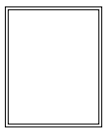
Ampliación de Sistemas Operativos

pipe.c



mayo de 2001

Marco Galluzzi
F^co Javier Cazorla Almeida

Indice

1. Introducción

Los *pipes*, término traducido al español como *tuberías* o *conductos*, son archivos que se usan corrientemente para la comunicación entre dos procesos, donde un proceso escritor va llenando la tubería y el proceso lector la va vaciando. De entre las diferentes formas de comunicación es la más rápida ya que se suele emplear como canal de transmisión la memoria principal.

Las tuberías se parecen a los archivos ordinarios pero difieren de estos principalmente en dos aspectos:

La forma de creación. Mientras los ficheros ordinarios se crean mediante la llamada *creat()*, las tuberías se crean mediante la llamada *pipe()*. La llamada *pipe()* se ejecuta con la función *do_pipe()*. Las tuberías son propiedad del sistema, no del usuario, y se encuentran en el dispositivo de tuberías designado en el fichero *include/minix/config.h*, que bien podría ser, como ya hemos dicho, un disco en RAM, ya que los datos de las tuberías no se tienen que conservar permanentemente.

La lectura y la escritura. La diferencia con la lectura y escritura de un fichero ordinario está en que una tubería tiene capacidad finita. Un intento de escritura en una tubería que ya está llena causa la suspensión del escritor. Asimismo, la lectura de una tubería vacía suspende al lector. Para realizar la suspensión de un lector o un escritor se lleva a cabo un control implícito de dos punteros, la posición y el tamaño actual de la tubería.

2. El fichero pipe.c

Este fichero, a parte que con la creación de una tubería, tiene que ver fundamentalmente con la suspensión y la reanimación (del inglés *revive*) de procesos. Un proceso puede ser suspendido porque quiere leer o escribir en una tubería y no puede, o porque quiere leer o escribir en un fichero especial y no puede. Cuando un proceso no puede continuar es suspendido, y reanimado más tarde cuando tiene la posibilidad de continuar.

Las funciones existentes en este fichero son:

do_pipe: realiza la llamada al sistema *pipe()*.
pipe_check: controla si una lectura o una escritura es posible en este momento.
suspend: suspende un proceso que no puede realizar la lectura o escritura pedida.
release: controla si un proceso suspendido puede ser desbloqueado y lo desbloquea.
revive: marca un proceso suspendido como capacitado para ejecutarse otra vez.
do_unpause: una señal se ha enviado al proceso, mira si está suspendido.

Este fichero se encuentra en la línea de código 24300.

A continuación vemos la cabecera del fichero pipe.c:

```
/*===========================================================================*
 *                                pipe.c                                     *
 *===========================================================================*/
#include "fs.h"
#include <fcntl.h>
#include <signal.h>
#include <minix/boot.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "dev.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"

PRIVATE message mess;
```

3. Estructuras, variables globales, constantes

En este apartado vamos a ver todas aquellas estructuras, variables globales y constantes que se utilizan en muchas de las funciones de este fichero.

3.1. Estructuras

Las estructuras que se manejan en estas funciones son tres y seguramente ya las han visto. El siguiente diagrama, similar a uno sacado del libro *Sistemas Operativos. Diseño e implementación*, nos las recuerda:

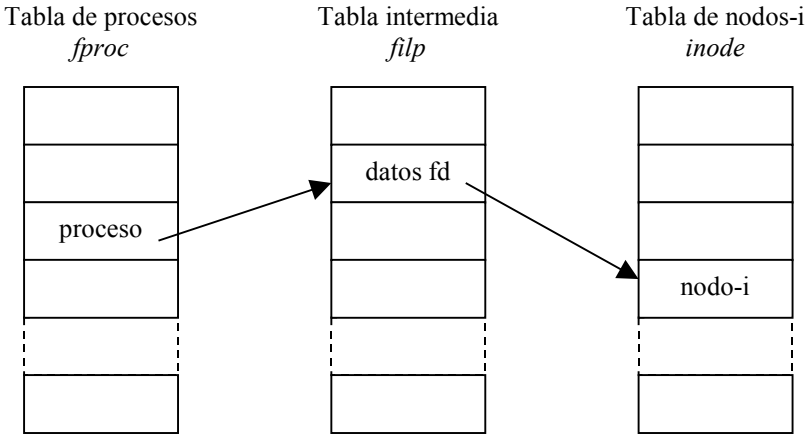


Figura 1: cómo accede un proceso a un nodo-i a través de un descriptor de fichero

A continuación vamos a describir los campos de interés para las funciones de *pipe.c* para cada una de las tres estructuras.

Tabla de procesos del Sistema de Ficheros

Esta contiene la información necesaria para cada proceso. Se reserva una ranura para cada proceso potencial. Así, *NR_PROCS* debe ser igual que en el kernel. No es posible ni necesario saber aquí si una ranura está libre.

```
EXTERN struct fproc {
  ...
  struct filp *fp_filp[OPEN_MAX]; /* La tabla de descriptores de fichero */
  int fp_fd;                  /* guarda el fd si rd/wr no puede acabar */
  char *fp_buffer;            /* guarda el búfer si rd/wr no puede acabar */
  int   fp_nbytes;            /* guarda los bytes si rd/wr no puede acabar */
  char  fp_suspended;         /* activado para indicar proceso suspendido */
  char  fp_revived;           /* activado para indicar proceso reanimándose */
  char  fp_task;              /* en qué tarea está suspendido el proceso */
  ...
} fproc[NR_PROCS];
```

