FS/ MISC

ÍNDICE

1.- INTRODUCCIÓN **4**

2.- DIAGRAMAS DE LLAMADAS AL SISTEMA. **5**

3.- ESTRUCTURAS PRINCIPALES QUE SE UTILIZAN. **7**

4.- FUENTES **8**

4.1.- FUNCIÓN DO_DUP(). 8

4.2.- FUNCIÓN DO_FCNTL(). 10

4.3.- FUNCIÓN DO_SYNC(). 11

4.4.- FUNCIÓN DO_FORK(). 12

4.5.- FUNCIÓN DO_EXEC(). 14

4.6.- FUNCIÓN DO_EXIT(). 15

4.7.- FUNCIÓN DO_SET(). 18

4.8.- FUNCIÓN DO_REVIVE(). 19

1.- INTRODUCCIÓN¡Error! Marcador no definido.

El fichero Misc.c contiene un conjunto de procedimientos, de los cuales algunos realizan llamadas al sistema simples y otros realizan una parte de algunas llamadas al sistema que efectúa el manejador de la memoria (realizan la parte que tiene que ver con el manejo de ficheros).

Estos procedimientos son:

- **do_dup:** Se encarga de realizar la llamada al sistema DUP, que duplica el descriptor de un archivo creando uno nuevo, que apunta al mismo archivo que su argumento. Esta llamada al sistema tiene una variante, DUP v.2, que se encarga de que el descriptor que se pasa como segundo parámetro apunte a la misma entrada 'filp' que el descriptor que se quiere duplicar. Ambas llamadas al sistema son obsoletas, de forma que realmente se invoca a la llamada FCNTL cuando aparecen DUP o DUP2, pero se incluyen en el fichero como apoyo a programas binarios viejos.

- **do_sync:** Realiza la llamada al sistema SYNC, que copia en disco todos los bloques y nodos-i modificados desde la última vez que se cargaron.

- **do_fcntl:** realiza la llamada al sistema FCNTL. Se utiliza para realizar la operación dup y dup v.2. además de leer y cambiar los flags de los ficheros.

Cuando un proceso bifurca, es imprescindible avisar al Kernel y al File System para que se encarguen de ello. Así, las llamadas al sistema FORK, EXIT, EXEC y SET son manejadas primero por el manejador de memoria que luego llama al File System (las llamadas no proceden entonces de los procesos de usuario), para que éste efectúe una parte del trabajo usando para ello las siguientes funciones. En ellas, se registra la información pertinente en la parte de la tabla de procesos que corresponde al File System.

- **do_fork**: ajusta las tablas (tabla filp y fproc) después de que MM haya hecho una llamada al sistema FORK.

- **do_exit**: realiza la parte relacionada con los ficheros, de la llamada al sistema de MM EXIT (un proceso salió, anotarlo en las tablas).

- **do_set**: fija el uid o gid de un proceso.

- **do_exec**: Busca y cierra todos los archivos que se marcaron para cerrarse al ejecutar (manejar archivos con FD_CLOEXEC encendido después de EXEC por MM)

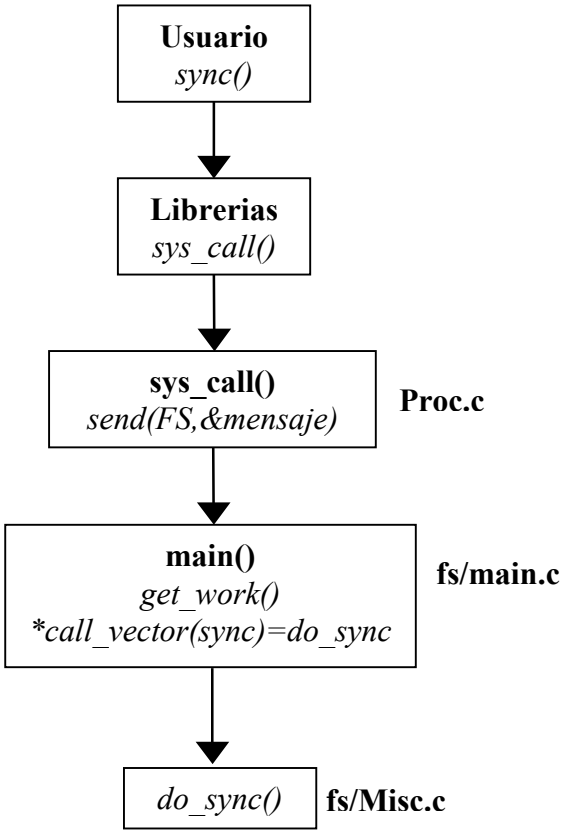
El siguiente procedimiento no es en realidad una llamada al sistema, pero se maneja como tal.

