LA TAREA DEL SISTEMA

Alberto Marínez Pérez

 C U R S O 2000/2001

INDICE

INTRODUCCIÓN:

Una consequencia de hacer el Sistema de Ficheros (**FS**) y el Manejador de Memoria (**MM**) fuera del Núcleo (**Kernel**) es que ocasionalmente algunas partes de información de estos son requeridas por el Núcleo. La estructura, sin embargo, prohíbe que se acabe escribiendo, por los anteriores , directamente dentro de la tabla del Núcleo.

Luego necesitamos una interface de comunicación entre:

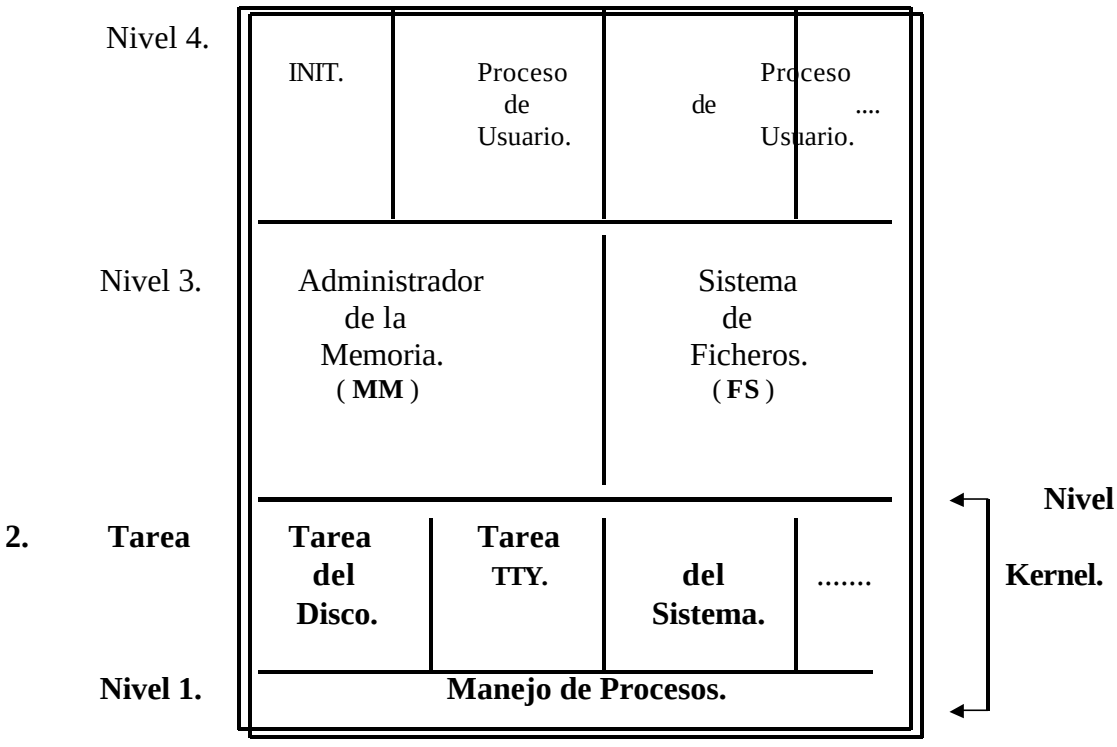
- Sistema de Ficheros. (FS)
- Mananejador de Memoria. (MM).

y el :

- Núcleo. (Kernel).

esta comunicación se produce mediante un mecanísmo estándard de mensaje el cual tiene acceso a toda la tabla del Núcleo, esto es a lo que denominamos Tarea del Sistema (system **task)** , el cual se haya en el **nivel 2** de la estructura por capas del Minix y funciona igual que las demás tareas, con la única diferencia que no controla ningún dispositivo de E/S.

Veamos la estructura por capas del Minix.:



Los servicios se obtienen realizando llamadas desde el MM o el FS de la forma:

SYS_XXX

a la Tarea del Sistema (system task) , y en él se especifica el tipo de servicio que se desea.

Los distintos tipos de mensajes se manejan por un proceso relacionado cuyo nombre vendrán en la forma.:

DO_XXX

por ejemplo, el Manejador de Memoria (MM) lanza un mensaje **sys_fork**, el mensaje, señal, será llevada a cabo por **do_fork**.

En resumen existen 3 tablas.:

- Tabla del Manejador de Mensajes.
- Tabla del Sistema de Ficheros.
- Tabla del Núcleo.

y en todas ellas la entrada k refiere a la misma posición. Resolviéndose de forma más eficiente el acceso a estas tablas..

TIPOS DE MENSAJES ACEPTADOS:

La Tarea del sistema acepta 19 clases-tipos de mensajes, los cuales son.:

Tipo de Mensaje.	**Procedente**.	**Significado.**
SYS_FORK	MM	Informa al Kernel que un proceso ha sido ramificado
SYS_NEWMAP	MM	Permite al MM actualizar el mapa de memoria de un proceso.
SYS_GETMAP	MM	Similar al anterior, sólo usado cuando el FS se incorpora inicialmente, lo que hace es requerir al Kernel información del proceso del mapeado.
SYS_EXEC	MM	Utilizado para colocar el contador de programa,PC y el puntero de pila, SP después de realizar una llamada exec.
SYS_XIT	MM	Informa al Núcleo que un proceso ha terminado exit.
SYS_GETSP	MM	Quiere leer el SP de un proceso.
SYS_TIMES	FS	Quiere leer los tiempos de ejecución de un pro--ceso.

SYS_ABORT	AMBOS	Pánico: Minix imposible continuar. El Minix abor--ta.
SYS_SENDSIG	MM	Envia una señal a un pro--ceso.
SYS_SIGRETURN	MM	Cuando termina una señal p.e.:sys_sendsig se encarga de reestaurar el registro del proceso señalado.Restaurándose la ejecución en el punto inte--rrumpido previamente .
SYS_KILL	FS	Envía la señal a un proce--so. En sí hace un chequeo del origen del mensaje.
SYS_ENDSIG	MM	Se encarga de reestaurar el registro de proceso se--ñalado después de una señal recibida del Núcleo.
SYS_COPY	AMBOS	Petición de la copia de un bloque de datos entre pro--cesos.
SYS_VCOPY	AMBOS	Petición de Copia de múl--tiples bloques de datos entre procesos.
SYS_GBOOT	FS	Copia los parámetros boot Generalmente usado cuan--do comienza el sistema.
SYS_MEM	MM	Petición del siguiente blo--que libre (chunk) de me--moria física.
SYS_UMAP	FS	Calcula la dirección física para una dirección virtual dada.
SYS_TRACE	MM	Petición de una operación de traza.

FICHEROS .H .:

Los tipos de mensajes declarados se encuentran en.:

/ usr/ include/ minix/ type.h

un ejemplo de las mismas serían.:

```
typedef struct{
                int m1i1,m1i2,m1i3;
                char *m1p1,*m2p2,*m3p3;
                }mess_1;

typedef struct{
                int m2i1,m2i2,m2i3;
                long m2l1,m2l2;
                char *m2p1;
                }mess_2;

typedef struct{
                int m3i1,m3i2;
                char *m3p1;
                char m3ca1[M3_STRING];
                }mess_3;

typedef struct{
                long m4l1,m4l2,m4l3,m4l4;
                }mess_4;

typedef struct{
                char m5c1,m5c2;
                int m5i1,m5i2;
                long m5l1,m5l2,m5l3;
                }mess_5;

typedef struct{
                int m6i1,m6i2,m6i3;
                long m6l1;
                void (*m6f1)();
                }mess_6;

typedef struct{
                int m_source;   /* quién envía
                el mensaje */
                int m_type;     /* tipo del
                mensaje       */
                union
                   {
                    mess_1 m_m1;
                    mess_2 m_m2;
                    mess_3 m_m3;
                    mess_4 m_m4;
                    mess_5 m_m5;
                    mess_6 m_m6;
                    }m_u;
                }message;
```

Los mensajes que se envían a la Tarea del Sistema suelen ser del Tipo m1, excepto.:

Tipo m2.:	SYS_TRACE
Tipo m5.:	SYS_UMAP, SYS_COPY
Tipo m6.:	SYS_ABORT, SYS_KILL.

También se utilizan defines para proporcionar nombres más cortos a los campos de la estructura de datos vistos antes.

```
#define m1_i1 m_u.m_m1.m1i1
#define m1_i2 m_u.m_m1.m1i2
#define m1_i3 m_u.m_m1.m1i3
#define m1_p1 m_u.m_m1.m1p1
#define m1_p2 m_u.m_m1.m1p2
#define m1_p3 m_u.m_m1.m1p3

#define m2_i1 m_u.m_m2.m2i1
#define m2_i2 m_u.m_m2.m2i2
#define m2_i3 m_u.m_m2.m2i3
#define m2_l1 m_u.m_m2.m2l1
#define m2_l2 m_u.m_m2.m2l2
#define m2_p1 m_u.m_m2.m2p1

#define m4_l1 m_u.m_m4.m4l1
#define m4_l2 m_u.m_m4.m4l2
#define m4_l3 m_u.m_m4.m4l3
#define m4_l4 m_u.m_m4.m4l4

#define m5_c1 m_u.m_m5.m5c1
#define m5_c2 m_u.m_m5.m5c2
#define m5_i1 m_u.m_m5.m5i1
#define m5_i2 m_u.m_m5.m5i2
#define m5_11 m_u.m_m5.m511
#define m5_12 m_u.m_m5.m512
#define m5_13 m_u.m_m5.m513

#define m6_i1 m_u.m_m6.m6i1
#define m6_i2 m_u.m_m6.m6i2
#define m6_i3 m_u.m_m6.m6i3
#define m6_l1 m_u.m_m6.m611
#define m6_f1 m_u.m_m6.m6f1
```

Además en el la ruta y fichero.:

/ usr/ include/ minix/ com.h

alguno de los anteriores campos son redefinidos.:

```
#define PROC1    m1_i1
#define PROC2    m1_i2


#define PID        m1_i3
#define PSTCK_PTR   m1_p1
```

```
#define MEM_PTR     m1_p1

#define DEVICE m2_i1
#define PROC_NR         m2_i3
#define COUNT m2_l1
#define POSITION        m2_l1
#define ADDRESS         m2_p1

#define SRC_SPACE       m5_c1
#define SRC_PROC_NR             m5_i1
#define SRC_BUFFER      m5_11
#define DST_SPACE       m5_c2
#define DST_PROC_NR m5_i2
#define DST_BUFFER      m5_12
#define COPY_BYTES      m5_13

#define USER_TIME m4_l1
#define USER_TIME m4_l2
#define USER_TIME m4_l3
#define USER_TIME m4_l4

#define PR              m6_i1
#define SIGNUM     m6_i2
#define FUNC            m6_F1
```

de esta forma es más fácil su comprensión.