Tabla intermedia

Es un intermediario entre los descriptores de fichero y los nodos-i. Está libre una ranura si *filp_count* es igual a cero.

```
EXTERN struct filp {
  mode_t filp_mode;        /* bits RW, indican cómo se abrió el fichero */
  int filp_flags;          /* flags de las llamadas open y fcntl */
  int filp_count;          /* cuántos des. de fichero comparten esta entrada*/
  struct inode *filp_ino;  /* puntero al nodo-i */
  off_t filp_pos;          /* posición en el fichero */
} filp[NR_FILPS];
```

Tabla de nodos-i

Contiene los nodos-i en uso. La primera parte de la estructura contiene campos que estan en el disco; la segunda contiene campos que no lo están.

```
EXTERN struct inode {
  ...
  mode_t i_mode;           /* tipo de fichero, protección, etc. */

  /* los siguientes campos no se encuentran en disco */
  int i_count;            /* num. veces se usa nodo-i; 0 = entrada libre */
  char i_pipe;            /* es igual a I_PIPE si sirve de tubería */
  char i_update;          /* guarda los bits ATIME, CTIME, y MTIME */
  ...
} inode[NR_INODES];
```

3.2. Variables globales

```
EXTERN struct fproc *fp; /* puntero a la entrada del proceso invocador */
EXTERN int susp_count;   /* número de procesos suspendidos en una tubería */
EXTERN int dont_reply;   /* normalmente 0; poner a 1 para inhibir respuesta */
EXTERN int fs_call;      /* número de llamada al sistema */
EXTERN int who;          /* número del proceso invocador */
EXTERN message m;        /* el propio mensaje de entrada */
EXTERN message m1;       /* el mensaje de salida usado para responder */
EXTERN int reviving;     /* número de procesos "de tubería" a reanimar */
```

3.3. Constantes

```
#define NR_PROCS  32        /* número máximo de procesos en la tabla fproc */
#define OPEN_MAX  20        /* núm. máximo de desc. de fichero por proceso */

#define NOT_SUSPENDED  0    /* proc. no suspendido en tubería o tarea */
#define SUSPENDED      1    /* proc. suspendido en tubería o tarea */
#define NOT_REVIVING   0    /* el proceso no está siendo reanimado */
#define REVIVING       1    /* se está reanimando el proceso en suspensión */

#define FILP_CLOSED    0    /* filp_mode: dispositivo asociado cerrado */
#define NIL_FILP (struct filp *) 0   /* indica ausencia de una entrada filp */

#define MM_PROC_NR     0    /* número de proceso del manejador de memoria */

#define pro         m.m1_i1    /* campo del mensaje de entrada */
#define fd          m.m1_i1    /* campo del mensaje de entrada */
#define reply_i1    m1.m1_i1   /* campo del mensaje de salida */
#define reply_i2    m1.m1_i2   /* campo del mensaje de salida */
```

4. Función do_pipe()

Esta función es la que se ejecuta cuando se realiza la llamada al sistema *pipe()*. Lo único que hace realmente es asignar un nodo-i para la tubería y asignar dos descriptores de fichero al proceso invocador. Uno será de sólo lectura y el otro de sólo escritura.

4.1. Funciones utilizadas

```
PUBLIC int get_fd(start, bits, k, fpt)
int start;            /* inicio de la búsqueda */
mode_t bits;          /* modo del fichero a crear (bits RWX) */
int *k;               /* var. donde se devuelve el descriptor de fichero */
struct filp **fpt;    /* var. donde se devuelve una entrada filp */
```

Busca un descriptor de fichero, entre los que puede usar un proceso, que esté libre y también una entrada libre en la tabla filp.

```
PUBLIC struct inode *alloc_inode(dev, bits)
dev_t dev;            /* dispositivo en el cual se va a ubicar el nodo-i */
mode_t bits;          /* modo del nodo-i */
```

Ubica un nodo-i libre en el dispositivo ‘dev’, y devuelve un puntero a él.

```
PUBLIC int read_only(ip)
struct inode *ip;      /* puntero al nodo-i cuyo FS se va a controlar */
```

Controla si el sistema de ficheros, sobre el cual el nodo-i pasado por parámetros está montado, es de sólo lectura.

```
PUBLIC void panic(s,n)
_CONST char *s;
int n;
```

El sistema termina la ejecución mostrando un mensaje por pantalla. Se llama esta función cuando el sistema se ha encontrado con un error irrecuperable.

```
PUBLIC void rw_inode(rip, rw_flag)
register struct inode *rip;    /* puntero al nodo-i a escribir o leer */
int rw_flag;                   /* se lee o se escribe el nodo-i */
```

Una entrada en la tabla de nodos-i se va a copiar a disco o se va copiar de disco.

```
PUBLIC void dup_inode(ip)
struct inode *ip;        /* puntero al nodo-i a duplicar */
```

