correspondiente, localiza el número de bloque físico (no de zona), en el que se encontrará dicha posición y lo devuelve.

PARÁMETROS

rip: puntero al nodo i desde donde se mapeará.
position: posición en el fichero cuyo bloque se desea.

PROCEDIMIENTOS LLAMADOS

get_block: Ya comentado.
put_block: Ya comentado.
rd_indir Se verá después.

VARIABLES IMPORTANTES

zone:	zona a la que corresponde un bloque determinado.
bp:	puntero al buffer.
b:	número de bloque absoluto.
boff:	número de bloque relativo dentro de la zona
excess:	contador del número de bloques indirectos.
dzones	nº de zonas directas
nr_indirects	nº de zonas indirectas por bloque indirecto

ALGORITMO

1º. Se calcula el factor de escala.
2º. Calcula el número de bloque relativo dentro de la zona.
3º. SI la zona está en el propio nodo i (su dirección está en las siete primeras zonas) ENTONCES
Se coloca en un bloque absoluto (no relativo a zona)
4º. SI la zona no está en el nodo i(habrá que leer uno o dos bloques indirectos) ENTONCES
excess = zone - dzone
SI **excess** < **NR_INDIRECTS** ENTONCES
La posición se localiza por el bloque indirecto simple
SINO
- La posición se localiza por el bloque indirecto doble.

- Una vez que nos hayamos movido al bloque indirecto doble, la indirección pasa a ser sencilla.
- Obtener buffer que contiene bloque indirecto doble (**get_block**).
- **z** nos indicará la zona del segundo bloque (**rd_indir**).
- Se libera el bloque indirecto doble y excess ahora apunta donde indica el bloque indirecto (**put_block**).

FIN SI

5º. **z** es el número de zona en bloque indirecto simple.
6º. Se halla la posición absoluta del bloque.
7º. Se obtiene el bloque indirecto, se halla la zona y se libera dicho bloque. Se halla la posición absoluta del bloque.
8º. Retorna la posición del bloque.

```
/*===========================================================================*
 *                    read_map                                               *
 *===========================================================================*/

PUBLIC block_t read_map(rip, position)

register struct inode *rip;     /* puntero al nodo i desde donde se mapeará */
off_t position;                 /* posición en el fichero cuyo bloque se desea */
{

  register struct buf *bp;
  register zone_t z;
  int scale, boff, dzones, nr_indirects, index, zind, ex;
  block_t b;
  long excess, zone, block_pos;

  scale = rip->i_sp->s_log_zone_size;     /* Para conversión bloque-zona*/
  block_pos = position/BLOCK_SIZE;   /* Número de bloque relativo en el fichero */
  zone = block_pos >> scale;    /* Zona de la posición, hallada mediante rotación */
  boff = (int) (block_pos - (zone << scale)); /*nº de bloque relativo dentro de la zona
*/
  dzones = rip->i_ndzones;
  nr_indirects = rip->i_nindirs;

  /* Comprueba si la zona está en el propio nodo i (no hay que usar direccionamiento indirecto) */
  if (zone < dzones) {
     zind = (int) zone;  /* index should be an int */
     z = rip->i_zone[zind];
     if (z == NO_ZONE) return(NO_BLOCK);
     b = ((block_t) z << scale) + boff;
     return(b);
  }

  /*   No está en el nodo i, por lo que debe de ser indirecto o doble indirecto. */
  excess = zone - dzones; /* El primer Vx_NR_DZONES no cuenta */

  if (excess < nr_indirects) {
      /* La posición se puede localizar por el bloque indirecto. Se coloca en la zona correspondiente al
bloque indirecto.*/
```

```
      z = rip->i_zone[dzones];
  } else {
```

/* La posición puede ser localizada por el bloque indirecto doble. Se coloca en la zona correspondiente al bloque indirecto doble. */

```
      if ( (z = rip->i_zone[dzones+1]) == NO_ZONE) return(NO_BLOCK);
      excess -= nr_indirects;          /* el indirecto no cuenta */
```