- **do_revive**: revive a un proceso que esperaba por algo.

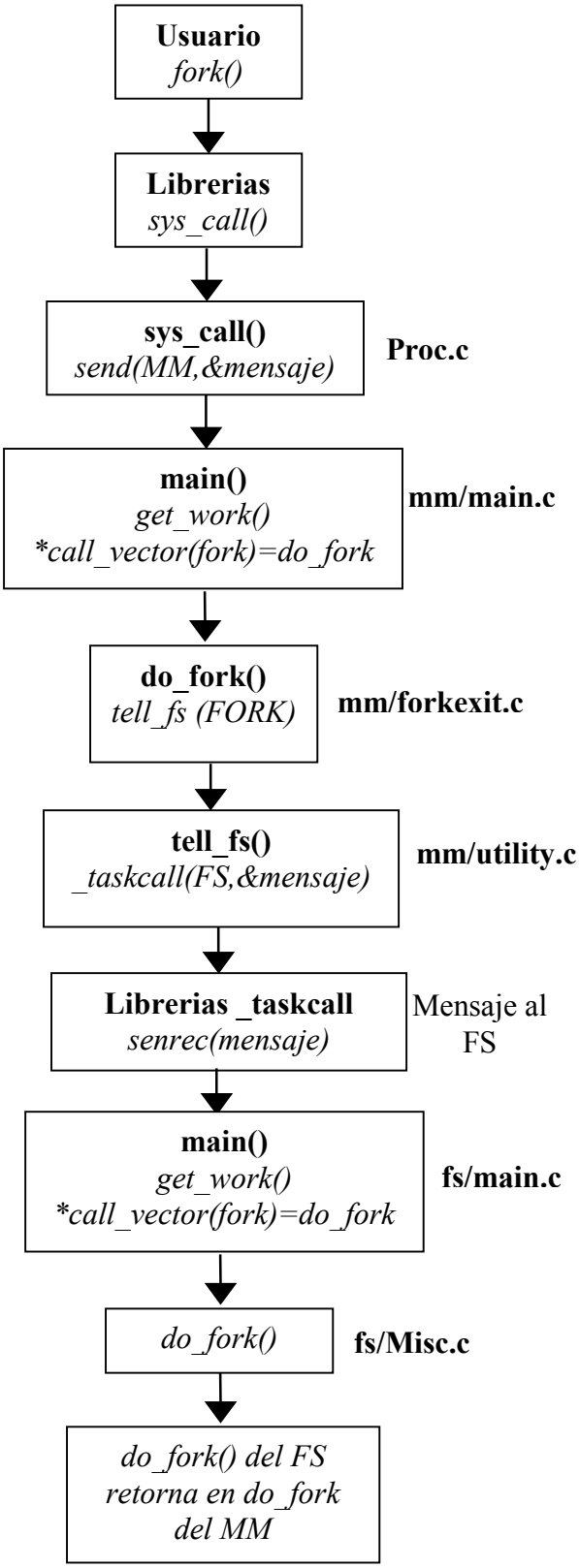
2.- DIAGRAMAS DE LLAMADAS AL SISTEMA.

A continuación se muestran unos diagramas que ayudarán a comprender mejor el proceso que tiene lugar desde que un usuario realiza, por ejemplo, un fork, hasta que se ejecuta el procedimiento correspondiente en la librería Misc.c.

- **Llamadas comunes al FS: sync, dup, fcntl**



Llamadas al FS del manejador de memoria: fork, exit, exec, set



3.- ESTRUCTURAS PRINCIPALES QUE SE UTILIZAN.

Todas estas estructuras son definidas en exposiciones anteriores.

- **FILP (src/fs/file.h)**.