PARÁMETROS PARA LOS TIPOS DE MENSAJES:

Los parámetros de los mensajes son los que vienen a continuación. (serán explicados en cada procedimiento más adelante.):

Mensaje.	**Proc1.**	**Proc2.**	**Pid.**	**Puntero.**
SYS_FORK	parent	child	pid	
SYS_NEWMAP	proc nr			map ptr
SYS_EXEC	proc nr	traced	new sp	
SYS_XIT	parent	exitee		
SYS_GETSP	proc nr			
SYS_TIMES	proc nr	buf ptr		
SYS_ABORT				
SYS_FRESH	proc nr	data_cl		
SYS_GBOOT	proc nr			boot ptr
SYS_GETMAP	proc nr			map ptr
m_tipo	**m1_i1**	**m1_i2**	**m1_i3**	**m1_p1**
SYS_VCOPY	src p	dst p	vec siz	vc addr
SYS_SENDSIG	proc nr			smp
SYS_SIGRETURN	proc nr			scp
SYS_ENDSIG	proc nr			
m_tipo	**m2_i1**	**m2_i2**	**m2_l1**	**m2_l2**
SYS_TRACE	proc_nr	request	addr	data

m_tipo	m6_i1	m6_i2	m6_i3	m6_f1
SYS_KILL	proc_nr	sig		

m_tipo	m5_c1	m5_i1	m5_l1	m5_c2	m5_i2	m5_l2	m5_l3
SYS_COPY	src seg	src proc	src vir	dst seg	dst proc	dst vir	byte ct
SYS_UMAP	seg	proc nr	vir adr				byte ct

m_tipo	m1_i1	m1_i2	m1_i3
SYS_MEM	mem base	mem size	tot mem

ENTRADAS AL PROGRAMA.:

En suma el punto pricipal de entrada al programa es SYS_TASK( ), aunque también hay otros 5 puntos de entrada pero de orden secundario comparado con la anterior, estos se encuentran al final del fichero system.c muy usados en varios momentos por el Núcleo y son.:

cause_sig: Cuando una tarea necesita lanzar una señal (p. e. el reloj una señal de alarma (sigalarm)), toma la acción para enviar la señal de la tarea hacia un proceso de usuario através del MM.procesa la señal activada, más tarde o más temprano.

inform: Comunica al MM las señales pendientes. En proc.c, cuando se envían o reciben mensajes se llama a sys_call que a su vez llama a la rutina mini_rec para recibir un mensaje y éste , a través de inform, avisa al MM de las señales pendientes para que el mensaje se pueda aceptar.

umap.: La usan todas las tareas que copian datos a o desde espacio de usario en la computadora o a una dirección física del buffer usan esta señal. No es más que para calcular la dirección física correspondiente de una virtual.

numap: Para que los drivers de los dispositivos puedan ser accedidos de forma adecuada, con el número del proceso , en vez de la que hace para que umap con el puntero en la tabla de procesos de entrada, ya que esto último (puntero) es desconocido para los drivers del dispositivo.

Se usa sobre todo en el main de MAIN.C, mem_task y do_mem de MEMORY.C, PRINTER.C, etc..

alloc_segments: Usado en la rutina principal del Kernel durante el proceso de iniciación. Es muy dependiente del Hardware. Toma los segmentos asignados que son registrados en la tabla de entrada de proceso y manipula los registro y descriptores que el procesador Pentium usa para suministrar protección a los segmentos del nivel Hardware.

PROGRAMA FUENTE.

• **Variables y Estructuras de Datos Externas.:**

message.: Estructura mensaje usada para la comunicación entre procesos.

• **Código Fuente.:**

/* Ficheros Include. */

```
#include "kernel.h"
#include <signal.h>
#include <unistd.h>
#include <sys/sigcontext.h>
#include <sys/ptrace.h>
#include <minix/boot.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"
#if (CHIP == INTEL)
#include "protect.h"
#endif
```

/* PSW máscara. */

```
#define IF_MASK 0x00000200
#define IOPL_MASK 0x003000
```

/* Variables Globales del Fichero.*/

```
PRIVATE message m;
```

/* Declaraciones por adelantado.*/

```
FORWARD _PROTOTYPE( int do_abort, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_copy, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_exec, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_fork, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_gboot, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_getsp, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_kill, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_mem, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_newmap, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_sendsig, (message *m_ptr) );
```

```
FORWARD _PROTOTYPE( int do_sigreturn, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_endsig, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_times, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_trace, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_umap, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_xit, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_vcopy, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_getmap, (message *m_ptr) );

#if (SHADOWING == 1)
FORWARD _PROTOTYPE( int do_fresh, (message *m_ptr) );
#endif
```

TAREA DEL SISTEMA SYS_TASK

- **Procedimientos Utilizados.:**

Primitivas de manejo de mensajes:
send(dest,&message): Envía un mensaje al proceso dest.
receive(source,&message): :Recibe un mensaje del proceso source.

- **<u>Código Fuente</u>.:**

```
PUBLIC void sys_task()
{
                        /*Punto de entrada principal. Recibe el mensaje de cualquier
                    fuente (ANY), determina el tipo de llamada al sistema a la que
                correponde y llama a la rutina que lo atiende.*/
 register int r;

 while (TRUE) {
        receive(ANY, &m);

        switch (m.m_type) {     /* Determina el tipo de mensaje.*/
           case SYS_FORK:       r = do_fork(&m);        break;
           case SYS_NEWMAP:     r = do_newmap(&m);      break;
           case SYS_GETMAP:     r = do_getmap(&m);      break;
           case SYS_EXEC:       r = do_exec(&m);        break;
           case SYS_XIT:        r = do_xit(&m);         break;
           case SYS_GETSP:      r = do_getsp(&m);       break;
           case SYS_TIMES:      r = do_times(&m);       break;
           case SYS_ABORT:      r = do_abort(&m);       break;
#if (SHADOWING == 1)
           case SYS_FRESH:      r = do_fresh(&m);       break;
#endif
           case SYS_SENDSIG:    r = do_sendsig(&m);     break;
           case SYS_SIGRETURN: r = do_sigreturn(&m);    break;
           case SYS_KILL:       r = do_kill(&m); break;
           case SYS_ENDSIG:     r = do_endsig(&m);      break;
           case SYS_COPY:       r = do_copy(&m);        break;
          case SYS_VCOPY:       r = do_vcopy(&m);       break;
           case SYS_GBOOT:      r = do_gboot(&m);       break;
           case SYS_MEM:        r = do_mem(&m);         break;
           case SYS_UMAP:       r = do_umap(&m);        break;
           case SYS_TRACE:      r = do_trace(&m);       break;
           default:             r = E_BAD_FCN;
        }

        m.m_type = r;           /* "r" informará del estado de la llamada.*/
        send(m.m_source, &m);   /* envía la respuesta al solicitante.*/
 }
}
```

DO_FORK

• **Tarea Llevada a Cabo.:**

Función encargada de llevar a cabo el manejo del mensaje correspondiente a una llamada.:

sys_fork( ).: Usado por el MM para comunicarle al Kernel la ramificación de un nuevo proceso (proceso-padre PROC1, proceso-hijo PROC2).
Ya que el Kernel lo debe reservar una entrada en su tabla de procesos para el procedimiento hijo creado.

do_fork ( ).: Un chequeo para ver si MM está introduciendo,en Kernel con procesos usuario correcto, existe ese PID en la tabla, para eso utiliza la macro **isoksusern**, que se haya definida en **proc.h**, entonces se copia la entrada en la tabla del proceso padre en la entrada reservada para el proceso hijo. En este momento se regulan algunas señales, tanto señales pendientes del padre para luego liberar al hijo, además el hijo no hereda la traza de estado del padre, por lo que los contadores del hijo deben ser puestos a 0.

• **Parámetros.:**

Parámetros punteros a un mensaje tipo1 con los siguientes campos.:

m_type.: SYS_FORK;
PROC1.: Identificador del proceso padre
PROC2.: Identificador del proceso hijo.
PID.: PID del proceso hijo.

· **Constantes.:**

NO_MAP.: Evita que un proceso-hijo se ejecute antes que se haya construído se mapa de memoria.
E_BAD_PROC.: El número del proceso es erróneo.

· **Variables Importantes.:**

rpc.: Puntero a una estructura de tipo proc. Va a contener la dirección de la entrada en la tabla de procesos del proceso hijo.
rpp : Idem pero para el proceso padre.

· **Variables y Estructura de Datos Externas.:**

Se utilizan algunas entradas de la Tabla de Procesos (array de estructuras con una entrada/registro por cada proceso/tarea).

p_nr: Número del proceso.

p_flags.: Se utilizan sus bits como señalización. Si es cero el proceso es ejecutable, si no lo es significa que alguno de sus bits está a 1 y el proceso no se puede ejecutar. Se utilizan los siguientes bits de p_flags.:

PENDING: Está a 1 cuando informa de una señal pendiente.

SIG_PENDING: Evita que el proceso a ser señalizado se ejecute.

P_STOP: Está a 1 cuando el proceso se está ejecutando en modo traza.

p_pendcount: Contador de señales pendientes.
p_pid: Identificador del proceso, pasado por el MM.
p_reg.retreg=0. {d0::registro máquina}
use_time: Tiempo de usuario en ticks.
sys_time: Tiempo de sistema en ticks.
child_utime: Tiempo de usuario acumulado por los hijos.
child_stime: Tiempo acumulado por los hijos.
p_ldt_sel: Selector en la Tabla Global de Descriptores de segmento(GDT), dando la base y el límite en la Tabla Local de Descriptores(LDT).
SHADOWING.: Variable utilizada para conocer si aún la tabla de memoria no ha sido construída. (1- Ya se ha construído. 0 - Caso contrario.)

· **Procedimientos Utilizados.:**

Externos:
i**sokusern** (nº proceso) : Verifica si el número de proceso de usuario es correcto.
proc_addr (nº proceso) : Obtiene el puntero a la entrada de la Tabla de Procesos para un proceso de usuario.
sigemptyset (señal pendiente).: Indica que la señal activada ya ha sido procesada.
mkshadow(rpp, (phys_clicks)m_ptr->m1_p1).:

· Código Fuente.:

```
PRIVATE int do_fork(m_ptr)
register message *m_ptr;          /* puntero a la estructura de mensaje.*/
{
                                  /* Manejo del mendaje tipo SYS_FORK.: m_ptr->PROC1 ha
                                  hecho un fork. El hijo es m_ptr->PROC2.*/
#if (CHIP == INTEL)
  reg_t old_ldt_sel;
#endif
  register struct proc *rpc;
  struct proc *rpp;
                                  /* Si el proceso padre o el proceso hijo no son procesos
entonces                          retorna un E_BAD_PROC.*/
  if (!isoksusern(m_ptr->PROC1) || !isoksusern(m_ptr->PROC2))
          return(E_BAD_PROC);
                                  /*Si son procesos entonces extrae la dirección en la tabla de
                                  procesos tanto del proceso padre como del hijo.*/
  rpp = proc_addr(m_ptr->PROC1);
  rpc = proc_addr(m_ptr->PROC2);
                                  /* Salva el selector de la tabla local de descriptores LDT del
                                  hijo.*/
#if (CHIP == INTEL)
  old_ldt_sel = rpc->p_ldt_sel;
#endif
                                  /* Copia la estructura del proceso padre en la del hijo.*/
  *rpc = *rpp;

#if (CHIP == INTEL)
  rpc->p_ldt_sel = old_ldt_sel;
#endif
                                  /* Se actualiza el número del proceso hijo, que se obtiene de un
                                  campo del mensaje. */
  rpc->p_nr = m_ptr->PROC2;
                                  /* Se inhibe el proceso-hijo se ejecute si no se ha construído aún
la                                tabla de memoria.*/
#if (SHADOWING == 0)
  rpc->p_flags |= NO_MAP;
#endif
                                  /* Anula los flags de señales pendientes y traza del proceso
hijo.*/
  rpc->p_flags &= ~(PENDING | SIG_PENDING | P_STOP);
                                  /* Únicamente uno el el grupo debería tener una señal
pendiente,                        el hijo no hereda el estado de la traza.*/
sigemptyset(&rpc->p_pending);
                                  /* Pone el contador de señales pendientes e inacabadas a 0.*/
  rpc->p_pendcount = 0;
                                  /* Inicializa el PID del proceso-hijo.*/
  rpc->p_pid = m_ptr->PID;
                                  /* indica el registro máquina d0 */ /* child sees pid = 0 to know it is child
*/
  rpc->p_reg.retreg = 0;
                                  /* Resetea todos los tiempos de cuenta del proceso hijo.*/
  rpc->user_time = 0;
  rpc->sys_time = 0;
  rpc->child_utime = 0;
```

```
  rpc->child_stime = 0;
                                        /* Se ha se ha construídola tabla dememoria.*/
#if (SHADOWING == 1)
  rpc->p_nflips = 0;
                                        /* por lo tanto se ejecuta el primer hijo. */
  mkshadow(rpp, (phys_clicks)m_ptr->m1_p1);
#endif

  return(OK);
}
```

DO_NEWMAP.