/* Ahora excess indica que al habernos movido al bloque indirecto doble, la indirección pasa a ser sencilla. */

```
      b = (block_t) z << scale;
```

/* Obtener el buffer que contiene el bloque indirecto doble:*/

```
      bp = get_block(rip->i_dev, b, NORMAL);

      index = (int) (excess/nr_indirects);
```

/* z es el número de zona para el indirecto del anterior indirecto doble. Es decir, ya tenemos la zona del segundo bloque índice de la cadena */

```
      z = rd_indir(bp, index);
      put_block(bp, INDIRECT_BLOCK);      /* liberar bloque indirecto doble */
      excess = excess % nr_indirects;     /* índice en le bloque indirecto simple */
  }
```

/* z es el número de zona en el bloque indirecto simple (en cualquier caso, estamos en él) y excess es el índice dentro de él. */

```
  if (z == NO_ZONE) return(NO_BLOCK);
  b = (block_t) z << scale;
  bp = get_block(rip->i_dev, b, NORMAL); /* obtener el bloque indirecto simple */
  ex = (int) excess;
  z = rd_indir(bp, ex);                  /* obtiene un puntero al bloque (zona)*/
  put_block(bp, INDIRECT_BLOCK);         /* liberar el bloque indirecto simple */
  if (z == NO_ZONE) return(NO_BLOCK);
  b = ((block_t) z << scale) + boff;
  return(b);
}
```

RD_INDIR

Este procedimiento es llamado para leer (obtener) un bloque indirecto. Esto es un procedimiento independiente porque los datos pueden estar almacenados de diferentes formatos en el disco dependiendo de la versión del FS y del hardware en el que el FS fue escrito. Entonces la conversión, si es necesaria, se realizará aquí y al resto del FS ve los datos de una sola forma.

PARÁMETROS

bp: puntero al bloque indirecto.
index: índice dentro de dicho bloque.

PROCEDIMIENTOS LLAMADOS

get_super: Busca un dispositivo especificado en la tabla de superbloques y si lo encuentra devuelve puntero al superbloque asociado a ese dispositivo.
conv2 Cambia el orden de los bytes de una palabra de 16 bits.

conv4 Cambia el orden de los bytes de una palabra de 32 bits.

VARIABLES IMPORTANTES

zone: zona a la que corresponde un bloque determinado.
sp: puntero asociado al superbloque del dispositivo.

```
/*===========================================================================*
 *                    rd_indir                                               *
 *===========================================================================*/

PUBLIC zone_t rd_indir(bp, index)

struct buf *bp;        /* puntero a un bloque indirecto */
int index;             /* índice en *bp */
{
/* Dado un puntero a un bloque indirecto, lee una entrada.  La razón de
separar esta rutina del resto del código es que pueden darse cuatro casos
distintos: V1 (IBM and 68000), and V2 (IBM and 68000).  */

  struct super_block *sp;
  zone_t zone;              /* en V2 las zonas son más largas uqe en V1 */

  sp = get_super(bp->b_dev); /* necesita el superbloque para encontrar el tipo de sistema de
fichero */

  /* lee una zona desde un bloque indirecto */
  if (sp->s_version == V1)
     zone = (zone_t) conv2(sp->s_native, (int)  bp->b_v1_ind[index]);
  else
     zone = (zone_t) conv4(sp->s_native, (long) bp->b_v2_ind[index]);

  if (zone != NO_ZONE && zone < (zone_t) sp->s_firstdatazone || zone >=
      sp->s_zones)) {
     printf("Illegal zone number %ld in indirect block, index %d\n",
            (long) zone, index);
     panic("check file system", NO_NUM); /* llamada para abortar el sistema */
  }
  return(zone);
}
```

READ_AHEAD

Convierte la posición lógica en un número de bloque físico, llama a **get_block** para asegurar que el bloque esté en la reserva y después devuelve el bloque de inmediato. Después de todo, no puede hacer nada con el bloque, sólo desea mejorar la oportunidad de que el bloque esté cerca si se llega a necesitar pronto.