La tabla FILP es la tabla intermedia entre los descriptores de archivo y los nodos i, para permitir la compartición de ficheros. Cada ranura en esta tabla tiene la siguiente estructura:

filp_mode: indica el modo en que se abre el fichero.
filp_count: indica el número de descriptores que comparten esa ranura.
filp_ino: apunta al nodo i del fichero en la estructura inode.
filp_pos: posición actual en el fichero.

- **FPROC (src/fs/fproc.h).**

Es la tabla de procesos con la información de cada proceso importante para el sistema de ficheros. De esta estructura usamos el campo **fp_filp[NR_FDS]** , que es el array donde se encuentran los 20 apuntadores posibles a las ranuras **FILP** para cada proceso.

- **INODE (src/fs/inode.h)**.

En esta tabla están los nodos i que se asocian a cada fichero abierto.

- **BUF (src/fs/buf.h).**

Reserva del buffer; para adquirir un bloque una rutina llama a get_block(), indicándole qué bloque desea. Después el bloque se considera como en uso y se incrementa su campo b_count. Todos los bloques están en una lista LRU. Cuando un bloque se modifica se actualiza su campo 'b_dirt' a **DIRTY** para indicar que se debe reescribir en disco.

- **MENSAJE TIPO 1 (src/fs/param.h).**

Mensaje compuesto por 5 enteros y tres apuntadores a caracteres, donde los dos primeros enteros contienen el número del proceso fuente y el tipo de mensaje respectivamente. Los siguientes campos tendrán diversos significados según el mensaje.

4.- FUENTES

4.1.- FUNCIÓN do_dup().

Duplica el descriptor de un archivo.

- ***Funciones utilizadas:***

➢ **get_filp(fd):** Para el descriptor de archivo fd, comprueba si es un descriptor válido devuelve un puntero a su entrada en la tabla filp (/src/fs/filedes.c).

➢ **get_fd (start,bits,k,fdt):** Se encarga de encontrar un descriptor de archivo libre (k) y un puntero a una entrada de la tabla filp (fpt). Comienza a buscar un descriptor a partir de 'start'. En caso de encontrar un descriptor libre, se copia el modo que tendrá el archivo indicado por 'bits', y se inicializa la posición del fichero. El procedimiento devuelve un entero indicando si encontró algún descriptor libre (/src/fs/filedes.c).

➢ **do_close ():** Cierra el fichero con descriptor de archivo indicado por el mensaje de entrada (campo fd). (/src/fs/fopen.c).

- ***Variables globales*:**

***fp:** Apuntador a la estructura fproc del proceso solicitante.
fd: Tercer entero del mensaje de entrada tipo 1 que contiene un descriptor de archivo.
fd2: Cuarto entero del mensaje de entrada tipo 1 que contiene un descriptor de archivo.

- ***Variables locales*:**

rfd: entero que contendrá el número del descriptor de archivo a duplicar.
***f:** Apuntador a estructura filp que guarda la entrada a esta tabla del descriptor fd.
***dummy:** apuntador a estructura filp usado como dummy para la llamada a get_fd().
***r:** Almacenará el resultado de llamada a la función get_fd().

- ***Constantes*:**

DUP_MASK: Máscara para distinguir entre DUP y DUP2.
NIL_FILP: Ausencia de ranura filp.
NR_FDS: Número de descriptores de archivo máximo por proceso.