•Tarea Llevada a Cabo.:

Función encargada de llevar a cabo el manejo del mensaje correspondiente a una llamada.:

sys_newmap( ).: Después de un FORK, el MM asigna memoria para el proceso-hijo. El Kernel debe conocer dónde está el hijo localizado en memoria, tal que pueda establecer el registro adecuado cuando se ejecute el proceso-hijo. La señal sys_newmap permite al MM dar alKernel el mapa de memoria de cualquier proceso. Esta señal es muy usada en operaciones normales del istema MINIX.

do_newmap( ).: El cual lo primero que hace es copiar el nuevo mapa del espacio de direcciones del MM. Este mapa no va a darse en el propio mensaje ya que es demasiado grande. El mapa es copiado directamente dentro del campo *p_map* de la tabla de procesosde entrada para el proceso que tomará el nuevo mapeo. La llamada a **alloc_segmentes** extrae la información del mapa y lo carga dentro del campo *p_reg* que mantiene el segmento de registros.

• Parámetros.:

Parámetro es un puntero a un mensaje tipo 1 con los siguientes campos:

m_type: SYS_NEWMAP;
PROC1: Identificador del proceso para el que se va a crear el mapa de memoria.
MEM_PTR: Dirección del mapa dememoria en el área de datos de MM

· Constantes.:

E_BAD_PROC: El número de proceso es erróneo.
NO_NUM: No ha podido realizar la conversión de dirección virtual a física.

·Variables Importantes.:

caller: Número del proceso solicitante. Normalmente es el del MM.

k : Número del proceso para el que se va a crear el mapa de memoria.

map_ptr: Puntero a una estructura de tipo mem_map que tendrá la dirección virtual del mapa dentro del MM

rp : Puntero a la estructura proc. Contiene la dirección del slot del proceso para el que se va a crear el mapa de memoria.

src_phys: Variable tipo long. Contiene la dirección física del proceso solicitante (del MM).

old_flags: Guarda el valor de los flags antes de ser modificado.

· **Variables y Estructuras de Datos Externas.:**

Se utilizan algunas entradas de la **Tabla de Procesos** (array de estructuras con una entrada/registro por cada proceso/tarea).

p_flags: Se utilizan sus bits como señalización. Si es cero el proceso es ejecutable, si no lo es significa que alguno de sus bits está a 1 y el proceso no se puede ejecutar.

p_map: Array con tres campos (código,datos,pila) para guardar el mapa de memoria. Cada campo almacena a su vez tres valores: dirección virtual,
dirección física,
longitud del segmento.

SHADOWING.: Variable utilizada para conocer si aún la tabla de memoria no ha sido construída. (1- Ya se ha construído. 0 - Caso contrario.)

· **Procedimientos Utilizados.:**

Externos :

i**sokprocn** (nº proceso) : Verifica si el número de proceso de usuario es correcto.

proc_addr (nº proceso) : Obtiene el puntero a la entrada de la Tabla de Procesos para un proceso de usuario.

panic(mensaje error,identificador error)**:** Se ha producido un error irrecuperable. Provoca el término de la ejecución en Minix. (De Main.c).

phys_copy (fuente,destino,k)**:** Copia de la dirección fuente a la dirección destino k bytes. (De Klib88).

lock_ready (rp)**:** Coloca un proceso en la **Cola ready** (un proceso que ha pasado a ser ejecutable se coloca en la cola ready y se ejecutará en orden a su prioridad.Hay 3 listas de espera, una

para cada uno de los niveles**).** (De Proc.c).

umap: Calcula la dirección física de una virtual dada. (De System.c).

alloc_segments: Asigna los segmentos para un 8088 o procesador superior. (De System.c)

pmmu_init_prc.:

· **Código Fuente.:**

```
PRIVATE int do_newmap(m_ptr)
                                        /* Puntero al mensaje pedido.*/
message *m_ptr;
{
                                        /* Maneja sys_newmap(). Trae el mapa de memoria del
                                    MM.*/
  register struct proc *rp;
  phys_bytes src_phys;
  int caller;                       /* cuyo espacio tiene el nuevo mapa ( generalmente MM ). */
  int k;                            /* proceso cuyo mapa es el que va a ser cargado.*/
  int old_flags;                    /* valor del flags antes de ser modificado.*/
  struct mem_map *map_ptr;          /* dirección virtual del mapa dentro de caller ( MM ). */

                                    /* Extrae los parámetros del mensaje :
                                            nº de procesos que dá el mapa de memoria ( el MM ).
                                            proceso al que se le va a dar el mapa de memoria ( k ).
                                            dirección del mapa de memoria en el área de datos del
                                    MM.
                                    y copia el nuevo mapa de memoria del MM. */
  caller = m_ptr->m_source;
  k = m_ptr->PROC1;
  map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
                                    /* Si k ( proceso al que se le va a crear el mapa de memoria) no
es                                  un proceso entonces retorna.*/
  if (!isokprocn(k)) return(E_BAD_PROC);
                                    /* Calcula la dirección del proceso y al del MM ( el solicitante )
                                    en la Tabla de Procesos. */
  rp = proc_addr(k);
                                    /*  Va a copiar el mapa de memoria del MM.
                                    Para ello, primero calcula la dirección física de la virtual dada,
es                                  decir.:
                                            convierte la dirección virtual (vir_bytes) map_ptr del
                                    mapa de memoria,
                                            que está en el segmento de datos ( D )
                                            del MM ( proc_addr(caller) ) a dirección física
                                    si es 0 llama a Panic.*/
  src_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, sizeof(rp->p_map));
                                    /* Si no se ha podido, indica que se ha producido un error
                                irrecuperable provocando el término de la ejecución MINIX.*/
```

```
  if (src_phys == 0) panic("bad call to sys_newmap", NO_NUM);
                                        /* Si todo ha ido bien, copia el mapa de memoria del MM al del
                                    proceso. */
  phys_copy(src_phys, vir2phys(rp->p_map), (phys_bytes) sizeof(rp->p_map));
                                        /* Si la tabla de memoria aún no ha sido construída.*/
#if (SHADOWING == 0)
#if (CHIP != M68000)
                                        /* Actualiza los segmentos del proceso mapeado.*/
  alloc_segments(rp);
#else
  pmmu_init_proc(rp);
#endif
  old_flags = rp->p_flags;              /* Salva el valor previo de los flags.*/
                                        /* Permite que el proceso mapeado se ejecute ya que ha sido
                                    mapeado en memoria.*/
  rp->p_flags &= ~NO_MAP;
                                        /* Si todos los flags están a 0 entonces será ejecutable y se
coloca                                  en  la Cola Ready.*/
  if (old_flags != 0 && rp->p_flags == 0) lock_ready(rp);
#endif

  return(OK);
}
```

DO_GETMAP.

•Tarea Llevada a Cabo.:

Función encargada de llevar a cabo el manejo del mensaje correspondiente a una llamada.:

sys_getmap.:Únicamente usado cuando el FS se incorpora inicialmente. Podemos ver esta señal de forma similar a sys_newmap.

El procediminento hace, a grandes rasgos.:

do_getmap.: Lleva a cabo la transferencia de la información del mapa de procesos en la dirección opuesta a la que realizaba do_newmap.

El código de do_newmap y do_getmap son similares su diferencia principal en el intercambio de los argumentos fuente y destino de la llamada en *phys_copy* usada en cada función.

• Parámetros. Constantes. Variables Importantes. Variables y Estructuras de Datos Externas. Procedimientos Utilizados.:

Los anteriores puntos son idénticos a los del *do_newmap*.

· <u>Código Fuente</u>.:

Sólo será comentada la parte diferencial con respecto al código fuente de *do_newmap*.

```
PRIVATE int do_getmap(m_ptr)
message *m_ptr;
{
  register struct proc *rp;
  phys_bytes dst_phys;
  int caller;
  int k;
  struct mem_map *map_ptr;

  caller = m_ptr->m_source;
  k = m_ptr->PROC1;
  map_ptr = (struct mem_map *) m_ptr->MEM_PTR;

  if (!isokprocn(k))
          panic("do_getmap got bad proc: ", m_ptr->PROC1);

  rp = proc_addr(k);

  dst_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, sizeof(rp->p_map));
  if (dst_phys == 0) panic("bad call to sys_getmap", NO_NUM);
```

```
                        /* La siguiente función se diferencia de la anterior en que va a
                    convertir una dirección física en otra virtual.:
                        Para ello, primero calcula la dirección virtual de la física dada,
es                      decir.:
                            convierte la dirección física ( dst_phys),
                            en la dirección virtual ( vir2phys )
                        .*/
  phys_copy(vir2phys(rp->p_map), dst_phys, sizeof(rp->p_map));

 return(OK);
}
```

DO_EXEC.

•Tarea Llevada a Cabo.:

Función encargada de llevar a cabo el manejo del mensaje correspondiente a una llamada.:

do_exec.: Cuando un proceso hace una llamada al sistema EXEC con éxito, el MM debe componer una nueva imagen de la memoria incluyendo una nueva Pila para que contenga los argumentos y el entorno del proceso solicitante. A continuación, pasa el SP resultante al Kernel mediante la llamada sys_exec() .

Además de colocar el puntero de pila, do_exec pone a cero el Cronómetro de Alarma.

Por último, en la llamada a EXEC existe una particularidad: El proceso que invoca esta llamada envía un mensaje al manejador de la memoria y se bloquea esperando recibir respuesta (su flag RECEIVING esta a 1). A diferencia de otras llamadas al sistema en las que la respuesta desbloquea al proceso solicitante, en EXEC no hay respuesta, el MM no lo desbloquea y por tanto debe ser **do_exec** quien lo haga.

• Parámetros.

El parámetro es un puntero a un mensaje tipo 1 con los siguientes campos:

m_type: SYS_EXEC;
PROC1: Proceso que realiza la llamada EXEC.
PROC2: Indica si el proceso está en modo traza.
STACK_PTR: Puntero de pila del proceso solicitante.

· Constantes.:

E_BAD_PROC: El número de proceso es erróneo.
SIGTRAP: Señal que indica que un proceso está en modo traza.

· Variables y Estructuras de Datos Externas.:

Se utilizan algunas entradas de la **Tabla de Procesos** (array de estructuras con una entrada/registro por cada proceso/tarea).

p_flags: Se utilizan sus bits como señalización. Si es cero el proceso es ejecutable, si no lo es significa que alguno de sus bits está a 1 y el proceso no se puede ejecutar. Utilizaremos los siguientes bits de p_flags:

RECEIVING: Está a 1 cuando el proceso está bloqueado después de recibir un mensaje sin éxito.

p_alarm:Tiempo para la siguiente alarma en ticks, o cero..

p_reg.pc: Contiene los registros del proceso salvados en un stack.
p_reg.sp: Puntero de pila del proceso.
NLEN.: Contiene el tamaño del campo de los flags.

· **Procedimientos Utilizados.:**

Externos:
i**sokusern** (nº proceso) **:**Verifica si el número de proceso de usuario es correcto.
proc_addr (nº proceso) **:**Obtiene el puntero a la entrada de la Tabla de Procesos para un proceso de usuario.
lock_ready(rp)**:** Coloca un proceso en la **Cola ready**. (De Proc.c).
cause_sig: Su acción es enviar una señal de una tarea hacia un proceso de usuario. Lo llama el procedimiento do_click_tick de Clock.c para enviar una señal a un proceso cuya alarma ha expirado. En Minix un proceso puede solicitar al S.O que le interrumpa después de un cierto tiempo. (De System.c).
phys_copy (fuente,destino,k)**:**Copia de la dirección fuente a la dirección destino k bytes. (De Klib88).
numap: Calcula la dirección virtual de una física. (De System.c).