Hay que observar que el **read_ahead** sólo se llama del ciclo principal en main. No se le llama como parte del procesamiento de a llamada al sistema READ. Es importante comprender que la llamada a **read_ahead** se efectúa después de que se envía la respuesta, de manera que el usuario podrá continuar la ejecución aun si el sistema de archivo tiene que esperar un bloque del disco mientras hace la lectura anticipada.

PROCEDIMIENTOS LLAMADOS

put_block, read_map: Ya comentados.
rahead: Comentado más adelante.

VARIABLES IMPORTANTES

rip: puntero al nodo i en el que se realiza una lectura anticipada.
bp: puntero al buffer.
b: número de bloque físico para seguir leyendo.

ALGORITMO:

1º. Se determina el puntero al nodo i del cual se realiza la lectura adelantada.
2º. Desactivar la lectura adelantada.
3º. Determina el bloque del nodo i (**read_map**). En el caso de que la lectura adelantada se realizara después del final del fichero, retorna.
5º. Gestiona la lectura adelantada del bloque (**rahead**).
6º. Libera bloque (**put_block**).

```
/*=====================================================================*
 *                   read_ahead                                        *
 
*=====================================================================*/

PUBLIC void read_ahead()
{
/* lee un bloque de la caché antes de que sea necesario */

  register struct inode *rip;
  struct buf *bp;
  block_t b;

  rip = rdahed_inode;      /* puntero al nodo i en el que se realiza una lectura anticipada */
  rdahed_inode = NIL_INODE;     /* desactivar la lectura anticipada*/
  if ( (b = read_map(rip, rdahedpos)) == NO_BLOCK) return;    /* en EOF */
  bp = rahead(rip, b, rdahedpos, BLOCK_SIZE);
  put_block(bp, PARTIAL_DATA_BLOCK);
}
```

RAHEAD

Este procedimiento se encarga de la búsqueda de un bloque desde la caché o desde el dispositivo. Si se requiere lectura física, se hace una prebúsqueda de tantos bloques como convengan en la caché.

PARÁMETROS

rip: puntero al nodo i para el fichero a leer.
baseblock: bloque en la posición actual.
position: posición dentro del fichero.
bytes_ahead: bytes más allá de la posición para uso inmediato.

PROCEDIMIENTOS LLAMADOS

put_block, get_block, read_map: Ya comentados.
rw_scattered: Realiza lectura/escritura múltiple.

VARIABLES IMPORTANTES

block: número de bloque.
blocks_ahead: número de bloques con lectura adelantada.
blocks_left: bloques que faltan para el final del fichero
bp: puntero al buffer.
block_spec: identificador de fichero especial de bloque.
dev: número de dispositivo.
fragment offset dentro de un bloque
read_q: puntero a un array de buffers.
read_q_size: contador de la variable read_q.
scale para la conversión de zona a bloque

ALGORITMO:

