Señales en MINIX

ALUMNOS:

CARLOS DÍAZ SUÁREZ
ROBERTO JORGE ALEJANDRO

INDICE

1 Señales

Las señales son un mecanismo para transferir información entre procesos de forma asíncrona. Son para el software lo que las interrupciones para el hardware.

Se dispone de un conjunto finito de señales:

Señal	#	Notas	Descripción
SIGHUP	1	k	
SIGINT	2	k	Interrupción (usualmente DEL o CRTL – C)
SIGQUIT	3	kc	Salir (usualmente CRTL – \)
SIGILL	4	kc	Instrucción ilegal
SIGTRAP	5	xkc	Trampa para traza
SIGABRT	6	kc	Abortar el proceso
SIGUNUSED	7	---	No se utiliza
SIGFPE	8	k	Excepción de punto flotante
SIGKILL	9	k	Matar al proceso
SIGUSR1	10	k	Señal definida por el usuario #1
SIGSEGV	11	kc	Violación de segmento
SIGUSR2	12	k	Señal definida por el usuario #2
SIGPIPE	13	k	Escritura en una tubería rota
SIGALRM	14	k	Alarma de reloj
SIGTERM	15	k	Terminación

Notas:

- k: Se mata al proceso si la señal no es interceptada.
- c: La señal causa un volcado (core).
- x: Se trata de una extensión de MINIX, no definida por POSIX.

Estás se encuentran definidas en signal.h, donde podemos encontrar algunas otras señales más de uso menos frecuente. El número de señales que MINIX soporta se define tambien en dicho fichero:

```
#define _NSIG             16     /* number of signals used */
```

Cada señal tiene un conjunto de atributos que indican como se debe actuar cuando un proceso recibe dicha señal, así pues, el Manejador de Memoria (MM) mantiene, en cada entrada de su tabla de procesos, los atributos de cada señal para un proceso determinado. El siguiente fragmento de código extraído de mproc.h muestra la estructura de una entrada en la tabla de procesos del MM, donde se han resaltado aquellos en los que se guardan tales atributos. Como veremos más adelante, se utilizarán muchos de los campos de esta estructura.

```
EXTERN struct mproc {
  struct mem_map mp_seg[NR_SEGS];/* apunta al código, datos y pila */
  char mp_exitstatus;            /* guarda el estado de un proceso al salir */
  char mp_sigstatus;             /* guarda el estado tras matar un proceso */
  pid_t mp_pid;                  /* id del proceso */
  pid_t mp_procgrp;              /* pid del grupo de procesos */
  pid_t mp_wpid;                 /* pid del proceso esperado */
  int mp_parent;                 /* indice al proceso padre */

  /* uids y gids reales y efectivos. */
  uid_t mp_realuid;              /* uid real del proceso */
  uid_t mp_effuid;               /* uid efectivo del proceso */
  gid_t mp_realgid;              /* gid real del proceso */
  gid_t mp_effgid;               /* gid efectivo del proceso */

  /* Identificación de fichero para compartir. */
  ino_t mp_ino;                  /* inode del fichero */
  dev_t mp_dev;                  /* device number of file system */
  time_t mp_ctime;               /* datos modificación del fichero */

  /* Información para el manejo de señales */
  sigset_t mp_ignore;            /* 1 significa ignorar la señal, 0 no */
  sigset_t mp_catch;             /* 1 significa interceptar la señal, 0 no */
  sigset_t mp_sigmask;           /* señales que han de bloquearse */
  sigset_t mp_sigmask2;          /* copia de mp_sigmask */
  sigset_t mp_sigpending;        /* señales bloqueadas */
  struct sigaction mp_sigact[_NSIG + 1]; /* Atributos de cada señal */
  vir_bytes mp_sigreturn;        /* dirección de la función __sigreturn */

  /* Compatibilidad */
  sighandler_t mp_func;          /* 1 manejador usario para todas las señales */

  unsigned mp_flags;             /* flags */
  vir_bytes mp_procargs;         /* puntero a argumentos del proceso en pila */
} mproc[NR_PROCS];
```

Los tipos de dichos campos se definen en signal.h:

```
typedef unsigned long sigset_t;

struct sigaction {
  __sighandler_t sa_handler;     /* SIG_DFL, SIG_IGN, o un puntero a función */
  sigset_t sa_mask;              /* señales a bloquear durante el manejo */
  int sa_flags;                  /* flags (véase signal.h) */
};
```

El conjunto de señales se mapea sobre los campos de tipo sigset_t, es decir, cada bit de dicho campo se corresponde con una de las señales. Así pues, un bit a 1 en los campos mp_ignore, mp_catch o mp_sigmask, indica que ha de ignorarse, interceptarse o bloquearse dicha señal respectivamente, y un 0 lo contrario. Gracias a los campos de tipo sigset_t el MM sabe como ha de manejar una señal destinada a un proceso, de esta forma puede que dicha señal sea ignorada o deba ser interceptada.

Cuando una señal ha de ser interceptada el MM acude al vector mp_sigact. Los elementos de dicho vector son de tipo sigaction. Una estructura sigaction contiene un puntero al manejador de la señal, que puede ser una función especificada por el usuario, o un indicador de que ha de usarse el manejador por defecto (SIG_DFL), o de que ha de ignorarse la señal (SIG_IGN). La estructura también contiene un campo llamado sa_mask que indica que señales se han de bloquear durante el tratamiento de la señal que nos ocupa, y por último contiene una serie de flags que son interpretados por el MM.

2 Origen de una señal

Las señales pueden provenir del teclado, del reloj o de otro proceso. A continuación veremos cada uno de estos casos por separado y como en todos ellos es enviado un mensaje al MM para informale de que ha tenido lugar una señal.