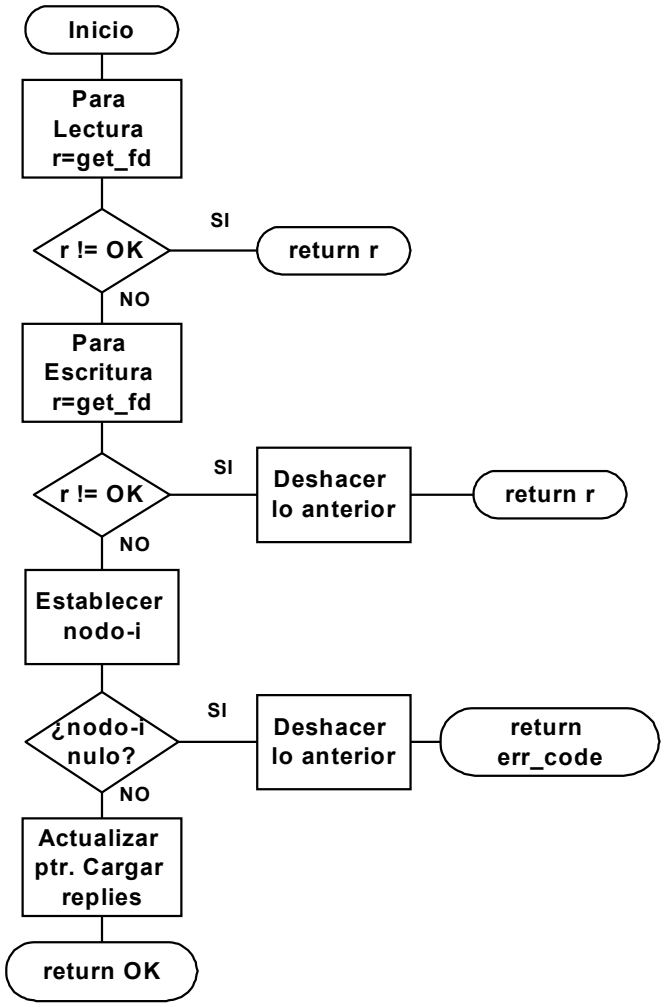
Crea un duplicado del nodo-i de la variable que se le pasa como parámetro. Lo único que hace realmente es incrementar en uno el número de veces que está siendo utilizado el nodo-i.

4.2. Código fuente

```
/*===========================================================================*
 *                                do_pipe                                    *
 *===========================================================================*/
PUBLIC int do_pipe()
{
  register struct fproc *rfp;         /* puntero a la tabla de procesos */
  register struct inode *rip;         /* puntero a la tabla de nodos-i */
  int r;                              /* almacena el valor a devolver */
  struct filp *fil_ptr0, *fil_ptr1;   /* punteros a la tabla filp */
  int fil_des[2];                     /* almacena dos descrip. de fichero */

  /* obtener el puntero al proceso invocador */
  rfp = fp;

  /* obtener un desc. de fichero para lectura */
  if ( (r = get_fd(0, R_BIT, &fil_des[0], &fil_ptr0)) != OK) return(r);

  /* rellenar la entrada de la tabla de desc. de fichero del proceso */
  rfp->fp_filp[fil_des[0]] = fil_ptr0;
  fil_ptr0->filp_count = 1;

  /* obtener un desc. de fichero para escritura */
  if ( (r = get_fd(0, W_BIT, &fil_des[1], &fil_ptr1)) != OK)
  {
   /* borrar la entrada conseguida antes */
   rfp->fp_filp[fil_des[0]] = NIL_FILP;
   fil_ptr0->filp_count = 0;
   return(r);
  }

  /* rellenar la entrada de la tabla de desc. de fichero del proceso */
  rfp->fp_filp[fil_des[1]] = fil_ptr1;
  fil_ptr1->filp_count = 1;

  /* obtener un nodo-i del dispositivo de tuberías */
  if ( (rip = alloc_inode(PIPE_DEV, I_REGULAR) ) == NIL_INODE)
  {
   /* borrar las dos entradas conseguidas antes */
   rfp->fp_filp[fil_des[0]] = NIL_FILP;
   fil_ptr0->filp_count = 0;
   rfp->fp_filp[fil_des[1]] = NIL_FILP;
   fil_ptr1->filp_count = 0;
   return(err_code);
  }
```

```
  /* si el disp. asociado al pipe es de sólo lectura, parar la ejecución */
  if (read_only(rip) != OK) panic("pipe device is read only", NO_NUM);

  /* actualizar campos del nodo-i */
  rip->i_pipe = I_PIPE;
  rip->i_mode &= ~I_REGULAR;
  rip->i_mode |= I_NAMED_PIPE;   /* pipes y FIFOs tienen activado este bit */

  /* actualizar campos de la entradas filp */
  fil_ptr0->filp_ino = rip;
  fil_ptr0->filp_flags = O_RDONLY;
  dup_inode(rip);                        /* el nodo-i va a tener dos usos */
  fil_ptr1->filp_ino = rip;
  fil_ptr1->filp_flags = O_WRONLY;
  rw_inode(rip, WRITING);                /* marcar el nodo-i como asignado */

  /* son sinónimos de las variables del mensaje de salida, asignar fd's */
  reply_i1 = fil_des[0];
  reply_i2 = fil_des[1];
  rip->i_update = ATIME | CTIME | MTIME;

  return(OK);
}
```

4.3. Diagrama de flujo



5. Función pipe_check()

Efectúa las distintas pruebas para comprobar si se puede realizar una operación con una tubería. Una escritura sobre una tubería llena suspenderá al escritor, y una lectura sobre una tubería vacía bloqueará al lector si todavía existe algún escritor, sino, devolverá cero. Además de las pruebas mencionadas *pipe_check()* invoca *release()* para ver si un proceso que antes se había suspendido por falta o exceso de datos ya puede reanimarse. También se detecta aquí la escritura de una tubería rota, condición que se da cuando no existen lectores para la misma.

5.1. Funciones utilizadas

```
PUBLIC struct filp *find_filp(rip, bits)
register struct inode *rip; /* nodo-i al que apunta la entrada filp a buscar*/
Mode_t bits;                /* modo de la entrada filp a buscar (bits RWX) */
```

Busca una entrada en la tabla filp que apunte al nodo-i pasado por parámetros y que además esté en el modo especificado. Esta función se utiliza para determinar si todavía alguien está interesado en un nodo-i.

```
PRIVATE int do_kill(m_ptr)
register message *m_ptr;  /* puntero al mensaje solicitado */
```