- ***Código***:

```
PUBLIC int do_dup()
{ /*Lleva a cabo la llamada al sistema dup(fd) o dup2(fd,fd2). Esta llamada a quedado obsoleta.
 de hecho, no es posible invocarla usando la librería de la v.2, debido a que la librería actual invoca a la
   rutina fcntl() para realizar esta operación. Esta rutina se mantiene para permitir que viejos
   programas binarios puedan ejecutarse.*/

  register int rfd;
  register struct filp *f;
  struct filp *dummy;
  int r;

/* Guarda el valor de fd  en la variable local rfd. Si la  llamada solicitada es DUP2, en rfd se almacena el
valor  del descriptor de archivo sin el bit de llamada DUP2 */

  rfd = fd & ~DUP_MASK;                 /* kill off dup2 bit, if on */

 /*comprueba si el descriptor es válido*/
  if ((f = get_filp(rfd)) == NIL_FILP) return(err_code);

  /* Si la llamada corresponde a DUP (fd=rfd), debemos obtener un descriptor de archivo libre mediante la
función get_fd()  almacenando este valor en fd2. Si la llamada corresponde a DUP2 (fd#rfd),
comprobamos que fd2 es un descriptor de archivo válido. Si el descriptor indicado fd2 es igual al original
no hace  nada.  */

  if (fd == rfd) {                      /* bit not on */
          /* dup(fd) */
          if ( (r = get_fd(0, 0, &fd2, &dummy)) != OK) return(r);
  } else {
          /* dup2(fd, fd2) */
          if (fd2 < 0 || fd2 >= OPEN_MAX) return(EBADF);
          if (rfd == fd2) return(fd2);/* ignore the call: dup2(x, x) */

/*cierra el descriptor de archivo fd2*/
          fd = fd2;             /* prepare to close fd2 */
          (void) do_close();            /* cannot fail */
  }

/* Por último actualiza el descriptor de archivo fd2, apuntando a la misma entrada de la tabla filp que
rfd,e  incrementa  el número de procesos que comparten esa entrada en  filp */

  /* Success. Set up new file descriptors. */
  f->filp_count++;
  fp->fp_filp[fd2] = f;
  return(fd2);
}
```

4.2.- FUNCIÓN do_fcntl().

Para realizar operaciones con un archivo abierto. La llamada se invoca con un descriptor de archivo, un código de solicitud y argumentos adicionales que dependen del tipo de solicitud. Varias de estas solicitudes establecen o leen un flag, y otras sirven para manejar el empleo de comandos de archivos.

Dependiendo del valor que tome request (la solicitud) se realizan distintas operaciones:

Operación	Significado
F_DUPFD	Duplica el descriptor de un fichero, sustituye a DUP
F_GETFD	Obtiene el flag close-on-exec: indica si cerrar el fich.en un exec.
F_SETFD	Cambia flag close-on-exec.
F_GETFL	Obtiene los flags de estado. O_NOMBLOCK, O_APPENED.
F_SETFL	Cambia los flags de estado.
F_GETLK	Obtiene el estado de bloqueo de una región de un fichero.
F_SETLK	Bloquea para lectura y escritura una región.
F_SETLKW	bloquea para escritura..

O_NOMBLOCK: no retardo.
O_APPENED: indica que todas las escrituras se hacen desde el final.

- ***Funciones utilizadas*:**

➢ **lock_op (f, request)**: (lock.c) realiza:

F_GETLK
F_SETLK
F_SETLKW

- ***Código***:

```
PUBLIC int do_fcntl()
{  register struct filp *f;
  int new_fd, r, fl;
  long cloexec_mask;                    /* máscara para el flag FD_CLOEXEC */
  long clo_value;
  struct filp *dummy;

/*comprueba si el descriptor es válido */
  if ((f = get_filp(fd)) == NIL_FILP) return(err_code);

/*dependiendo de request*/
```

```
 switch (request) {
   case F_DUPFD:
        /*remplaza la llamada dup()*/
         /* This replaces the old dup() system call. */
         if (addr < 0 || addr >= OPEN_MAX) return(EINVAL);

        /*obtener un descriptor*/
         if ((r = get_fd(addr, 0, &new_fd, &dummy)) != OK) return(r);

        /* Por último actualiza el descriptor de archivo fd2  apuntando a la misma entrada de la tabla filp
        que rfd, e incrementa  el número de procesos que comparten esa entrada en  filp */
         f->filp_count++;
         fp->fp_filp[new_fd] = f;
         return(new_fd);

   case F_GETFD:
        /*obtiene el flags close-on-exec*/
       return( ((fp->fp_cloexec >> fd) & 01) ? FD_CLOEXEC : 0);

   case F_SETFD:

         /*cambia el valor de close-on-exec falg segun indique addr*/
         cloexec_mask = 1L << fd;/* singleton set position ok */
         clo_value = (addr & FD_CLOEXEC ? cloexec_mask : 0L);
         fp->fp_cloexec = (fp->fp_cloexec & ~cloexec_mask) | clo_value;
         return(OK);

   case F_GETFL:
        /* obtiene el estado de los flags (O_NONBLOCK and O_APPEND)
         fl = f->filp_flags & (O_NONBLOCK | O_APPEND | O_ACCMODE);
         return(fl);

   case F_SETFL:
        /*cambia el valor de O_NONBLOCK | O_APPEND dependiendo de addr*/
         fl = O_NONBLOCK | O_APPEND;
         f->filp_flags = (f->filp_flags & ~fl) | (addr & fl);       return(OK);

case F_GETLK:case F_SETLK: case F_SETLKW:
        /*para bloquear y desbloquear zonas de un fichero*/
         r = lock_op(f, request);
         return(r);
  default:
         return(EINVAL);
 }
}
```