1º. Determinar el número de dispositivo.
2º. Pedir bloque a la caché (**get_block**).
3º Cálculo del nº de bloques que se van a adelantar
4º Calcula el nº de bloques que restan hasta el final del fichero
5º Se chequea que el nº de bloques a adelantar cumpla que:
  no sea mayor que NR_IOREQS
  no sea menor que el mínimo nº de bloques definido al principio excepto que sea un acceso no secuencial
  no pasarse del final del fichero
6º. Se obtienen los bloques de caché **(get_block)**
7º. Llamada a **rw_scattered**
8º. Devolver bloque que nos proporcione la caché (**get_block**).

```
/*===========================================================================*
 *                    rahead                                                 *
 *===========================================================================*/

PUBLIC struct buf *rahead(rip, baseblock, position, bytes_ahead)

register struct inode *rip;     /* puntero al nodo i para el fichero a leer */
block_t baseblock;              /* bloque en la posición actual*/
off_t position;                 /* posición dentro del fichero */
unsigned bytes_ahead;           /* bytes más allá de la posición para uso inmediato */
{

/* Mínimo nº de bloques para la prelectura. */
# define BLOCKS_MINIMUM       (NR_BUFS < 50 ? 18 : 32)

  int block_spec, scale, read_q_size;
  unsigned int blocks_ahead, fragment;
  block_t block, blocks_left;
  off_t ind1_pos;
```

```
dev_t dev;
struct buf *bp;
static struct buf *read_q[NR_BUFS];

block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
if (block_spec) {
   dev = (dev_t) rip->i_zone[0];
} else {
   dev = rip->i_dev;
}

block = baseblock;
bp = get_block(dev, block, PREFETCH);
if (bp->b_dev != NO_DEV) return(bp);

fragment = position % BLOCK_SIZE;     /*offset dentro de un bloque*/
position -= fragment;
bytes_ahead += fragment;

blocks_ahead = (bytes_ahead + BLOCK_SIZE - 1) / BLOCK_SIZE;

if (block_spec && rip->i_size == 0) {
   blocks_left = NR_IOREQS;
} else {
   blocks_left = (rip->i_size - position + BLOCK_SIZE - 1) / BLOCK_SIZE;

   /* Go for the first indirect block if we are in its neighborhood. */
   if (!block_spec) {
        scale = rip->i_sp->s_log_zone_size; /* Para conversión zona-bloque*/
        ind1_pos = (off_t) rip->i_ndzones * (BLOCK_SIZE << scale);
        if (position <= ind1_pos && rip->i_size > ind1_pos) {
             blocks_ahead++;
             blocks_left++;
        }
   }
}

/* no se transfieren más de NR_IOREQS */
if (blocks_ahead > NR_IOREQS) blocks_ahead = NR_IOREQS;

/* Leer al menos el mínimo nº de bloques, a no ser que la lectura sea no secuancial. */
if (blocks_ahead < BLOCKS_MINIMUM && rip->i_seek == NO_SEEK)
   blocks_ahead = BLOCKS_MINIMUM;

/* No se puede pasar del final del fichero. */
if (blocks_ahead > blocks_left) blocks_ahead = blocks_left;

read_q_size = 0;

/* Adquirir bloques de caché. */
for (;;) {
   read_q[read_q_size++] = bp;

   if (--blocks_ahead == 0) break;
```

```
      /* No llenar la cache, dejar al menos 4 libres. */
      if (bufs_in_use >= NR_BUFS - 4) break;

      block++;

      bp = get_block(dev, block, PREFETCH);
      if (bp->b_dev != NO_DEV) {
            /* ¡uy!, el bloque ya se encuentra en caché, sácalo. */
            put_block(bp, FULL_DATA_BLOCK);
            break;
      }
  }
  rw_scattered(dev, read_q, read_q_size, READING);
  return(get_block(dev, baseblock, NORMAL));
}
```

CUESTIONES

1. Pasos que se dan ante una llamada **READ**.
2. ¿Cómo se realiza la transferencia en el procedimiento **read_write**?

FIGURA 1.