· **<u>Código Fuente</u>.:**

```
PRIVATE int do_exec(m_ptr)
register message *m_ptr;                    /* puntero al mensaje requerido.*/
{
                                            /* Se realiza el manejo del tipo de mensaje sys_exec().
Un                                          proceso ha hecho  un EXEC con éxito.*/
register struct proc *rp;
  reg_t sp;                                 /* Nuevo SP .*/
  phys_bytes phys_name;
  char *np;
#define NLEN (sizeof(rp->p_name)-1)
                                            /* Si no es un proceso el que ha hecho el EXEC retorna
                                          .*/
  if (!isoksusern(m_ptr->PROC1)) return E_BAD_PROC;
                                            /* El campo PROC2 es usado como un flag para indicar
                                          que al proceso se le está realizando una tarza.*/
  if (m_ptr->PROC2) cause_sig(m_ptr->PROC1, SIGTRAP);
                                            /* Se obtinen el SP del mensaje.*/
  sp = (reg_t) m_ptr->STACK_PTR;
                                            /* Se obtiene la dirección dentro de la tabla de procesos
                                            del proceso que ha realizado el EXEC.*/
  rp = proc_addr(m_ptr->PROC1);
  rp->p_reg.sp = sp;                        /* Coloca el nuevo SP al proceso que ha realizado el
                                          EXEC*/
```

```
#if (CHIP == M68000)
  rp->p_splow = sp;                     /* Coloca el SP en el mínimo ( low water ). */
#ifdef FPP
                                        /* Inicializa el FPP para este proceso.*/
  fpp_new_state(rp);
#endif
#endif
  rp->p_reg.pc = (reg_t) m_ptr->IP_PTR; /* Coloca el PC. */
  rp->p_alarm = 0;                      /* Resetea los timers de alarma.*/
  rp->p_flags &= ~RECEIVING;            /* Coloca el bit RECEIVING de los flags del proceso
que                                     hizo el EXEC a 0 para desbkloquearlo ya que el MM no
                                    lo desbloquea.*/
  if (rp->p_flags == 0) lock_ready(rp);     /* Si los flags están a 0 se coloca en la cola de ready. */
                                    /* Convierte una dirección física en virtual.*/
  phys_name = numap(m_ptr->m_source, (vir_bytes) m_ptr->NAME_PTR,
                                                        (vir_bytes) NLEN);
                                        /* Si la conversión se ha realizado sin problemas realiza
                                        la copia y se espera a pulsar F1 para que se actualize la
                                    tabla de procesos activos que hay en el sistema.*/
  if (phys_name != 0) {
          phys_copy(phys_name, vir2phys(rp->p_name), (phys_bytes) NLEN);
          for (np = rp->p_name; (*np & BYTE) >= ' '; np++) {}
          *np = 0;
  }
  return(OK);
}
```

DO_XIT.

•Tarea Llevada a Cabo.:

La salida o terminación de un proceso puede ocurrir de dos formas:

- Mediante una llamada **EXIT**.
- Al ser eliminado por una señal (**SIG_KILL**).

Tanto en un caso como en otro, el **MM** le mandará un mensaje de tipo **SYS_XIT** que será manejado por la función **do_xit()**. El proceso se eliminará de todas las colas en las que esté.

do_xit( ).: Su tarea es más complicada de lo que se puede esperar. Debe tener cuidado del estado de la información está en orden antes de actuar. La tarea del reloj siempre debe ser chequeada para ver si alguien va a ser afectado. Parte de la situación delicada del do_exit es: Si ocurre que el proceso que está siendo terminado va al ser encolado intenta enviar un mensaje (p.e.: el proceso fué terminado mediante una señal, antes que el EXIT), entonces debe se removido de la cola de mensajes con sumo cuidado.

• Parámetros.:

El parámetro es un puntero a un mensaje tipo 1 con los siguientes campos:

PROC1: Número del proceso padre.
PROC2:Número del proceso que ha hecho la salida.

· Constantes.:

E_BAD_PROC: Número del proceso erróneo.
BEG_PROC_ADDR: Comienzo de la Tabla de Procesos
END_PROC_ADDR: Final de la Tabla de Procesos
NIL_PROC: Indica que la lista de procesos que llaman está vacía.

· Variables Importantes.:

parent: Número del padre del proceso que ha salido.
proc_nr: Número del proceso que ha salido.

rp: Dirección de la tabla de procesos del padre e indice para recorrer la Tabla de Procesos.
rc: Dirección de la tabla de procesos del que sale

· **Variables y estructura de datos externas.:**

Se utilizan algunas entradas de la **Tabla de Procesos**:

p_flags: Se utilizan sus bits como señalización. Si es cero el proceso es ejecutable, si no lo es significa que alguno de sus bits está a 1 y el proceso no se puede ejecutar.
SENDING: Está a 1 cuando el proceso está bloqueado después de intentar infructuosamente enviar un mensaje
p_pendcount: Contador de señales pendientes.
use_time: Tiempo de usuario en ticks.
sys_time: Tiempo de sistema en ticks.
child_utime: Tiempo de usuario acumulado por los hijos.
child_stime: Tiempo acumulado por los hijos.
p_callerq: Cabeza de la lista de procesos llamadores de la entrada actual.
p_sendlink: Puntero al siguiente proceso de la lista de procesos llamadores.
p_pending: Mapa de bits de las señales pendientes (1-16) que todavía no se han enviado al MM.
p_alarm: Tiempo para la siguiente alarma en ticks, o cero.
SHADOWING.: Variable utilizada para conocer si aún la tabla de memoria no ha sido construída. (1- Ya se ha construído. 0 - Caso contrario.)

· **Procedimientos Utilizados.:**

Externos:
i**sokusern** (nº proceso) :Verifica si el número de proceso de usuario es correcto.
proc_addr (nº proceso) : Obtiene el puntero a la entrada de la Tabla de Procesos para un proceso de usuario.
lock_unready: (rp) Saca a un proceso de la **Cola ready.** (De Proc.c).
lock : Permite las interrupciones hardware. (De Klib88).
unlock : Impide las interrupciones hardware. (De Klib88).

· **Código fuente.:**

```
PRIVATE int do_xit(m_ptr)
message *m_ptr;
{
                                        /* Manejo del mensaje sys_xit. Un proceso ha salido.*/

 register struct proc *rp, *rc;
```

```
  struct proc *np, *xp;
  int parent;                   /* Nº del padre del proceso que ha salido.*/
  int proc_nr;                  /* Nº del proceso que ha hecho la salida.*/
  phys_clicks base, size;

  parent = m_ptr->PROC1;        /* Se obtiene el nº del slot del proceso padre.*/
  proc_nr = m_ptr->PROC2;       /* Se obtiene el nº del slot del proceso hijo que sale.*/
                                /* Si alguno no es un proceso se retorna E_BAD_PROC.*/
  if (!isoksusern(parent) || !isoksusern(proc_nr)) return(E_BAD_PROC);
                                /* Si no son procesos entonces se obtienen las direcciones de la
                             tabla de procesos.*/
  rp = proc_addr(parent);
  rc = proc_addr(proc_nr);
                                /* Añade en el padre los tiempos acumulados del hijo.*/
  lock();
  rp->child_utime += rc->user_time + rc->child_utime;
  rp->child_stime += rc->sys_time + rc->child_stime;
  unlock();
  rc->p_alarm = 0;              /* Apaga la alarma.*/
  if (rc->p_flags == 0) lock_unready(rc);

#if (SHADOWING == 1)
  rmshadow(rc, &base, &size);
  m_ptr->m1_i1 = (int)base;
  m_ptr->m1_i2 = (int)size;
#endif

  strcpy(rc->p_name, "<noname>");
                                        /* Si ocurre que el proceso que está siendo terminado va
                                        al ser encolado intenta enviar un mensaje ( p.e.: el
                                     proceso fué terminado mediante una señal, antes que el
                             EXIT ), entonces debe se removido de la cola de
                        mensajes. */
  if (rc->p_flags & SENDING) {
                                        /* Chequea todos los slots de la tabla de procesos para
ver                                     si el proceso que sale está en su cola.*/
          for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
                  if (rp->p_callerq == NIL_PROC) continue;
                                        /* El proceso que sale está en la cabecera de la cola de
                                     procesos llamadores al proceso actual del bucle ( rp ). Si
                                     es así se extrae*/

                  if (rp->p_callerq == rc) {
                          rp->p_callerq = rc->p_sendlink;
                          break;
                  } else {
                                        /* Se mira si el proceso que sale está en la cola,
                                     recorriéndose esta y eliminándose se está.*/
                          np = rp->p_callerq;
                          /* Mientras no esté vacía.*/
                          while ( ( xp = np->p_sendlink) != NIL_PROC)
                                  if (xp == rc) {



                                        np->p_sendlink = xp->p_sendlink;
                                        break;
```

```
                                        } else {
                                                np = xp;
                                        }
                                }
                        }
                }
#if (CHIP == M68000) && (SHADOWING == 0)
  pmmu_delete(rc);                      /* donde se realiza el ajuste de las tablas.*/
#endif

  if (rc->p_flags & PENDING) --sig_procs;
                                        /*Coloca el mapa de bits de señales pendientes del
                                        proceso que sale y el contador de señales pendientes e
                                inacabadas a 0.*/
 sigemptyset(&rc->p_pending);
  rc->p_pendcount = 0;
                                        /* El slot que ocupaba el proceso que salió se marca
como                                    libre.*/
  rc->p_flags = P_SLOT_FREE;
  return(OK);
}
```

DO_GETSP.

•Tarea Llevada a Cabo.:

Utilizado por el **MM** para conocer el Puntero de Pila actual de un proceso. Este valor se necesita en las llamadas al sistema **BRK y SBRK** para ver si el segmento de Datos y el de Pila han entrado en conflicto.

También se necesita cuando se produce una señal a un proceso para actualizar el Top de éste.

· Parámetros.:

El parámetro es un puntero a un mensaje tipo 1 con los siguientes campos:

PROC1: Identificador del proceso que desea conocer su SP.

· Constantes.:

E_BAD_PROC: Número de proceso erróneo.

· Variables y Estructura de Datos Externas.:

Utilizamos la siguiente entrada de la Tabla de Procesos:

p_reg.sp: Puntero de pila del proceso.

· Procedimientos Utilizados.:

Externos:

i**sokusern** (nº proceso) : Verifica si el número de proceso de usuario es correcto.

proc_addr (nº proceso) : Obtiene el puntero a la entrada de la Tabla de Procesos para un proceso de usuario.

• <u>Código Fuente</u>.:

```
PRIVATE int do_getsp(m_ptr)
register message *m_ptr;
{
```

```
                                        /* Manejo del tipo de mensaje sys_getsp. MM quiere
                                    conocer el SP de un determinado proceso.*/
  register struct proc *rp;
                                        /* Si el nº de proceso del que se desea conocer el SP no
es                                      correcto retorna E_BAD_PROC.*/
  if (!isoksusern(m_ptr->PROC1)) return(E_BAD_PROC);
                                        /* Obtiene la dirección de la tabla de procesos.*/
  rp = proc_addr(m_ptr->PROC1);
                                        /* Devuelve el SP en el propio mensaje.*/
  m_ptr->STACK_PTR = (char *) rp->p_reg.sp;
  return(OK);
}
```

DO_TIMES.

•Tarea Llevada a Cabo.:

Maneja el único tipo de mensaje usado exclusivamente por el **FS**. Se necesita para implementar la llamada al sistema **sys_times()**, que devuelve la contabilidad de los tiempos de un proceso al solicitante (el FS).

do_times( ).: Todo lo que hace es colocar los tiempos solicitados en el mensaje respuesta.

· Parámetros.:

El parámetro es un puntero a un mensaje tipo 1 con los siguientes campos:

PROC1: Identificador del proceso solicitante, en este caso el FS.

Do_times devuelve la contabilidad de los tiempos del proceso solicitante en un puntero a buffer,(la información corresponde a mensajes tipo 4) en el cual se encuentra la información sobre los tiempos requeridos.:

Tiempo de usuario consumido por el proceso. (USER_TIME).
Tiempo de sistema consumido por el proceso. (SYSTEM_TIME).
Tiempo de usuario consumido por el proceso hijo. (CHILD_UTIME).
Tiempo de sistema consumido por el proceso hijo. (CHILD_STIME).