4.3.- FUNCIÓN do_sync().

Copia en disco todos los bloques nodos i modificados desde que se cargaron. Los nodos i deben procesarse primero ya que *rw_inode* deja sus resultados en la cache de bloques.

Una vez que se han escrito todos los nodos i sucios en la cache de bloques, todos los bloques sucios se escriben en disco.

- ***Funciones utilizadas*:**

- **rw_inode(rip,rw_flag):** Realiza una transferencia a/o desde disco, según lo especifique la variable rw_flag, del nodo i indicado por el apuntador *rip (src/fs/inode.c).

- **flusshall(bp->dev)**: Desaloja todos los bloques sucios para un dispositivo (/src/fs/cache.c).

- ***Variables locales*:**

***rip**: Puntero al nodo i para recorrer la tabla inode.
***bp:** Puntero a los buffers para recorrer la cadena de buffers a actualizar en disco.

- ***Constantes*:**

DIRTY: Indica que los datos de memoria se han modificado y deben reescribirse en disco.
NO_DEV: Indica que no hay ningún dispositivo asignado.

- ***Código***:

```
PUBLIC int do_sync()
{
/*Copia todos los bloques y nodos  que se han modificado desde que se cargaron desde disco. Lo único que hace este procedimiento es ir buscando en estas dos listas entradas sin actualizar en disco e ir llamando a los correspondientes procedimientos para su  reescritura. Hay que tener en cuenta el orden en el que se realiza este barrido de las listas, ya que los procedimientos de escritura de nodos i y de superbloques dejan resultados intermedios en los bloques, lo que nos obliga a barrer esta lista en último lugar.*/

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

  /*recorre la lista de inodes y escribe los inodes sucios en disco*/
  for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++)
          if (rip->i_count > 0 && rip->i_dirt == DIRTY) rw_inode(rip, WRITING);

/*recorre la lista de bloques y escribe los bloques sucios en disco, una unidad a la vez*/
  /* Write all the dirty blocks to the disk, one drive at a time. */
  for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
          if (bp->b_dev != NO_DEV && bp->b_dirt == DIRTY) flushall(bp->b_dev);

  return(OK);                   /* sync() can't fail */
}
```

4.4.- FUNCIÓN do_fork().

Sólo el manejador de la memoria puede realizar esta llamada directamente. Este procedimiento se encarga de realizar aquellos aspectos de la llamada al sistema fork() que está relacionada con los archivos, como son:

- Copiar la estructura fproc del padre en el hijo, de forma que el hijo herede los descriptores de archivo de su padre.
- Incrementar los contadores en la tabla filp.
- Incrementar el numero de procesos que utilizan el directorio raíz y el de trabajo.

- ***Funciones utilizadas*:**

➢ **dup_inode(*ip):** Forma simplificada de la rutina get_inode() del caso en que ya se conoce el nodo i. Lo que hace es incrementar el número de procesos que lo utilizan (/src/fs/inode.c).

- ***Variables globales**:*

who: Entero con el número de proceso del solicitante.
Parent: Tercer entero del mensaje de entrada tipo 1 que contendrá el número de entrada en fproc del proceso padre.
Child: Cuarto entero del mensaje de entrada tipo 1 que contendrá el número de entrada en fproc del proceso hijo.