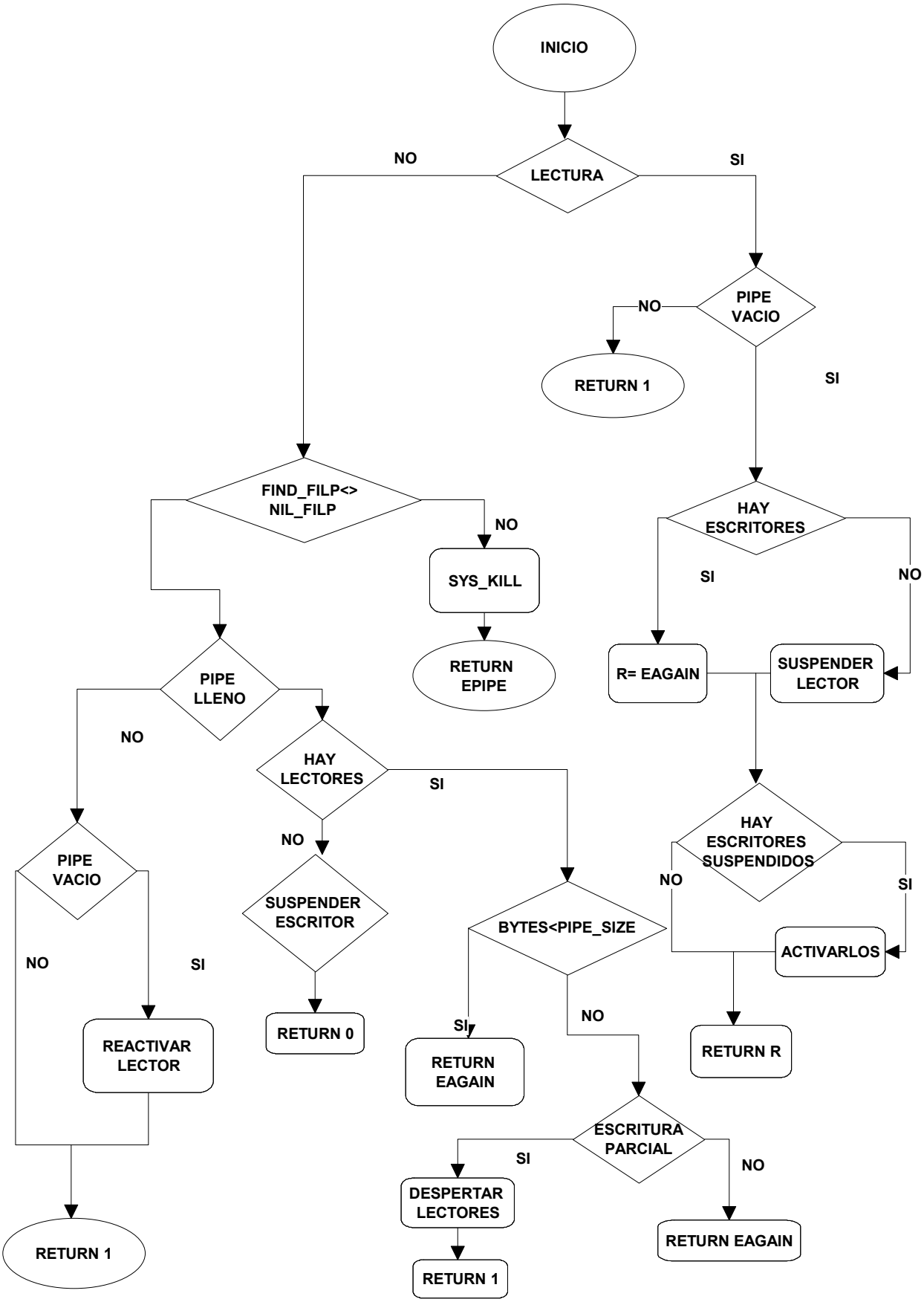
Lleva a cabo la función *sys_kill()*. Provoca el envío de una señal a un proceso por medio del MM. Nótese como no tiene nada que ver con la llamada al sistema *kill()*.

5.2. Código fuente

```
/*===========================================================================*
 *                                pipe_check                                 *
 *===========================================================================*/
PUBLIC int pipe_check(rip, rw_flag, oflags, bytes, position, canwrite)
register struct inode *rip;  /* puntero a la tabla de nodos-i */
int rw_flag;                 /* escribimos o leemos */
int oflags;                  /* flags activados por open o fcntl */
register int bytes;          /* bytes a leer o escribir */
register off_t position;     /* la posición actual del fichero */
int *canwrite;               /* devuelve num. de bytes que podemos escribir */
{
  int r = 0;

  /* ¿se quiere leer? */
  if (rw_flag == READING)
  {

   /* ¿la tuberia está vacia? */
   if (position >= rip->i_size)
   {
      /* ¿existen escritores? */
      if (find_filp(rip, W_BIT) != NIL_FILP)
      {
         /* ¿el tipo de llamada es bloqueante? */
         if (oflags & O_NONBLOCK)
            r = EAGAIN;
         else
            suspend(XPIPE);   /* suspender al lector */


         /* si existen escritores suspendidos -> desbloquearlos */
```

```
            if (susp_count > 0) release(rip, WRITE, susp_count);
        }
        return(r);
     }
    }
    else /* se quiere escribir en una tubería */
    {
     /*   if (bytes > PIPE_SIZE) return(EFBIG); */

     /* controlar si hay lectores */
     if (find_filp(rip, R_BIT) == NIL_FILP)
     {
        /* comunicar al kernel que debe generar una señal SIGPIPE */
        sys_kill((int)(fp - fproc), SIGPIPE);
        return(EPIPE);                      /* retornar tubería rota */
     }

     /* ¿se va a llenar la tubería? */
     if (position + bytes > PIPE_SIZE)
     {
        /* ¿se va a escribir menos que PIPE_SIZE y llamada no bloqueante? */
        if ((oflags & O_NONBLOCK) && bytes < PIPE_SIZE)
           return(EAGAIN);
        else
           /* ¿se va a escribir más que PIPE_SIZE y llamada no bloqueante? */
           if ((oflags & O_NONBLOCK) && bytes > PIPE_SIZE)
           {
              /* ¿puede escribirse algo en la tubería? */
              if ( (*canwrite = (PIPE_SIZE - position)) > 0)
              {
                 /* realizar escritura parcial y despertar lectores */
                 release(rip, READ, susp_count);
                 return(1);
              }
              else
              {
                 return(EAGAIN);
              }
           }

        /* ¿se va a escribir más que PIPE_SIZE? (llamada bloqueante) */
        if (bytes > PIPE_SIZE)
        {
           /* ¿puede escribirse algo en la tubería? */
           if ((*canwrite = PIPE_SIZE - position) > 0)
           {
             /* realizar escritura parcial y despertar lectores */
              release(rip, READ, susp_count);
              return(1);
           }
        }

        suspend(XPIPE);   /* parar escritor -> tubería llena */
        return(0);
     }

     /* si la tubería está vacía, despertar lectores suspendidos */
     if (position == 0) release(rip, READ, susp_count);
    }

    *canwrite = 0;
    return(1);
}
```