· Constantes.:

E_BAD_PROC:Número del proceso erróneo.

· Variables y Estructura de Datos externas.:

Se utilizan algunas entradas de la **Tabla de Procesos**:

use_time: Tiempo de usuario en ticks.
sys_time:Tiempo de sistema en ticks.
child_utime: Tiempo de usuario acumulado por los hijos.
child_stime: Tiempo acumulado por los hijos.

· Procedimientos Utilizados.:

Externos:

isokusern (nº proceso) : Verifica si el número de proceso de usuario es correcto.

proc_addr (nº proceso) : Obtiene el puntero a la entrada de la Tabla de Procesos para un proceso de usuario.

lock : Permite las interrupciones hardware. (De Klib88).

unlock : Impide las interrupciones hardware. (De Klib88).

· **Código Fuente.:**

```
PRIVATE int do_times(m_ptr)
register message *m_ptr;
{
                                        /* Manejo del tipo de mensaje sys_times.*/
 register struct proc *rp;
                                        /*Si el número del procceso del cual se quieren
                                    obtener los tiempos no es correcto, se retorna
                                E_BAD_PROC. */
 if (!isoksusern(m_ptr->PROC1)) return E_BAD_PROC;
                                        /*Se obtiene la dirección en la tabla de procesos
                                        del proceso del que se quiere extraer los
                                        tiempos.*/
 rp = proc_addr(m_ptr->PROC1);
                                        /*Inserta los 4 tiempos necesitados por la
                                    llamada al sistema TIMES en el mensaje.*/
 lock();
 m_ptr->USER_TIME   = rp->user_time;
 m_ptr->SYSTEM_TIME = rp->sys_time;
 unlock();
 m_ptr->CHILD_UTIME = rp->child_utime;
 m_ptr->CHILD_STIME = rp->child_stime;
 m_ptr->BOOT_TICKS  = get_uptime();
 return(OK);
}
```

DO_ABORT

•Tarea Llevada a Cabo.:

Se encarga de finalizar el MINIX cuando el **MM** o **FS** detectan un error irrecuperable. Por ejemplo, si al iniciar el sistema el **FS** detecta que el superbloque del dispositivo raiz está corrompido, envía un mensaje **SYS_ABORT** al núcleo. También puede ocurrir que el superusuario quiera volver al monitor de arranque y/o reiniciar el sistema, usando el comando **reboot**. En cualquiera de estos casos, la tarea del sistema ejecuta el procedimiento **do_abort** que copia instrucciones al monitor, si fuera necesario, y luego llama a **wreboot** para completar el proceso.

· Parámetros.:

El parámetro es un puntero a un mensaje tipo 6 en el que se indica si se ha de invocar al monitor antes de reiniciar.

· Procedimientos Utilizados.:

numap(num_proceso, dir_virtual, num_bytes_requeridos): Calcula la dirección física correspondiente a una dirección virtual de un proceso dado.

panic(mensaje_error, id_error): Provoca el fin de la ejeución de MINIX

phys_copy(fuente, destino, num_bytes): Copia num_bytes de la fuente al destino.

vir2phys(dir_virtual): Convierte una dirección virtual en su equivalente físico.

wreboot(m_ptr->m1_i1): Espera a que se pulse una tecla antes de resetea.

· Código Fuente.:

```
PRIVATE int do_abort(m_ptr)
message *m_ptr;                    /* puntero al mensaje SYS_ABORT */
{
 char monitor_code[64];
 phys_bytes src_phys;

 /* reiniciar con el monitor */
  if (m_ptr->m1_i1 == RBT_MONITOR) {
        /* Comprueba la validez de la dirección del código a ejecutar */
        src_phys=numap(m_ptr->m_source,(vir_bytes)m_ptr->m1_p1,
                    (vir_bytes) sizeof(monitor_code));
       if(src_phys == 0)
         panic("bad monitor code from", m_ptr->m_source);
         phys_copy(src_phys, vir2phys(monitor_code),
                 (phys_bytes) sizeof(monitor_code));
         reboot_code = vir2phys(monitor_code);
```

```
  }

  wreboot(m_ptr->m1_i1);
  return(OK);
}
```

DO_SENDSIG

•Tarea Llevada a Cabo.:

La mayor parte del manejo de señales se hace en el **MM**. Este verifica si el proceso que es señalado está capacitado para capturar o ignorar la señal, si el emisor de la señal tiene derecho a hacerlo, etc. Lo que no puede hacer el **MM** es producir realmente la señal ya que eso requiere modificar la pila del proceso señalado. Este trabajo se efectúa enviando el mensaje **SYS_SENDSIG** a la tarea del sistema **sys_task** para que sea manejado por **do_sendsig**.

La información necesaria para manejar las señales se encuentra en una estructura de typo **sigcontext**, que incluye el contenido de los registros del procesador, y una estructura de typo **sigframe**, que contiene información relativa a cómo han de tratarse las señales por parte de los procesos. Ambas estructuras necesitan cierta inicialización pero el trabajo esencial de **do_sendsig** es colocar la información en la pila del proceso señalado y ajustar el contador de programa y el puntero de pila de forma que el código correspondiente al tratamiento de la señal sea lo próximo que ejecute el proceso señalado.

· Parámetros.:

El parámetro es un puntero a un mensaje tipo 1 en el que se incluye un puntero al mensaje que se quiere enviar.

· Constantes.:

MM_PROC_NR: Número de proceso del MM.
SIG_CTXT_PTR: puntero para recuperar el contexto de la señal.
NO_NUM: argumento numérico para la función panic().

· Procedimientos Utilizados.:

isokusern(num_proceso): Verifica si el número de proceso es correcto.

proc_addr(num_proceso): Obtiene un puntero a la Tabla de Procesos.

umap(dir_proceso, segmento, dir_virtual, bytes_requeridos): Calcula la dirección física del proceso a partir de su dirección virtual.

vir2phys(dir_virtual): Convierte una dirección virtual en su equivalente físico.

panic(mensaje_error, id_error): Provoca el término de la ejeución de MINIX

phys_copy(fuente, destino, num_bytes): Copia num_bytes de la dirección fuente a la direccción destino.

· Código Fuente.:

```
PRIVATE int do_sendsig(m_ptr)
message *m_ptr;                    /* puntero al mensaje SYS_SENDSIG */
{
 struct sigmsg smsg;
 register struct proc *rp;
 phys_bytes src_phys, dst_phys;
 struct sigcontext sc, *scp;
 struct sigframe fr, *frp;

 /* Comprueba la validez del número de proceso de usuario */
 if (!isokusern(m_ptr->PROC1))
   return(E_BAD_PROC);

 /* Obtiene un puntero a la entrada de la tabla de procesos */
 rp = proc_addr(m_ptr->PROC1);

 /* Obtiene la dirección física de la estructura sigmsg dentro del espacio de direcciones
  virtual del MM */
 src_phys=umap(proc_addr(MM_PROC_NR), D,
                (vir_bytes)m_ptr->SIG_CTXT_PTR, (vir_bytes) sizeof(struct sigmsg));
 if (src_phys == 0)
        panic("do_sendsig can't signal: bad sigmsg address from MM", NO_NUM);

 /* Copia la estructura sigmsg asociada al proceso señalado en el espacio de direcciones
    del procedimiento */
 phys_copy(src_phys, vir2phys(&smsg), (phys_bytes) sizeof(struct sigmsg));

 /* Calcula el valor de SP para poder guardar la estructura sigcontext. */
 scp = (struct sigcontext *) smsg.sm_stkptr - 1;

 /* Copia los registros a la estrutura sigcontext. */
 memcpy(&sc.sc_regs, &rp->p_reg, sizeof(struct sigregs));

 /* Termina la inicialización de la estructura sigcontext. */
 sc.sc_flags = SC_SIGCONTEXT;          /* para indicar que se incluye el contexto de la señal */
 sc.sc_mask = smsg.sm_mask;

 /* Copia la estructura sigcontext en la pila del usuario. */
 dst_phys = umap(rp, D, (vir_bytes) scp, (vir_bytes) sizeof(struct sigcontext));
 if (dst_phys == 0)
   return(EFAULT);
 phys_copy(vir2phys(&sc), dst_phys, (phys_bytes) sizeof(struct sigcontext));

 /* Inicializa la estructura sigframe. */
 frp = (struct sigframe *) scp - 1;
 fr.sf_scpcopy = scp;
 fr.sf_retadr2= (void (*)()) rp->p_reg.pc;
 fr.sf_fp = rp->p_reg.fp;
 rp->p_reg.fp = (reg_t) &frp->sf_fp;
 fr.sf_scp = scp;
 fr.sf_code = 0;  /*debería usarse para el tipo de excepción de punto flotante */
```

```
  fr.sf_signo = smsg.sm_signo;
  fr.sf_retadr = (void (*)()) smsg.sm_sigreturn;

  /* Copia la estructura sigframe en la pila del usuario. */
  dst_phys = umap(rp, D, (vir_bytes) frp, (vir_bytes) sizeof(struct sigframe));
  if (dst_phys == 0)
    return(EFAULT);
  phys_copy(vir2phys(&fr), dst_phys, (phys_bytes) sizeof(struct sigframe));

  /* Cambia los registros del proceso señalado para que continúe su ejecución en la rutina
  de tratamiento de la señal. */
  rp->p_reg.sp = (reg_t) frp;
  rp->p_reg.pc = (reg_t) smsg.sm_sighandler;
  return(OK);
}
```

DO_SIGRETURN

•Tarea Llevada a Cabo.:

Al final de toda rutina de tratamiento de señales se incluye una instrucción de retorno que, debido a la manipulación de la pila por parte de la rutina **do_sendsig**, produce la ejecución de la llamada del sistema **SIGRETURN**. El **MM** envía a la tarea del sistema un mensaje **SYS_SIGRETURN** que es tratado por **do_sigreturn** copiando la estructura **sigcontext** de vuelta al espacio de memoria del núcleo y recuperando los registros del proceso señalado. El proceso señalado continuará su ejecución en el punto en que fue interrumpido.

· Constantes.:

MM_PROC_NR: Número de proceso del MM.
SIG_CTXT_PTR: puntero para recuperar el contexto de la señal.
NO_NUM: argumento numérico para la función panic().

· Procedimientos Utilizados.:

isokusern(num_proceso): Verifica si el número de proceso es correcto.

proc_addr(num_proceso): Obtiene el puntero a la entrada de la Tabla de Procesos.

umap(dir_proceso, segmento, dir_virtual, bytes_requeridos): Calcula la dirección física correspondiente a una dirección virtual de un proceso dado.

vir2phys(dir_virtual): Convierte una dirección virtual en su equivalente físico.

phys_copy(fuente, destino, num_bytes): Copia num_bytes de la dirección fuente a la direccción destino.