- ***Variables locales*:**

***cp:** Apuntador a la estructura fproc que contendrá la dirección en la tabla fproc de la entrada del proceso ramificado.
***sptr**: Apuntador a un carácter que almacenará la dirección de la estructura fproc del proceso padre.
***dptr**: Apuntador a un carácter que almacenará la dirección de la estructura fproc del proceso hijo.
I: Entero utilizado para recorrer las 64 entradas de la tabla filp para incrementar el número de procesos que comparten esa ranura.

- ***Constantes*:**

MM_PROC_NR: Número de proceso del manejador de memoria (0).
NR_FDS: Número de descriptores de archivo por proceso (20).
NIL_FILP: Indica que una ranura filp está libre.

- ***Código***:

```
PUBLIC int do_fork()
```

```
{/* Perform those aspects of the fork() system call that relate to files.
In particular, let the child inherit its parent's file descriptors.
The parent and child parameters tell who forked off whom. The file
system uses the same slot numbers as the kernel.  Only MM makes this call.
 */

  register struct fproc *cp;
  int i;

/* Comprobamos que la llamada la ha realizado el MM */
  /* Only MM may make this call directly. */
  if (who ¡= MM_PROC_NR) return(EGENERIC);

/*Copia la estructura fproc del padre en  el hijo. */
  /* Copy the parent's fproc struct to the child. */
  fproc[child] = fproc[parent];

/* Incrementamos los contadores de la tabla filp que estaban  siendo utilizados anteriormente en el
proceso padre, ya que  a partir de ahora serán también utilizados por el proceso hijo */
  /* Increase the counters in the 'filp' table. */
  cp = &fproc[child];
  for (i = 0; i < OPEN_MAX; i++)
          if (cp->fp_filp[i] ¡= NIL_FILP) cp->fp_filp[i]->filp_count++;

/*rellena el campo fp_pid de la entrada f_proc del hijo con el nuevo identificador*/
  /* Fill in new process id. */
  cp->fp_pid = pid;

  /*un hijo nunca puede ser leader->marca de leader a off*/
  /* A child is not a process leader. */
  cp->fp_sesldr = 0;

/* Igualmente incrementamos el número de procesos que utilizan  el directorio raíz y el de trabajo
mediante la función  dup_inode() */
  /* Record the fact that both root and working dir have another user. */
  dup_inode(cp->fp_rootdir);
  dup_inode(cp->fp_workdir);
  return(OK);
}
```

4.5.- FUNCIÓN do_exec().

- ***Código***:

```
PUBLIC int do_exec()
{/*Los ficheros pueden ser marcados con FD_CLOSEXEC bit, el mapa de bits de los ficheros marcados
con closeexec se encuentra en el campo fp_closexex de la entrada del proceso a la estructura fproc.
Cuando el MM hace un EXEC, esta llama al FS que permite encontrar y cerrar los ficheros con close-
on.exec.

 register int i;
 long bitmap;

 /*comprueba que la llamada a sido hecha por el mm*/
 /* Only MM may make this call directly. */
 if (who != MM_PROC_NR) return(EGENERIC);

 /*se obtiene el mapa de bits de ficheros que deben ser cerrados*/
 /* The array of FD_CLOEXEC bits is in the fp_cloexec bit map. */
 fp = &fproc[slot1];                    /* get_filp() needs 'fp' */
 bitmap = fp->fp_cloexec;
 if (bitmap == 0) return(OK);           /* normal case, no FD_CLOEXECs */

/*va chequeando para cada descriptor utilizando el bitmap si deben ser cerrados, en caso afirmativo
do_close();*/
 /* Check the file desriptors one by one for presence of FD_CLOEXEC. */
 for (i = 0; i < OPEN_MAX; i++) {
        fd = i;
        if ( (bitmap >> i) & 01) (void) do_close();
 }

 return(OK);
}
```

4.6.- FUNCIÓN do_exit().

Su función es ejecutar la parte del sistema de archivo de la llamada al sistema exit (status). Esto, lo realiza de la siguiente forma:

- Si no es el manejador de la memoria el que realiza la llamada a EXIT, se produce un error y termina.
- Hace como si la llamada viniera del usuario.
- Si el proceso está pendiente y se suspendió en la tubería, entonces decrementa el contador de procesos pendientes y pone el proceso como no suspendido.
- Cierra todos los descriptores de archivo que estén abiertos.
- Libera los directorios raíz y de trabajo.
- Si se trata del exit de un proceso líder, entonces se revoca el acceso a este controlador de tty donde se encuentra el proceso líder de todos los procesos que lo estaban usando.