5.3. Diagrama de flujo



6. Función suspend()

El acto de suspender un proceso lo lleva a cabo esta función. Lo único que hace es guardar los parámetros de la llamada en la tabla de procesos y poner el flag *dont_reply* a TRUE para inhibir el mensaje de respuesta del sistema de archivos hacia el proceso invocador.

6.1. Código fuente

```
/*===========================================================================*
 *                                suspend                                    *
 *===========================================================================*/
PUBLIC void suspend(task)
int task;               /* ¿por quién espera el proceso? (XPIPE = tubería) */
{
  /* incrementar el número de procesos suspendidos */
  if (task == XPIPE || task == XPOPEN) susp_count++;

  /* actualizar los campos del proceso invocador */
  fp->fp_suspended = SUSPENDED;
  fp->fp_fd = fd << 8 | fs_call;
  fp->fp_task = -task;

  /* ¿la tarea es una llamada a fcntl()? */
  if (task == XLOCK)
  {
     fp->fp_buffer = (char *) name1; /* 3er argumento a fcntl() */
     fp->fp_nbytes = request;        /* 2ndo argumento a fcntl() */
  }
  else
  {
     fp->fp_buffer = buffer;      /* para lecturas y escrituras */
     fp->fp_nbytes = nbytes;
  }

  dont_reply = TRUE;    /* no enviar ahora al proc. invocador una respuesta */
}
```

7. Función release()

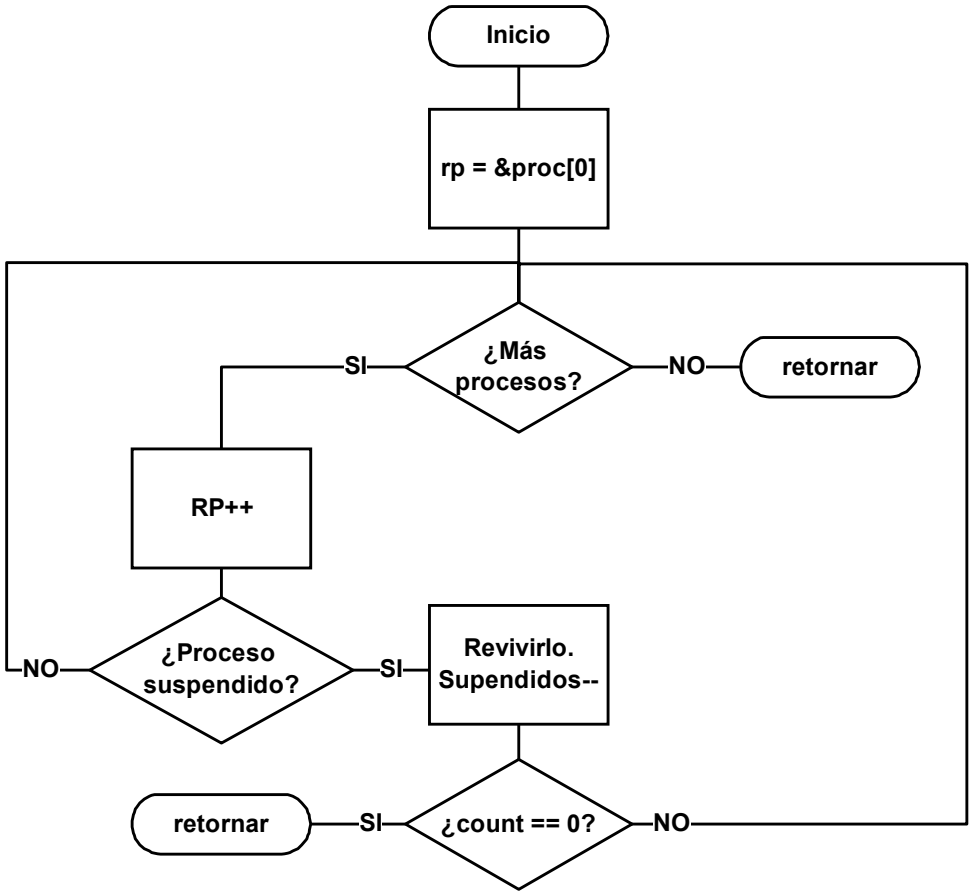
Este procedimiento se invoca para ver si un proceso que estaba suspendido esperando una tubería ya puede continuar. Si *release()* encuentra uno, invoca *revive()* para activar un flag y que el ciclo principal se fije en él. Esta función no es una llamada al sistema, sin embargo se suele asociar con ellas porque usa el mecanismo de transferencia de mensajes.