· Código Fuente.:

```
PRIVATE int do_sigreturn(m_ptr)
register message *m_ptr;                    /* puntero al mensaje SYS_SIGRETURN */
{
 struct sigcontext sc;
 register struct proc *rp;
 phys_bytes src_phys;

 /* Comprueba la validez del número de proceso */
 if (!isokusern(m_ptr->PROC1))
   return(E_BAD_PROC);

 /* Obtiene un puntero a la entrada en la tabla de procesos asociada al proceso */
 rp = proc_addr(m_ptr->PROC1);
```

```
  /* Recupera la estructura sigcontext. */
  src_phys = umap(rp, D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
                        (vir_bytes) sizeof(struct sigcontext));
  if (src_phys == 0)
    return(EFAULT);
  phys_copy(src_phys, vir2phys(&sc), (phys_bytes) sizeof(struct sigcontext));

  /* Comprueba que no se trate de un jmp_buf. */
  if ((sc.sc_flags & SC_SIGCONTEXT) == 0)
    return(EINVAL);

  /* Ajusta algunos registros si el compilador no utiliza registros dentro de las funciones
   que contienen setjmp */
  if (sc.sc_flags & SC_NOREGLOCALS) { /* distinto de cero cuando no hay que gardar los registros */
            rp->p_reg.retreg = sc.sc_retreg;
            rp->p_reg.fp = sc.sc_fp;
            rp->p_reg.pc = sc.sc_pc;
            rp->p_reg.sp = sc.sc_sp;
            return (OK);
  }
  sc.sc_psw  = rp->p_reg.psw;

#if (CHIP == INTEL)
  sc.sc_cs = rp->p_reg.cs;
  sc.sc_ds = rp->p_reg.ds;
  sc.sc_es = rp->p_reg.es;
#if _WORD_SIZE == 4
  sc.sc_fs = rp->p_reg.fs;
  sc.sc_gs = rp->p_reg.gs;
#endif
#endif

  /* Recupera los registros. */
  memcpy(&rp->p_reg, (char *)&sc.sc_regs, sizeof(struct sigregs));

  return(OK);
}
```

DO_KILL

•Tarea Llevada a Cabo.:

Algunas señales se generan en el núcleo del sistema o bien son tratadas por él antes de pasar al **MM** (por ejemplo, las señales generadas por el la tarea del reloj o las señales generadas por el **FS**). El mensaje **SYS_KILL** lo utiliza el **FS** para solicitar que se genere una de éstas señales. Este mensaje será tratado por el procedimiento **do_kill** quien llamará a **cause_sig** para enviar realmente la señal al proceso. Las señales originadas en el núcleo tambien pasan a través de esta función quien inicia la señal enviando un mensaje **KSIG** al **MM**.

· Procedimientos Utilizados.:

isokusern(num_proceso): Verifica que el número de proceso de usuario es correcto.

cause_sig(num_proceso, id_señal): Envía la señal al proceso vía MM.

· Código Fuente.:

```
PRIVATE int do_kill(m_ptr)
register message *m_ptr; /* puntero al mensaje SYS_KILL */
{
  /* Comprueba la validez del número de proceso */
  if (!isokusern(m_ptr->PR))
     return(E_BAD_PROC);

  /* Produce la señal */
  cause_sig(m_ptr->PR, m_ptr->SIGNUM);
  return(OK);
}
```

DO_ENDSIG

•Tarea Llevada a Cabo.:

Cuando el **MM** ha terminado con una de las señales tipo **KSIG** envía un mensaje **SYS_ENDSIG** de vuelta a la tarea del sistema. Este mensaje lo trata el procedimiento **do_endsig**, el cual decrementa la cuenta de señales pendientes y, si llega a cero, resetea el bit **SIG_PENDING** del proceso señalado.

· Procedimientos Utilizados.:

isokusern(num_proceso): Verifica que el número de proceso de usuario es correcto.

proc_addr(num_proceso): Obtiene el puntero a la entrada de la Tabla de Procesos.

lock_ready(proceso): Coloca al proceso en la cola de procesos listos para ejecución.

· Código Fuente.:

```
PRIVATE int do_endsig(m_ptr)
register message *m_ptr; /* puntero al mensaje SYS_ENDSIG */
{
 register struct proc *rp;

 /* Comprueba la validez del número de proceso de usuario */
 if (!isokusern(m_ptr->PROC1))
   return(E_BAD_PROC);
 rp = proc_addr(m_ptr->PROC1);

 /* El MM ha terminado un mensaje KSIG y si no hay nada que lo impida, se coloca
 al proceso en la lista de procesos listos para ejecución. */
 if (rp->p_pendcount != 0 && --rp->p_pendcount == 0
    && (rp->p_flags &= ~SIG_PENDING) == 0)
          lock_ready(rp);
 return(OK);
}
```

DO_COPY

•Tarea Llevada a Cabo.:

El mensaje **SYS_COPY** es el más ampliamente utilizado ya que se necesita para permitir que el **FS** y el **MM** copien información desde y hacia los procesos de usuario. Por ejemplo, cuando un usuario hace una llamada **READ**, el **FS** verifica si tiene el bloque requerido en la cache y si no lo tiene envía un mensaje a la tarea del disco para cargarlo en la cache. A continuación, el **FS** envía un mensaje a la tarea del sistema para que copie el bloque al proceso de usuario. El tratamiento de **SYS_COPY** lo realiza el procedimiento **do_copy** y consiste, básicamente, en extraer los parámetros del mensaje y llamar a **phys_copy**.

· Procedimientos Utilizados.:

umap(dir_proceso, segmento, dir_virtual, bytes_requeridos): Calcula la dirección física correspondiente a una dirección virtual de un proceso dado.

panic(mensaje_error, id_error): Provoca el término de la ejeución de MINIX

proc_addr(num_proceso): Obtiene el puntero a la entrada de la Tabla de Procesos.

phys_copy(fuente, destino, num_bytes): Copia num_bytes de la dirección fuente a la direccción destino.

· Código Fuente.:

```
PRIVATE int do_copy(m_ptr)
register message *m_ptr; /* puntero al mensaje SYS_COPY */
{
  int src_proc, dst_proc, src_space, dst_space;
  vir_bytes src_vir, dst_vir;
  phys_bytes src_phys, dst_phys, bytes;

  /* Descompone el mensaje. */
  src_proc = m_ptr->SRC_PROC_NR;
  dst_proc = m_ptr->DST_PROC_NR;
  src_space = m_ptr->SRC_SPACE;
  dst_space = m_ptr->DST_SPACE;
  src_vir = (vir_bytes) m_ptr->SRC_BUFFER;
  dst_vir = (vir_bytes) m_ptr->DST_BUFFER;
  bytes = (phys_bytes) m_ptr->COPY_BYTES;

  /* Calcula las direcciones origen y destino y realiza la copia. */
#if (SHADOWING == 0)
  if (src_proc == ABS)
          src_phys = (phys_bytes) m_ptr->SRC_BUFFER;
  else {
```

```
        if (bytes != (vir_bytes) bytes)
                /* Esto ocurriría para segmentos de 64K y direcciones vir_bytes de
                16 bits. */
                panic("overflow in count in do_copy", NO_NUM);
#endif
        src_phys = umap(proc_addr(src_proc), src_space, src_vir,  (vir_bytes) bytes);
#if (SHADOWING == 0)
        }
#endif

#if (SHADOWING == 0)
  if (dst_proc == ABS)
        dst_phys = (phys_bytes) m_ptr->DST_BUFFER;
  else
#endif
        dst_phys = umap(proc_addr(dst_proc), dst_space, dst_vir, (vir_bytes) bytes);

  if (src_phys == 0 || dst_phys == 0)
    return(EFAULT);
  phys_copy(src_phys, dst_phys, bytes);
  return(OK);
}
```

DO_VCOPY

•Tarea Llevada a Cabo.:

Para intentar superar las ineficiencias del mecanismo de paso de mensajes se define un mensaje, **SYS_VCOPY**, que permite empaquetar múltipes peticiones en un único mensaje. El contenido de este mensaje es un puntero a un vector especificando múltiples bloques a copiar entre diferentes zonas de memoria. El procedimiento **do_vcopy** ejecuta un bucle extrayendo las direcciones fuente y destino y el tamaño de los bloques y llama a **phys_copy** repetidamente hasta que se hayan realizado todas las copias.

· Procedimientos Utilizados.:

numap(num_proceso, dir_virtual, num_bytes_requeridos): Calcula la dirección física correspondiente a una dirección virtual de un proceso dado.

phys_copy(fuente, destino, num_bytes): Copia num_bytes de la dirección fuente a la direccción destino.

vir2phys(dir_virtual): Convierte una dirección virtual en su equivalente físico.

· Código Fuente.:

```
PRIVATE int do_vcopy(m_ptr)
register message *m_ptr; /* puntero al mensaje SYS_VCOPY */
{
  int src_proc, dst_proc, vect_s, i;
  vir_bytes src_vir, dst_vir, vect_addr;
  phys_bytes src_phys, dst_phys, bytes;
  cpvec_t cpvec_table[CPVEC_NR];

  /* Descompone el mensaje. */
  src_proc = m_ptr->m1_i1;
  dst_proc = m_ptr->m1_i2;
  vect_s = m_ptr->m1_i3;
  vect_addr = (vir_bytes)m_ptr->m1_p1;

  if (vect_s > CPVEC_NR)
    return EDOM;

  /* Recupera la dirección del vector */
  src_phys= numap (m_ptr->m_source, vect_addr, vect_s * sizeof(cpvec_t));
  if (!src_phys)
     return EFAULT;
  phys_copy(src_phys, vir2phys(cpvec_table), (phys_bytes) (vect_s * sizeof(cpvec_t)));

  /* Ejecuta un bucle para copiar todos los bloques de memoria */
  for (i = 0; i < vect_s; i++) {
```

```
            src_vir= cpvec_table[i].cpv_src;
            dst_vir= cpvec_table[i].cpv_dst;
            bytes= cpvec_table[i].cpv_size;
            src_phys = numap(src_proc,src_vir,(vir_bytes)bytes);
            dst_phys = numap(dst_proc,dst_vir,(vir_bytes)bytes);
            if (src_phys == 0 || dst_phys == 0)
              return(EFAULT);
            phys_copy(src_phys, dst_phys, bytes);
  }
  return(OK);
}
```

DO_GBOOT

•Tarea Llevada a Cabo.:

Durante el arranque del sistema, el **FS** envía un mensaje **SYS_GBOOT** para solicitar los parámetros de arranque, una estructura **bparam_s**. El procedimiento **do_gboot** es el encargado de tratar este mensaje, copiando información de una zona de memoria a otra.

· Variables externas.:

boot_parameters: estructura que contiene los parámetros de arranque.

· Procedimientos Utilizados.:

proc_addr(num_proceso): Obtiene el puntero a la entrada de la Tabla de Procesos.

umap(dir_proceso, segmento, dir_virtual, bytes_requeridos): Calcula la dirección física correspondiente a una dirección virtual de un proceso dado.

panic(mensaje_error, id_error): Provoca el término de la ejeución de MINIX

vir2phys(dir_virtual): Convierte una dirección virtual en su equivalente físico.

phys_copy(fuente, destino, num_bytes): Copia num_bytes de la dirección fuente a la direccción destino.

· Código Fuente.:

```
PUBLIC struct bparam_s boot_parameters;

PRIVATE int do_gboot(m_ptr)
message *m_ptr;                 /* puntero al mensaje SYS_GBOOT */
{
 phys_bytes dst_phys;

 /* Obtiene la dirección física de destino */
 dst_phys = umap(proc_addr(m_ptr->PROC1), D, (vir_bytes) m_ptr->MEM_PTR,
                                 (vir_bytes) sizeof(boot_parameters));
 if (dst_phys == 0)
   panic("bad call to SYS_GBOOT", NO_NUM);

 /* Copia la información */
 phys_copy(vir2phys(&boot_parameters), dst_phys,
           (phys_bytes)sizeof(boot_parameters));
 return(OK);
}
```

DO_MEM

•Tarea Llevada a Cabo.:

Durante el arranque del sistema, el **MM** envía a la tarea del sistema un conjunto de mensajes **SYS_MEM** para obtener las direcciones base y el tamaño de los distintos bloques de memoria disponibles. El procedimiento encargado de manejar estas peticiones es **do_mem**.

· Constantes.:

NR_MEMS: número de bloques de memoria (inicialmente 3).