- ***Funciones utilizadas***:

- **do_unpause()**: Envía una señal a un usuario que ha hecho una pausa en el sistema de archivo (src/fs/pipe.c).

- **do_close()**: Realiza la llamada al sistema close(fd) que lleva a cabo el cierre de un archivo (src/fs/fopen.c).

- **put_inode(i)**: Libera el nodo "i" (src/fs/inode.c).

- ***Variables globales***:

slot1: m.m1_il. Parámetro del mensaje m1, que nos indica el número del proceso que hizo la llamada al sistema.
susp_count: Números de procesos suspendidos en la tubería.
fp: Puntero a la estructura fproc del solicitante.
Who: Número del proceso que realizó la llamada.

- ***Variables locales***:

i: Entero usado como contador
task: Tarea en la que se suspendió el proceso.

- ***Constantes***:

MM_PROC_NR: Número de proceso del manejador de memoria = 0.
OPEN_MAX: Número de descriptores de archivos máximo por proceso = 20.
SUSPENDED: El proceso está suspendido en la tubería o la tarea = 1.
NOT_SUSPENDED: El proceso no está suspendido en la tubería o la tarea = 0.
XPIPE: Se usa en fp_task cuando se suspende en la tubería.
XOPEN: Se usa en fp_task cuando se suspende en la tarea.

- ***Código***:

```
PUBLIC int do_exit()
{
/* Perform the file system portion of the exit(status) system call. */

  register int i, exitee, task;
  register struct fproc *rfp;
  register struct filp *rfilp;
  register struct inode *rip;
  int major;
  dev_t dev;
  message dev_mess;

  /*Solo el mm puede realizar la llamada directamente*/
  if (who != MM_PROC_NR) return(EGENERIC);

  /* Nevertheless, pretend that the call came from the user. */
```

```
  fp = &fproc[slot1];                   /* get_filp() needs 'fp' */
  exitee = slot1;
```

/* Si el proceso está pendiente y se suspendió en la tubería, decrementamos el contador de procesos pendientes. */

```
  if (fp->fp_suspended == SUSPENDED) {
          task = -fp->fp_task;
          if (task == XPIPE || task == XPOPEN) susp_count--;
          pro = exitee;
          (void) do_unpause();        /* this always succeeds for MM */
          fp->fp_suspended = NOT_SUSPENDED;
  }
```

/* Cierra todos los descriptores de archivo que estén abiertos*/

```
/* Loop on file descriptors, closing any that are open. */
  for (i = 0; i < OPEN_MAX; i++) {
          fd = i;
          (void) do_close();
  }
```

/* Quita el proceso del directorio raíz y del de trabajo.*/

```
  put_inode(fp->fp_rootdir);
  put_inode(fp->fp_workdir);
  fp->fp_rootdir = NIL_INODE;
  fp->fp_workdir = NIL_INODE;
```

/* si se trata del exit de una proceso lider entonces se revoca el acceso a este controlador de tty donde se encuentra el proceso leader de todos los procesos que lo estaban usando.*/

```
  /* If a session leader exits then revoke access to its controlling tty from
   * all other processes using it.
   */
  if (!fp->fp_sesldr) return(OK);               /* not a session leader */
  fp->fp_sesldr = FALSE;
  if (fp->fp_tty == 0) return(OK);              /* no controlling tty */
```

/* si se trata de una sesion lider se obtiene el identificador del terminal*/

```
  dev = fp->fp_tty;
```

/*para cada proceso se comprueba si esta utilizando ese terminal*/

```
  for (rfp = &fproc[LOW_USER]; rfp < &fproc[NR_PROCS]; rfp++) {
          if (rfp->fp_tty == dev) rfp->fp_tty = 0; /*si se esta utilizando se pone en desuso*/
```