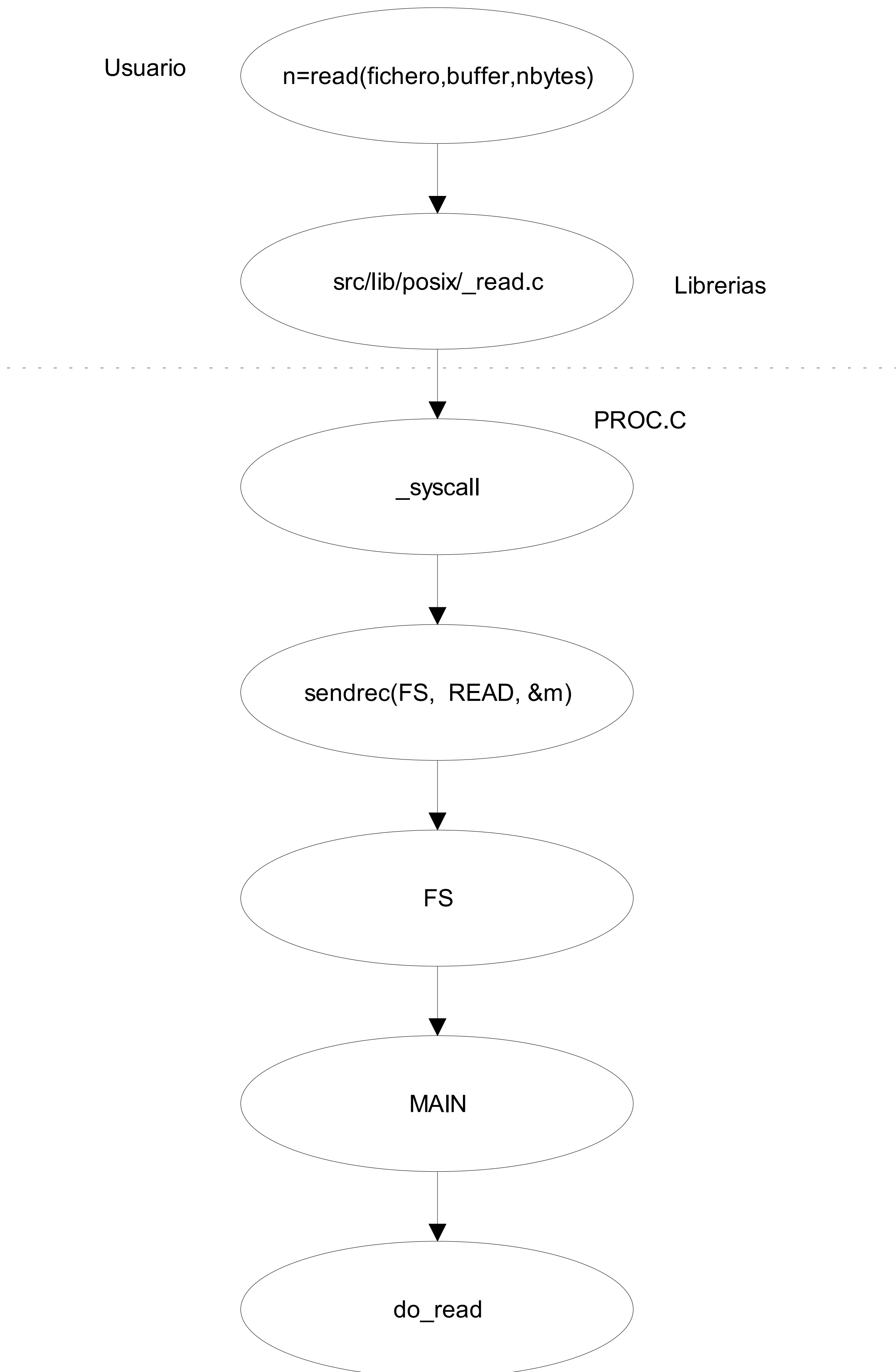
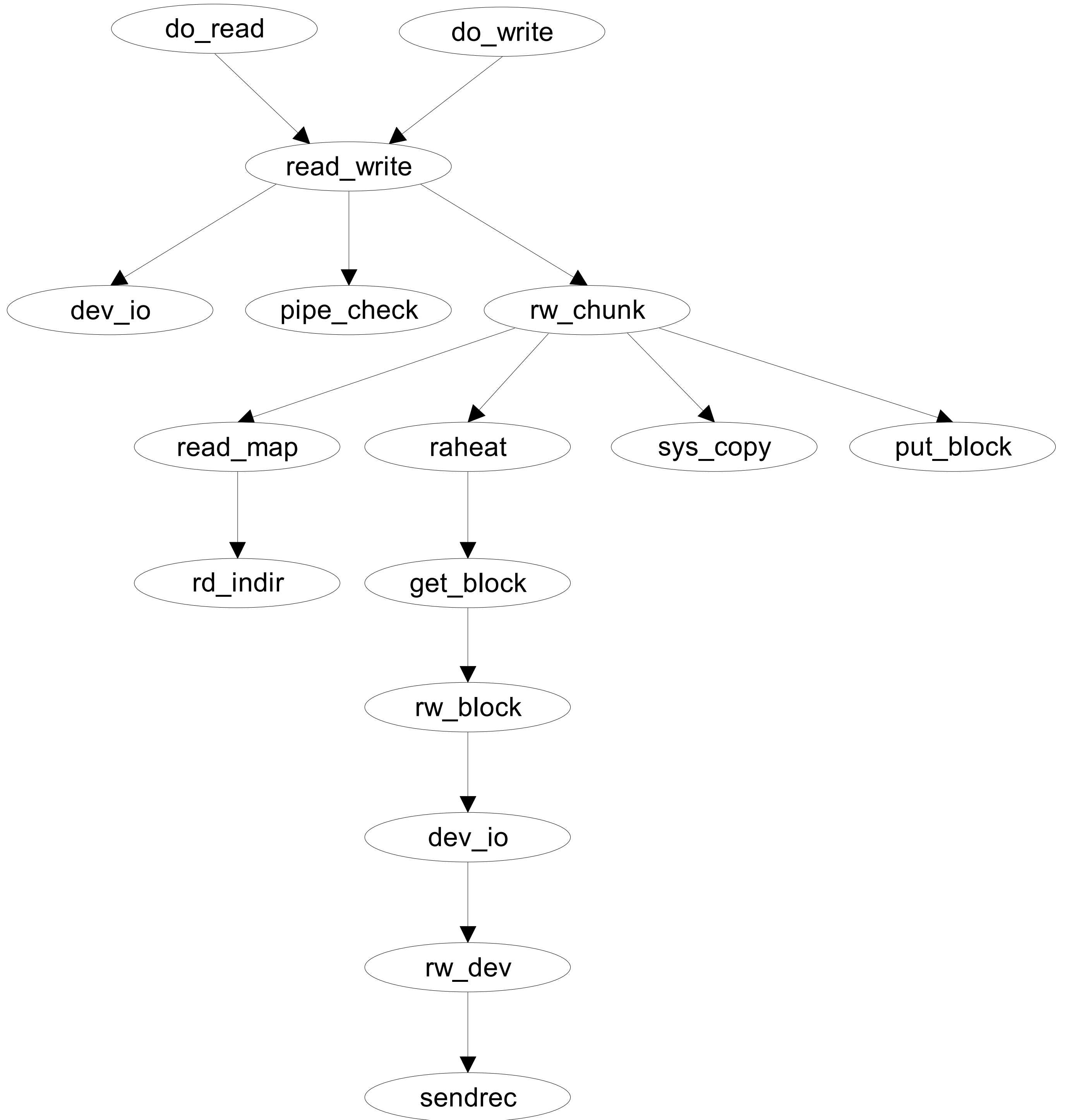


FIGURA 2



LIBRERÍA

++

src/lib/posix/_read.c

++

```
43400#include <lib.h>
43401#define read    _read
43402#include <unistd.h>
43403
43404PUBLIC ssize_t read(fd, buffer, nbytes)
43405int fd;
43406void *buffer;
43407size_t nbytes;
43408{
43409  message m;
43410
43411  m.m1_i1 = fd;
43412  m.m1_i2 = nbytes;
43413  m.m1_p1 = (char *) buffer;
43414  return(_syscall(FS, READ, &m));
43415}
```