7.1. Código fuente

```
/*===========================================================================*
 *                                release                                    *
 *===========================================================================*/
PUBLIC void release(ip, call_nr, count)
register struct inode *ip;   /* el nodo-i de la tubería */
int call_nr;                 /* modo: READ, WRITE, OPEN o CREAT */
int count;                   /* núm. máximo de procesos a desbloquear */
{
  register struct fproc *rp;  /* puntero a la tabla de procesos */

  /* buscar en la tabla de procesos */
  for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++)
  {
     if (rp->fp_suspended == SUSPENDED &&  /* si está suspendido */
```

```
            rp->fp_revived == NOT_REVIVING && /* pendiente de reanimación */
            (rp->fp_fd & BYTE) == call_nr &&  /* tiene el modo deseado */
            rp->fp_filp[rp->fp_fd>>8]->filp_ino == ip) /* apunta a la tubería */
        {
           revive((int)(rp - fproc), 0);

           susp_count--;       /* llevar la cuenta de los procesos suspendidos */
           if (--count == 0) return;
        }
     }
}
```

7.3. Diagrama de flujo



8. Función revive()

Reactiva un proceso bloqueado con anterioridad. Si el proceso está bloqueado en una terminal, ésta es la forma última en que se libera.

8.1. Funciones utilizadas

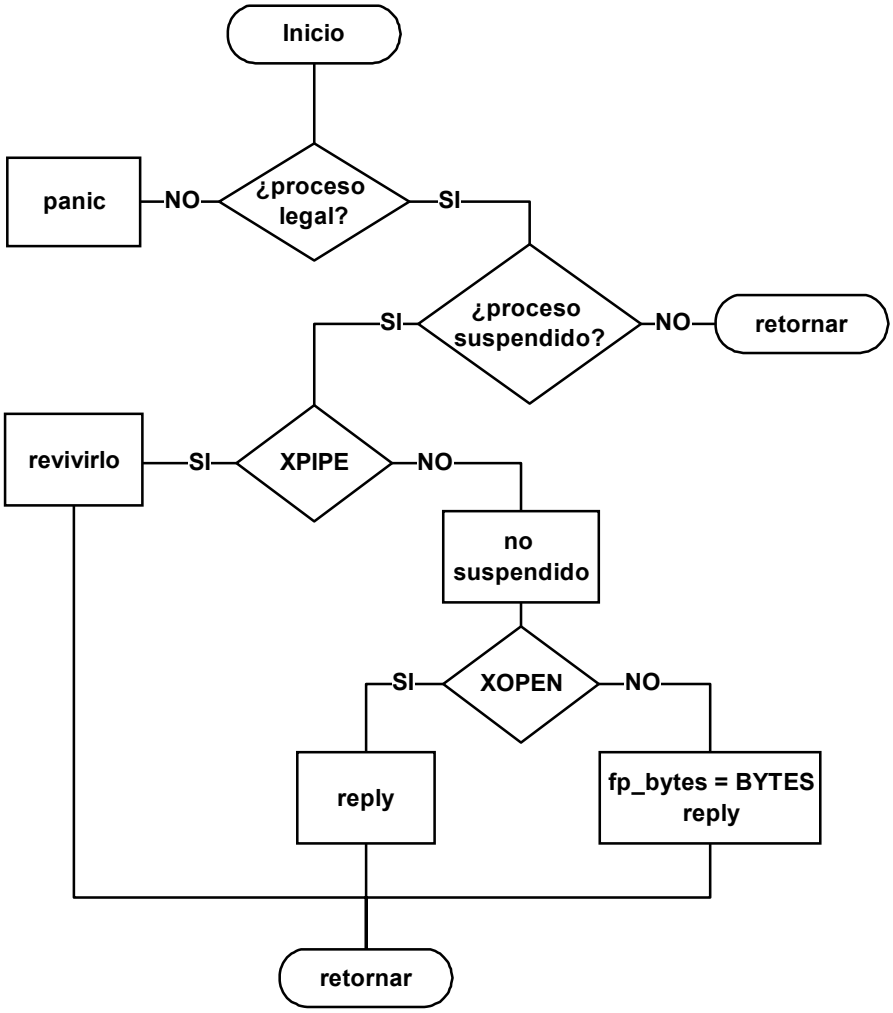
```
PUBLIC void reply(whom, result)
int whom;     /* proceso al que se le envia la respuesta */
int result;   /* resultado de la llamada (normalmente OK o num. de error #) */
```

Envia una respuesta a un proceso de usuario.

8.2. Código fuente

```
/*===========================================================================*
 *                                revive                                     *
 *===========================================================================*/
PUBLIC void revive(proc_nr, bytes)
int proc_nr;          /* número del proceso a reanimar */
int bytes;            /* si está suspendido en una tarea, bytes ya leídos */
{
  register struct fproc *rfp;     /* puntero al proceso a reanimar */
  register int task;              /* tarea */

  /* controlar que se pasa un núm. de proceso válido */
  if (proc_nr < 0 || proc_nr >= NR_PROCS) panic("revive err", proc_nr);

  /* obtener el puntero al proceso a reanimar */
  rfp = &fproc[proc_nr];

  /* controlar que el proceso está suspendido y pendiente de reanimación */
  if (rfp->fp_suspended == NOT_SUSPENDED || rfp->fp_revived == REVIVING) return;

  task = -rfp->fp_task;

  /* el flag de reanimación sólo se aplica a tuberías; los procesos que esperan
     al TTY obtienen un mensaje de inmediato */

  if (task == XPIPE || task == XLOCK)
  {
     /* reanimar un proceso suspendido en una tubería o cerrojo */
     rfp->fp_revived = REVIVING;
     reviving++;
  }
  else
  {
     rfp->fp_suspended = NOT_SUSPENDED;

     if (task == XPOPEN) /* proceso bloqueado en un open() o un create() */
        reply(proc_nr, rfp->fp_fd>>8);
     else
     {
        /* renimar un proceso suspendido en una TTY u otro dispositivo */
        rfp->fp_nbytes = bytes;
        reply(proc_nr, bytes);   /* desbloquear el proceso */
     }
  }
}
```