· Código Fuente.:

```
PRIVATE int do_mem(m_ptr)
register message *m_ptr; /* puntero al mensaje SYS_MEM */
{
 struct memory *memp;

 /* En el mensaje se devuelve la dirección base y el tamaño del siguiente bloque de
  memoria libre */
 for (memp = mem; memp < &mem[NR_MEMS]; ++memp) {
        m_ptr->m1_i1 = memp->base;
        m_ptr->m1_i2 = memp->size;
        m_ptr->m1_i3 = tot_mem_size;
        memp->size = 0;
        if (m_ptr->m1_i2 != 0)
          break;
 }
 return(OK);
}
```

DO_UMAP

•Tarea Llevada a Cabo.:

El mensaje **SYS_UMAP** lo utilizan los procesos que no forman parte del núcleo para solicitar el cálculo de la dirección física de memoria correspondiente a una dirección virtual dada. El procedimiento **do_umap** (perteneciente al kernel) es el encargado de responder a estas peticiones llamando a **umap**, que es la función que utilizan los procesos del kernel para realizar la conversión.

· Procedimientos Utilizados.:

proc_addr(num_proceso): Obtiene el puntero a la entrada de la Tabla de Procesos.

umap(dir_proceso, segmento, dir_virtual, bytes_requeridos): Calcula la dirección física correspondiente a una dirección virtual de un proceso dado.

· Código Fuente.:

```
PRIVATE int do_umap(m_ptr)
register message *m_ptr; /* puntero al mensaje SYS_UMAP */
{
 /* En el propio mensaje se devuelve la dirección física */
 m_ptr->SRC_BUFFER = umap(proc_addr((int) m_ptr->SRC_PROC_NR),
                    (int) m_ptr->SRC_SPACE,                  /* Tipo segmento */
                    (vir_bytes) m_ptr->SRC_BUFFER, /* Dir. virtual */
                    (vir_bytes) m_ptr->COPY_BYTES);          /* Tamaño del bloque */
 return(OK);
}
```

DO_TRACE

•Tarea Llevada a Cabo.:

El mensaje **SYS_TRACE** da soporte a la llamada del sistema **PTRACE**, la cual se utiliza para la depuración de programas. Hay once operaciones diferentes que se pueden hacer con **PTRACE**, de las cuales la mayoría las lleva a cabo el **MM** enviando un mensaje **SYS_TRACE** a la tarea del sistema, quien a su vez llama a **do_trace**. Las operaciones de depuración son, en general, bastante simples. El MINIX emplea el bit **P_STOP** en la tabla de procesos para reconocer si se está en proceso de depuración.

· Constantes.:

P_SLOT_FREE: puesto a uno para indicar que no está en uso.
P_STOP: puesto a uno cuando el proceso esta siendo traceado.

· Procedimientos Utilizados.:

proc_addr(num_proceso): Obtiene el puntero a la entrada de la Tabla de Procesos.

umap(dir_proceso, segmento, dir_virtual, bytes_requeridos): Calcula la dirección física correspondiente a una dirección virtual de un proceso dado.

vir2phys(dir_virtual): Convierte una dirección virtual en su equivalente físico.

phys_copy(fuente, destino, num_bytes): Copia num_bytes de la dirección fuente a la direccción destino.

lock_unready(proceso): Saca al proceso de la cola de procesos preparados para ejecución

lock_ready(proceso): Coloca al proceso en la cola de procesos preparados para ejecución.

· Código Fuente.:

```
#define TR_PROCNR     (m_ptr->m2_i1)
#define TR_REQUEST    (m_ptr->m2_i2)
#define TR_ADDR       ((vir_bytes) m_ptr->m2_l1)
#define TR_DATA       (m_ptr->m2_l2)
#define TR_VLSIZE     ((vir_bytes) sizeof(long))

PRIVATE int do_trace(m_ptr)
register message *m_ptr;          /* puntero al mensaje SYS_TRACE */
{
```

```
/* Maneja los comandos de depuración soportados por la llamada del sistema ptrace
 * Los comandos son:
 * T_STOP       detener el proceso
 * T_OK         permitir que el padre tracee este proceso
 * T_GETINS     devolver un valor desde el espacio de instrucciones
 * T_GETDATA    devolver un valor desde el espacio de datos
 * T_GETUSER    devolver un valor de la tabla de procesos de usuario
 * T_SETINS     establecer un valor del espacio de instrucciones
 * T_SETDATA    establecer un valor del espacio de datos
 * T_SETUSER    establecer un valor en la tabla de procesos de usuario
 * T_RESUME     continuar la ejecución
 * T_EXIT       salir
 * T_STEP       establecer el bit de traza
 *
 * Los comandos T_OK y T_EXIT los maneja completamente el MM.
 */

 register struct proc *rp;
 phys_bytes src, dst;
 int i;

 rp = proc_addr(TR_PROCNR);
 if (rp->p_flags & P_SLOT_FREE)
   return(EIO);
 switch (TR_REQUEST) {

 /* para el proceso */
 case T_STOP:
         if (rp->p_flags == 0) lock_unready(rp);
         rp->p_flags |= P_STOP;
         rp->p_reg.psw &= ~TRACEBIT;    /* borrar el bit de traza */
         return(OK);

 /* devuelve un valor desde el segmento de código */
 case T_GETINS:
         if (rp->p_map[T].mem_len != 0) {
                 if ((src = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
                 phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
                 break;
         }

 /* devuelve un valor desde el segmento de datos */
 case T_GETDATA:
         if ((src = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
         phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
         break;

 /* devuelve un valor de la tabla de procesos */
 case T_GETUSER:
         if ((TR_ADDR & (sizeof(long) - 1)) != 0 ||
            TR_ADDR > sizeof(struct proc) - sizeof(long))
                 return(EIO);
         TR_DATA = *(long *) ((char *) rp + (int) TR_ADDR);
         break;

 /* establece un valor en el segmento de código */
 case T_SETINS:
         if (rp->p_map[T].mem_len != 0) {
```

```
                if ((dst = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
                phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
                TR_DATA = 0;
                break;
        }

  /* establece un valor en el segmento de datos */
  case T_SETDATA:
        if ((dst = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
        phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
        TR_DATA = 0;
        break;

  /* establece un valor en la tabla de procesos */
  case T_SETUSER:
        if ((TR_ADDR & (sizeof(reg_t) - 1)) != 0 ||
            TR_ADDR > sizeof(struct stackframe_s) - sizeof(reg_t))
                return(EIO);
        i = (int) TR_ADDR;
#if (CHIP == INTEL)
        /* Alterar los registros de segmentos puede bloquear el kernel cuando intenta
         * acceder a ellos al reiniciar un proceso por lo que dicha acción no se permite */
        if (i == (int) &((struct proc *) 0)->p_reg.cs ||
            i == (int) &((struct proc *) 0)->p_reg.ds ||
            i == (int) &((struct proc *) 0)->p_reg.es ||
#if _WORD_SIZE == 4
            i == (int) &((struct proc *) 0)->p_reg.gs ||
            i == (int) &((struct proc *) 0)->p_reg.fs ||
#endif
            i == (int) &((struct proc *) 0)->p_reg.ss)
                return(EIO);
#endif
        if (i == (int) &((struct proc *) 0)->p_reg.psw)
                /* sólo ciertos bits son modificables */
                SETPSW(rp, TR_DATA);
        else
                *(reg_t *) ((char *) &rp->p_reg + i) = (reg_t) TR_DATA;
        TR_DATA = 0;
        break;

  /* continúa la ejecución */
  case T_RESUME:
        rp->p_flags &= ~P_STOP;
        if (rp->p_flags == 0) lock_ready(rp);                   /* desbloquea al proceso */
        TR_DATA = 0;
        break;
  /* activa el bit de traza */
  case T_STEP:
        rp->p_reg.psw |= TRACEBIT;
        rp->p_flags &= ~P_STOP;
        if (rp->p_flags == 0) lock_ready(rp);       /* desbloquea al proceso */
        TR_DATA = 0;
        break;
  default:
        return(EIO);
  }
  return(OK);
}
```

CAUSE_SIG

•Tarea Llevada a Cabo.:

Cada vez que una tarea necesita producir una señal llama a **cause_sig**. Este procedimiento modifica un bit en el campo **p_pending** en la entrada de la tabla de procesos del proceso señalado y luego comprueba si el **MM** está suspendido esperando por un mensaje para procesar la siguiente petición. Si es así, se utiliza el procedimiento **inform** para que el **MM** tramite la señal.

· Constantes:

RECEIVING: puesto a uno cuando el proceso está bloqueado.
PENDING: indica que el proceso tiene alguna señal pendiente.
SIG_PENDING: evita que se ejecute un proceso con señales pendientes.

· Procedimientos Utilizados.:

proc_addr(proceso): Obtiene el puntero en la entrada de la Tabla de Procesos.

sigismember(conjunto, señal): Determina si la señal pertenece o no al conjunto de señales pendientes de un proceso.

sigaddset(conjunto, señal): Añade la señal al conjunto de señales pendientes de un proceso.

lock_unready(proceso): Saca un proceso de la cola de procesos preparados para ejecución.

inform(): Informa al MM de las señales pendientes.

· <u>Código Fuente</u>.:

```
PUBLIC void cause_sig(proc_nr, sig_nr)
int proc_nr;                                    /* proceso a señalar */
int sig_nr;                                     /* señal a enviar */
{
 register struct proc *rp, *mmp;

 rp = proc_addr(proc_nr);

 /* Comprueba si el proceso ya tiene pendiente una señal de ese tipo */
 if (sigismember(&rp->p_pending, sig_nr))
        return;

 /* Añade la señal al conjunto de señales pendientes e incrementa el contador */
 sigaddset(&rp->p_pending, sig_nr);
 ++rp->p_pendcount;
```

```
  /* Si el proceso ya tenia una señal pendiente, simplemente retorna */
  if (rp->p_flags & PENDING)
            return;

  /* Si el proceso es ejecutable se pasa a la lista de procesos bloqueados */
  if (rp->p_flags == 0)
        lock_unready(rp);

  /* Marca el proceso como que tiene una señal pendiente y que no se debe ejecutar */
  rp->p_flags |= PENDING | SIG_PENDING;

  /* Incrementa el número de procesos con señales pendientes */
  ++sig_procs;

  /* Si el MM está ocupadoy no puede recibir mensajes simplemente retorna.
   * Si no, se le informa de que hay una señal pendiente */
  mmp = proc_addr(MM_PROC_NR);
  if ( ((mmp->p_flags & RECEIVING) == 0) || mmp->p_getfrom != ANY)
        return;
  inform();
}
```

INFORM

•Tarea Llevada a Cabo.:

Este procedimiento sólo se llama tras comprobar que el **MM** está ocioso. El procedimiento **inform** construye un mensaje de tipo **KSIG** y lo envía al **MM**. La tarea o el proceso que había invocado a **cause_sig** continua su ejecución tan pronto como el mensaje haya sido copiado en el buffer de entrada del **MM**, no se espera hasta que el **MM** se ejecute como ocurriría si se hubiera empleado el mecanismo normal de envío de mensajes (que provoca el bloqueo del emisor).

· Constantes:

END_PROC_ADDR: dirección del último proceso de la tabla de procesos.
BEG_SERV_ADDR: dirección del primer proceso.

· Procedimientos Utilizados.:

proc_number(dir_proceso): Obtiene el nº de proceso a partir del puntero a la Tabla de Procesos.

proc_addr(num_proceso): Obtiene el puntero a la Tabla de Procesos

lock_mini_send(proc_src,proc_dst,msg): Para enviar un mensaje entre procesos.

lock_pick_proc: Planifica la ejecución del **MM**.

panic(string,número): Termina la ejecución de Minix debido a un error fatal.