/* Para cada entrada en la tabla filp de los procesos, se recupera su estructura i-node y se comprueba si el descriptor esta abierto, es un fichero especial y si se trata del mismo dispositivo que el lider, en caso de que esto sea cierto se manda un mensaje a la tarea del dispositivo para cerrarlo y se cierra el descriptor que lo utilizaba.*/

```
          for (i = 0; i < OPEN_MAX; i++) {
                  if ((rfilp = rfp->fp_filp[i]) == NIL_FILP) continue;
                  if (rfilp->filp_mode == FILP_CLOSED) continue;
                  rip = rfilp->filp_ino;
                  if ((rip->i_mode & I_TYPE) != I_CHAR_SPECIAL) continue;
                  if ((dev_t) rip->i_zone[0] != dev) continue;
                  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);
                rfilp->filp_mode = FILP_CLOSED;
        }
  }
  return(OK);
}
```

4.7.- FUNCIÓN do_set().

Su función es fijar el campo del identificador de usuario (UID) o el del identificador de grupo (GID). Sólo puede ser llamada por el MM.

- ***Variables globales***:

slot1: m.m1_il
who: Número del proceso del solicitante.
fs_call: Número de la llamada al sistema.
real_user_id: m.m1_i2. Identificador de usuario real.
eff_user_id: m.m1_i3. Identificador de usuario efectivo.
eff_grp_id: m.m1_i3. Identificador de grupo efectivo.
real_grp_id: m.m1_i2. Identificador de grupo real.

- ***Variables locales***:

tfp: Puntero a la estructura fproc

- ***Constantes***:

MM_PROC_NR: Número del proceso del MM = 0.
SETUID: contiene el número de dicha llamada al sistema = 23.
SETGID: contiene el número de dicha llamada al sistema = 46.

- ***Código***:

```
PUBLIC int do_set()
{
/* Fija el uid o gid */
  register struct fproc *tfp;

  /* Sólo el MM puede hacer esta llamada directamente. */
  if (who != MM_PROC_NR) return(ERROR);

  tfp = &fproc[slot1];
```

```
 /* Si la llamada al sistema es SETUID, fija los uid real y efectivo */
 if (fs_call == SETUID) {
          tfp->fp_realuid = (uid_t) real_user_id;
          tfp->fp_effuid =  (uid_t) eff_user_id;
 }

 /* Si la llamada al sistema es SETGID, fija los gid real y efectivo */
 if (fs_call == SETGID) {
          tfp->fp_effgid =  (gid_t) eff_grp_id;
          tfp->fp_realgid = (gid_t) real_grp_id;  }
 return(OK);
}
```

4.8.- FUNCIÓN do_revive().

Esta función se invoca cuando una tarea que antes no había podido completar un trabajo solicitado por el FS, como suministrar datos de entrada a un proceso de usuario, ya ha completado dicho trabajo. A continuación el FS revive el proceso y le envía el mensaje de respuesta.

- ***Funciones utilizadas***:

- **revive(proc_nr,bytes)**: Reactiva el proceso proc_nr, leyendo el número de bytes que se le indiquen, si está pendiente en una tarea.

- ***Variables globales***:

who: Número del proceso solicitante.
m: Variable de tipo message.
dont_replay: Entero. Normalmente igual a cero. Se pone a 1 para inhibir la respuesta.

- ***Constantes***:

REP_PROC_NR: m2_i1. Número del procedimiento en cuyo beneficio se hizo la E/S.

REP_STATUS: m2_i2. Bytes transferidos o número de error.

- **Código**:

```
PUBLIC int do_revive

{
/* Una tarea, comúnmente TTY (terminal), ha obtenido ahora los caracteres que se necesitaban en una
lectura anterior. El proceso no obtuvo una respuesta cuando hizo la llamada, al contrario, fue suspendido.
Ahora podemos enviar la respuesta para desbloquearlo. Esta tarea se tiene que hacer con cautela, ya que
el mensaje que llega proviene de una tarea (a la cual no se puede enviar ninguna respuesta y la respuesta
debe ir a un proceso que se bloqueó antes. La respuesta al solicitante se inhibe colocando la señal
"dont_replay" y la respuesta al proceso bloqueado se hace en forma explícita en revive(). */
```

```
  if (who > 0) return(EPERM);
  revive(m.REP_PROC_NR, m.REP_STATUS);
  dont_reply = TRUE;                    /* don't reply to the TTY task */
  return(OK);
}
```