8.3. Diagrama de flujo



9. Función do_unpause()

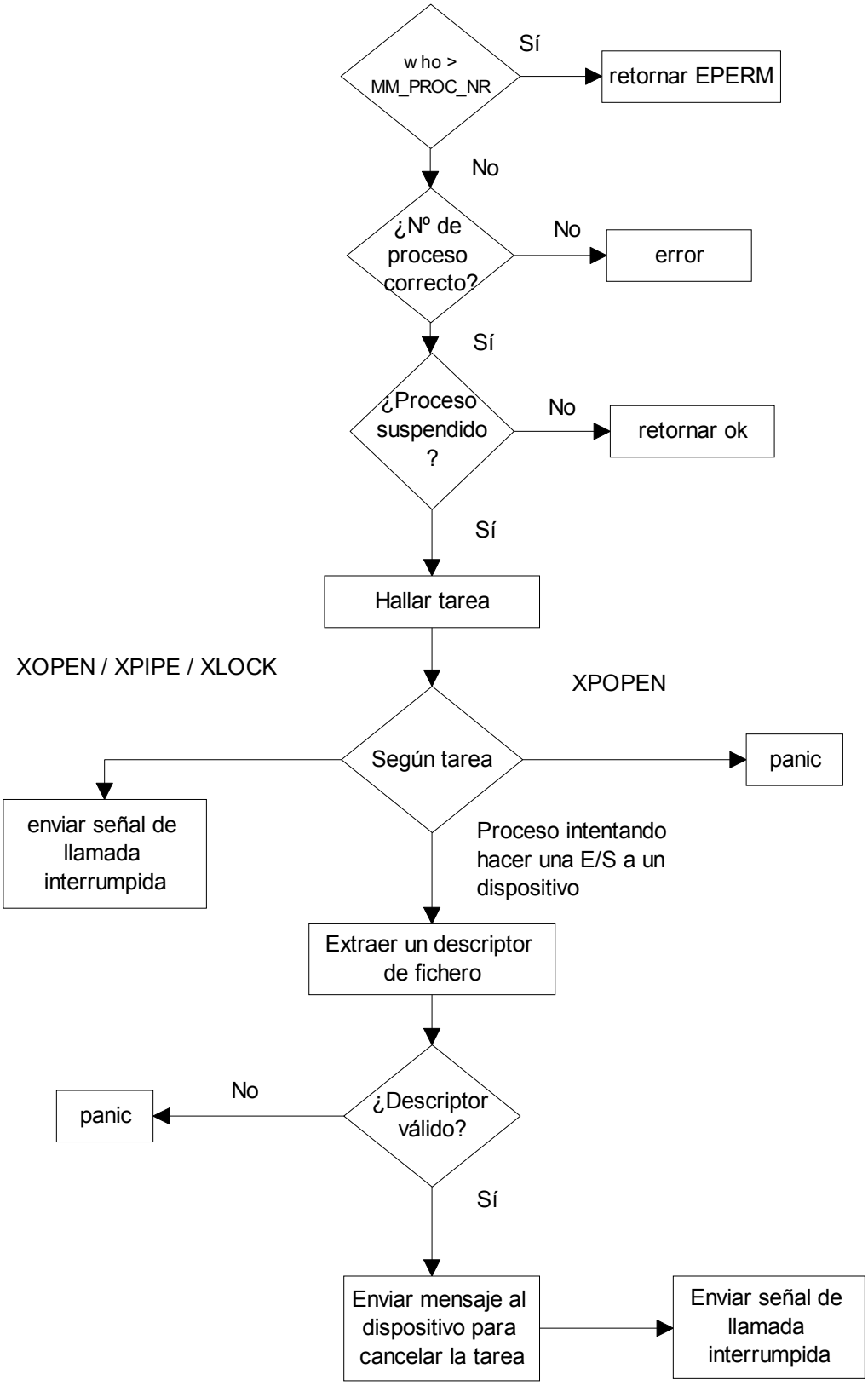
Cuando el manejador de memoria está tratando de enviar una señal a un proceso, debe averiguar si ese proceso está suspendido esperando una tubería o un fichero especial (en cuyo caso deberá despertársele con un error EINTR). Puesto que el manejador de memoria nada sabe acerca de tuberías ni de ficheros especiales, envía un mensaje al sistema de archivos para preguntarle. Ese mensaje es procesado por esta función. Al igual que *revive()*, *do_unpause()* tiene cierto parecido con las llamadas al sistema, aunque no es una de ellas.