· Código Fuente.:

```
PUBLIC void inform()
{
 register struct proc *rp;

 /* Se recorre la tabla de procesos en busca de alguno con señales pendientes. */
 for (rp = BEG_SERV_ADDR; rp < END_PROC_ADDR; rp++)
        if (rp->p_flags & PENDING) {
            /* Prepara el mensaje */
                m.m_type = KSIG;
                m.SIG_PROC = proc_number(rp);
                m.SIG_MAP = rp->p_pending;

            /* Decrementa el número de procesos con señales pendientes */
              sig_procs--;
```

```
                /* Envía la señal */
                  if (lock_mini_send(proc_addr(HARDWARE), MM_PROC_NR, &m)
                   != OK)
                      panic("can't inform MM", NO_NUM);

                  /* Vacía el conjunto de señales pendientes del proceso */
                  sigemptyset(&rp->p_pending);


                  /* Borra el bit de señales pendientes */
                  rp->p_flags &= ~PENDING;

                 /* Planifica al MM para que se pueda ejecutar */
                      lock_pick_proc();
                      return;
          }
}
```

UMAP

•Tarea Llevada a Cabo.:

El procedimiento **umap** se encarga de mapear direcciones virtuales en direcciones físicas. Sus parámetros son un puntero a la entrada en la tabla de procesos del proceso o tarea cuyo espacio virtual de direcciones se va a mapear, un indicador del tipo de segmento, la dirección virtual a mapear y un tamaño en bytes. Este último parámetro es útil ya que **umap** comprueba que el bloque completo esté dentro del espacio de direcciones del proceso.

· Código Fuente.:

```
PUBLIC phys_bytes umap(rp, seg, vir_addr, bytes)
register struct proc *rp;   /* Puntero a la tabla de procesos */
int seg;                            /* Segmento T, D, o S */
vir_bytes vir_addr;                 /* Dir. virtual (en bytes) dentro del segmento */
vir_bytes bytes;            /* Tamaño del bloque */
{
  vir_clicks vc;
  phys_bytes pa;
#if (CHIP == INTEL)
  phys_bytes seg_base;
#endif

  /* Si el tamaño del bloque es cero, devuelve un 0 */
  if (bytes <= 0) return( (phys_bytes) 0);

  /* Calcular la dirección virtual del final del bloque */
  vc = (vir_addr + bytes - 1) >> CLICK_SHIFT;       /* último click de datos */

#if (CHIP == INTEL) || (CHIP == M68000)
  /* Si el segmento no es tipo T entonces... */
  if (seg != T)
            /* ...si vc es menor que el límite del segmento D del proceso, entonces el
           segmento es D, si no, el segmento es S */
            seg = (vc < rp->p_map[D].mem_vir + rp->p_map[D].mem_len ? D : S);
#else
  /* Si el segmento no es tipo T entonces... */
  if (seg != T)
            /* ...si vc es menor que la base del segmento S del proceso, entonces el
           segmento es D, si no, el segmento es S */
            seg = (vc < rp->p_map[S].mem_vir ? D : S);
#endif

  /* Si la dirección virtual es mayor que el límite del segmento del proceso entonces
   devolver 0 */
  if((vir_addr>>CLICK_SHIFT) >=
     rp->p_map[seg].mem_vir+rp->p_map[seg].mem_len)
            return( (phys_bytes) 0 );
```

```
  /* Obtener de la tabla de procesos la dirección base física del segmento en bytes */
#if (CHIP == INTEL)
  seg_base = (phys_bytes) rp->p_map[seg].mem_phys;
  seg_base = seg_base << CLICK_SHIFT;   /* Origen del segmento en bytes */
#endif
  pa = (phys_bytes) vir_addr;
#if (CHIP != M68000)
  pa -= rp->p_map[seg].mem_vir << CLICK_SHIFT;

/* Devolver la dirección física: dirección base del segmento + offset */
  return(seg_base + pa);
#endif

#if (CHIP == M68000)
#if (SHADOWING == 0)
  pa -= (phys_bytes)rp->p_map[seg].mem_vir << CLICK_SHIFT;
  pa += (phys_bytes)rp->p_map[seg].mem_phys << CLICK_SHIFT;
#else
  if (rp->p_shadow && seg != T) {
            pa -= (phys_bytes)rp->p_map[D].mem_phys << CLICK_SHIFT;
            pa += (phys_bytes)rp->p_shadow << CLICK_SHIFT;
  }
#endif

  /* Devolver la dirección física: dirección base del segmento + offset */
  return(pa);
#endif
}
```

NUMAP

•Tarea Llevada a Cabo.:

Para los driver de dispositivos es conveniente poder acceder a los servicios de **umap** a partir del número de proceso en lugar del puntero a la tabla de procesos. La función **numap** es la encargada de hacerlo llamando a **proc_addr** para obtener la dirección de la entrada en la tabla de procesos y llamar a **umap**. Además, en este caso el segmento siempre es **D** por lo que no se especifica en la llamada.

· Procedimientos Utilizados.:

proc_addr(num_proceso): Obtiene el puntero a la entrada de la Tabla de Procesos.

umap(dir_proceso, segmento, dir_virtual, bytes_requeridos): Calcula la dirección física correspondiente a una dirección virtual de un proceso dado.

· Código Fuente.:

```
PUBLIC phys_bytes numap(proc_nr, vir_addr, bytes)
int proc_nr;                    /* Nº del proceso */
vir_bytes vir_addr;             /* Dir. virtual (en bytes) dentro del segmento D */
vir_bytes bytes;          /* Nº de bytes requeridos */
{
  return(umap(proc_addr(proc_nr), D, vir_addr, bytes));
}
```

ALLOC_SEGMENTS

•Tarea Llevada a Cabo.:

Esta función toma las asignaciones de segmentos guardadas en una entrada de la tabla de procesos y manipula los registros y los descriptores que utilizan los procesadores PENTIUM para dar soporte a la protección de segmentos a nivel hardware. Esta función es llamada por **do_newmap** y por la rutina **main** del kernel durante la inicialización.

· Constantes:

INTR_PRIVILEGE, TASK_PRIVILEGE, USER_PRIVILEGE: los posibles niveles de privilegio de los procesos.

· Procedimientos Utilizados.:

istaskp(rp): Devuelve la prioridad de un proceso (Tarea o Usuario).

init_codeseg: Inicializa el segmento de código.

init_dataseg: Inicializa el segmento de datos.

click_to_hclick

· Código Fuente.:

```
PUBLIC void alloc_segments(rp)
register struct proc *rp;
{
  phys_bytes code_bytes;
  phys_bytes data_bytes;
  int privilege;

  /* Si estamos en modo protegido... */
  if (protected_mode) {
        /* Se calcula el límite de memoria virtual del segmento S del proceso */
        data_bytes = (phys_bytes) (rp->p_map[S].mem_vir + rp->p_map[S].mem_len)
                << CLICK_SHIFT;

        /* Si la longitud del segmento T es cero... */
        if (rp->p_map[T].mem_len == 0)
                /* ... el límite del segmento T es igual al de S */
                code_bytes = data_bytes; /* common I&D, poor protect */
        else
                /* si no, se obtiene el límite del segmento T */
                code_bytes = (phys_bytes) rp->p_map[T].mem_len << CLICK_SHIFT;
```

```
        /* Calcular la prioridad del proceso: tarea o proceso de usuario */
        privilege = istaskp(rp) ? TASK_PRIVILEGE : USER_PRIVILEGE;

        /* Inicializar el segmento de código */
        init_codeseg(&rp->p_ldt[CS_LDT_INDEX],
                (phys_bytes) rp->p_map[T].mem_phys << CLICK_SHIFT,
                code_bytes, privilege);

        /* Inicializar el segmento de datos */
        init_dataseg(&rp->p_ldt[DS_LDT_INDEX],
                (phys_bytes) rp->p_map[D].mem_phys << CLICK_SHIFT,
                data_bytes, privilege);

        /* Calcular los valores de los registros cs, gs, fs, ss, es y ds */
        rp->p_reg.cs = (CS_LDT_INDEX * DESC_SIZE) | TI | privilege;

#if _WORD_SIZE == 4
        rp->p_reg.gs =
        rp->p_reg.fs =
#endif
        rp->p_reg.ss =
        rp->p_reg.es =
        rp->p_reg.ds = (DS_LDT_INDEX*DESC_SIZE) | TI | privilege;
 }
 /* Si no estamos en modo protegido... */
 else {
        /* Calcular los valores de los registros cs, ss, es y ds */
        rp->p_reg.cs = click_to_hclick(rp->p_map[T].mem_phys);
        rp->p_reg.ss =
        rp->p_reg.es =
        rp->p_reg.ds = click_to_hclick(rp->p_map[D].mem_phys);
 }
}
```

CUESTIONES

1. Justificación de la necesidad de la tarea del sistema.

2. Diferencias y similitudes entre los mensajes SYS_COPY y SYS_VCOPY. ¿Es realmente necesaria la existencia del mensaje SYS_VCOPY?

3. Tipos de mensajes que acepta la tarea del sistema.

4. ¿ Por qué es necesario el uso de señales en la comunicación entre el SF, MM con el Kernel.?. ¿ Qué son los DO_XXX.?.

5. ¿ Por qué un proceso tiene las misma entrada *k* correspondiente a él en las tres tablas.?

ALGORÍTMOS.

(A “GROSSO MODO” .).

ALGORITMO DO_FORK.:

1. SI proceso padre o hijo no son procesos de usuario
ENTONCES Error

2. Obtener las direcciones de los procesos padre e hijo

3. Copiar en la TP la estructura del padre en el hijo

4. Actualizar n_ de proceso del hijo

5. Colocar los Flags del hijo en la TP:

- No ejecutable por no estar mapeado (sin mapa de memoria)
- No tiene señales pendientes ni está en modo traza

6. Poner a 0 el Mapa y Contador de señales pendientes del hijo

7. Colocar el PID del hijo, y para el hijo su PID será 0

8. Resetea los tiempos de sistema y de usuario del proceso hijo

ALGORITMO DO_NEWMAP.

1. Extraer los del mensaje la direccion del nuevo mapa y los números del proceso a mapear y el solicitante (MM)

2. SI el proceso a mapear no lo es ENTONCES Error

3. Obtener los punteros en la TP para el MM y el proceso a mapear

4. Calcular el tamaño del mapa de memoria (T+D+S) a cargar del MM

5. Obtener la dirección virtual del nuevo mapa de memoria en el MM
Obtener de la TP la dirección virtual del mapa de memoria actual del proceso a mapear

6. Calcular las direcciones físicas correspondientes al área de datos(segmento D) del nuevo mapa en el MM y del actual

7. Copiar el nuevo mapa de memoria desde el MM al proceso

8. Actualiza los segmentos del proceso mapeado y pone a cero flag que indicaba proceso no mapeado

SI todos los flags igual a 0
ENTONCES colocarlo en la cola Ready

ALGORITMO DO_EXEC.

1. Extraer del mensaje el nuevo Puntero de pila.

2. SI proceso solicitante no es un proceso
ENTONCES Error

3. SI proceso esta en modo traza
ENTONCES Enviarle señal SIGTRAP

4. Obtener la direccion del proceso en la Tabla de procesos

5. Actualizar el valor de su SP en la TP
Resetear el valor de su PC y del Cronómetro de Alarma en la TP

6. Desbloquear el Proceso
SI todos sus flags son 0
ENTONCES Colocarlo en la Cola Ready

7. Incluye al proceso solicitante en la Tabla de Procesos Activos del sistema (se visualiza con F1)

ALGORITMO DO_XIT.

1. Obtener del mensaje los n_ del proceso padre y del que termina

2. SI no son procesos
ENTONCES Error

3. Obtener sus direcciones en la TP

4. Bloquear interrupciones de la CPU y añade en el padre los Tiempos de usuario y sistema del proceso que termina
Resetear el Cronómetro de la Alarma

5. SI proceso hijo es ejecutable
ENTONCES Extraerlo de la Cola Ready

6. Eliminarlo de la Tabla de procesos activos

7. SI proceso que sale bloqueado por envio de mensaje ENTONCES

PARA Todos los procesos en TP HACER

SI lista de procesos que envian al proceso actual vacia
ENTONCES Continuar

SI el proceso que sale envia al actual
ENTONCES Extraer proceso que sale de su Cola

FIN PARA

FIN SI

8. SI el proceso que sale tiene señales pendientes
ENTONCES Decrementar el contador de procesos con señales pendientes

Pone a 0 el Mapa de bits y el Contador de señales pendientes

9. Marca como libre su entrada en la Tabla de Procesos