9.1. Código fuente

```
/*===========================================================================*
 *                                do_unpause                                 *
 *===========================================================================*/
PUBLIC int do_unpause()
{
  register struct fproc *rfp; /* puntero a la tabla de procesos */
  int proc_nr, task, fild;    /* núm. de proceso, tarea y descr. de fichero */
  struct filp *f;             /* puntero a la tabla filp */
  dev_t dev;                  /* dispositivo */
```

```
  /* el proceso invocador debe ser el manejador de memoria */
  if (who > MM_PROC_NR) return(EPERM);

  /* copiar el identificador de proceso */
  proc_nr = pro;

  /* controlar que se pasa un núm. de proceso válido */
  if (proc_nr < 0 || proc_nr >= NR_PROCS) panic("unpause err 1", proc_nr);

  /* obtener el puntero al proceso */
  rfp = &fproc[proc_nr];

  if (rfp->fp_suspended == NOT_SUSPENDED) return(OK);
  task = -rfp->fp_task;

  switch(task)
  {
     case XPIPE:      /* el proceso intenta leer o escribir en una tubería */
        break;

     case XOPEN:      /* el proceso intenta abrir un fichero especial */
        panic ("fs/do_unpause called with XOPEN\n", NO_NUM);

     case XLOCK:      /* el proceso intenta activar un cerrojo con FCNTL */
        break;

     case XPOPEN:     /* el proceso intenta abrir un fifo */
        break;

     default:/* el proceso intenta realizar E/S a un dispositivo (e.g. tty) */

        /* obtener descriptor de fichero */
        fild = (rfp->fp_fd >> 8) & BYTE;

        /* controlar que es un descriptor de fichero válido */
        if (fild < 0 || fild >= OPEN_MAX) panic("unpause err 2",NO_NUM);
        f = rfp->fp_filp[fild];

        /* obtener dispositivo en el que se suspendió el proceso */
        dev = (dev_t) f->filp_ino->i_zone[0];
        mess.TTY_LINE = (dev >> MINOR) & BYTE;
        mess.PROC_NR = proc_nr;

        /* decir al kernel R o W; el modo se obtiene de la llamada actual,
           no abierto */
        mess.COUNT = (rfp->fp_fd & BYTE) == READ ? R_BIT : W_BIT;
        mess.m_type = CANCEL;
        fp = rfp;             /* hack - call_ctty usa fp */
        (*dmap[(dev >> MAJOR) & BYTE].dmap_rw)(task, &mess);
  }

  rfp->fp_suspended = NOT_SUSPENDED;
  reply(proc_nr, EINTR);             /* señal de llamada interrumpida */
  return(OK);
}
```

9.2. Diagrama de flujo



10. Preguntas y respuestas

¿Cuándo se dice que una tubería está rota?

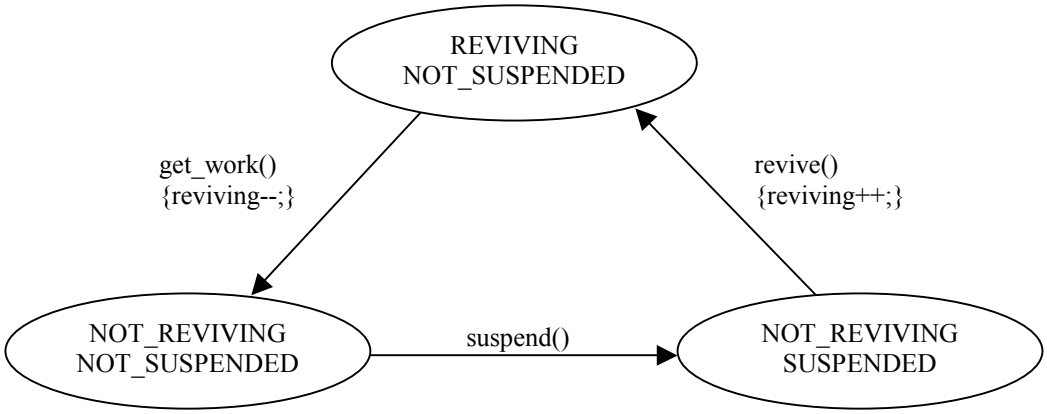
Cuando un proceso quiere escribir en una tubería pero no hay lectores para la misma.

En una tubería, ¿puede haber lectores pero no escritores?

Sí. Pueden aparecer lectores para una tubería sin escritores. En tal caso la función *pipe_check()* simplemente devuelve un cero y no una macro de error.

Explique brevemente cómo se actualizan los flags que indican si un proceso está activo o no.

Los campos de la estructura *fproc* (asignada a un proceso) que guardan los flags del estado de un proceso son *fp_suspended* y *fp_revived*. El primer campo nos indica si el proceso está suspendido o no, mientras el segundo indica si el bucle principal del FS, en concreto la función *get_work()*, debe volver a reanimar este proceso o no. Además la variable global *reviving* indica el número de procesos que hay que reanimar. A continuación se explica gráficamente la modificación de dichos valores.



¿Qué condiciones se tienen que dar para suspender a un lector?

Un lector se suspende en la función *pipe_check()*. Esto sólo sucede en un caso y las condiciones que se deben dar son:

- La tubería está vacía, existen escritores y el tipo de llamada es bloqueante.

¿Qué condiciones se tienen que dar para suspender a un escritor?

Un escritor se suspende en la función *pipe_check()*. Esto sólo sucede en un caso y las condiciones que se deben dar son:

- La tubería está llena y la llamada es bloqueante.

¿Cómo se lleva a cabo la suspensión de un proceso?

La suspensión se lleva a cabo mediante una llamada a la función *suspend()*. Lo que hace es guardar los parámetros de la llamada en la tabla de procesos y poner el flag *dont_reply* a TRUE para inhibir el mensaje de respuesta del sistema de archivos hacia el proceso invocador.

¿Qué condiciones se tienen que dar para reanimar a un lector?

Un lector se reanima en la función *pipe_check()*. Esto puede pasar en dos casos:

- Un proceso quiere escribir en una tubería, la tubería se va llenar, se quiere escribir más que PIPE_SIZE y la tubería no está del todo llena.
- Un proceso quiere escribir en una tubería, la tubería no se va llenar y ésta está del todo vacía.

¿Qué condiciones se tienen que dar para reanimar a un escritor?

Un escritor se reanima en la función *pipe_check()*. Esto sólo puede pasar en un caso:

- Un proceso quiere leer en una tubería y la tubería no está vacía.

¿Cómo se lleva a cabo la reanimación de un proceso?

La reanimación se lleva a cabo mediante una llamada a la función *release()*. Si esta función encuentra un proceso suspendido, invoca *revive()* para activar un flag y que el ciclo principal se fije en él.