2.1 Señales desde el teclado

Desde teclado se pueden originar tres señales: SIGINT, SIGQUIT y SIGABRT.

Cuando se teclea un carácter, el manejador de la interrupción de teclado (kbd_hw_int) almacena dicho carácter en el buffer de teclado, activa un flag de la consola asociada al teclado para indicar a la tarea de terminal (tty_task) que dicha consola ha experimentado un evento y prepara un timer para que finalice en el próximo tick de reloj.

Al expirar dicho timer, la tarea de reloj (clock_task) envía un mensaje a tty_task indicandole que algo ha pasado. Cuando tty_task recibe el mensaje, chequea el flag de evento de todas las terminales y llama a handle_events para cada dispositivo que tenga el flag activado. En el caso de la consola asociada al teclado, handle_events llamará a kb_read, función del controlador de teclado que se encarga de atender la petición de lectura de los caracteres pendientes en el buffer de teclado.

La función kb_read llamará a la función in_process, en tty.c, que se ocupa de procesar el buffer de caracteres que se le pasa como argumento. Si al procesar dicho buffer se detectan los caracteres ^? (SIGINT) ó ^\ (SIGQUIT), se invoca a la función sigchar, tb. en tty.c. Los argumentos de sigchar son el puntero a la terminal en la que ocurrió el evento y la señal a tratar, SIGINT ó SIGQUIT.

La función sigchar invoca a su vez a la función cause_sig, en system.c, pasándole como argumentos la entrada en la tabla de procesos del último proceso que ha abierto la terminal, y la señal detectada. La función cause_sig se ocupa de informar al MM mediante la función inform, tb. en system.c, de que ha tenido lugar una señal en el kernel, para ello crea un mensaje tipo KSIG que envía al MM gracias a la función lock_mini_send y por último llama a lock_pick_prock para que el Manejador de Memoria tenga oportunidad de pasar a ejecución y recibir el mensaje.

De esta forma, quizás un poco extensa debido al número de funciones que intervienen, llega un mensaje de tipo KSIG (Kernel SIGnal) al MM desde el teclado.

Sin embargo, aún falta la señal SIGABRT que es detectada por kb_read si encuentra la secuencia CTRL+ALT+DEL en el buffer de caracteres, antes de pasar dicho buffer a in_process. Entonces, kb_read invoca directamente a cause_sig con los parámetros pertinentes para que ésta informe al MM.

2.2 Señales desde otro proceso

Un proceso puede enviar una señal a otro proceso por medio de llamadas al sistema. Dichas llamadas serán estudiadas más adelante, y por ahora es suficiente con saber que todas ellas crean un mensaje que envían al MM gracias a la función _syscall, en syscall.c. No está de más recordar que una llamada al sistema es una simple función como otra cualquiera.

La función _syscall recibe como parámetros, el destino del mensaje que siempre es el MM, un número que identifica el tipo de llamada y el mensaje creado por la propia llamada. _syscall se limita a copiar el tipo de llamada en el tipo de mensaje y a enviar dicho mensaje al MM por medio de la función _sendrec.

2.3 Señales desde el reloj

Si un proceso ha solicitado a la tarea de reloj (clock_task) una alarma o una pausa por medio de la llamada al sistema correspondiente, cuando dicha alarma o pausa expira, clock_task invoca a cause_sig pasándole como argumentos el proceso que solicitó la alarma o pausa y la señal SIGALARM.

Como último punto dentro de este apartado, es importante observar como se ha realizado la comunicación entre las distintas capas del s.o. MINIX. Los procesos de usuario en la capa 4 se comunican con el MM en la capa 3 gracias a las llamadas al sistema que hacen uso del paso de mensajes, mediante la función _syscall, para implementar la comunicación.

A sú vez, las distintas tareas (tty_task y clock_task) en la capa 2 se comunican con el MM en la capa 3 mediante la función cause_sig, también mediante paso de mensajes.

3 Como actúa el MM

Los mensajes que recibe el MM, en cuanto al manejo de señales se refiere, se pueden agrupar en dos categorías, aquellos que son enviados por una llamada al sistema, para los que el tipo del mensaje depende de la llamada (ALARM, PAUSE, SIGPROCMASK, etc.), y aquellos que son enviados por las tareas cuando detectan una señal y cuyo tipo es KSIG.

Si el tipo de mensaje recibido corresponde a una llamada al sistema, entonces el MM invoca a la función que da soporte a dicha llamada. Dichas funciones de soporte se recogen en signal.c y son por tanto el objeto de estudio de este trabajo.

Cuando un mensaje de tipo KSIG es recibido por el MM, éste determina a que procesos se les debería enviar dicha señal en función del tipo de la misma y de si el proceso desea interceptar dicha señal. Así pues, para cada uno de esos procesos, el MM lee los atributos que éste define para dicha señal y actúa en consecuencia, es decir, la ignora, invoca al manejador por defecto o llama a la función especificada por el usuario, tal y como se comentó anteriormente (apartado 1).

En el caso de que se intercepte la señal, es necesario guardar la información acerca del estado del proceso con el fin de poder reanudar la ejecución normal del mismo. El MM guarda esta información en la pila del proceso, llamando a la tarea de sistema, la cual modifica el contador de programa del proceso para que continúe su ejecución en el manejador de la señal. Cuando el manejador de la señal ha terminado, se produce una llamada al sistema SIGRETURN para que, entre la tarea del sistema y el MM recuperen el estado del proceso antes de la señal.

4 Casos especiales

- Un proceso zombie no puede recibir señales.

- Un proceso que explicitamente ha utilizado una llamada al sistema, para ignorar una señal, o para bloquearla, no puede recibir la señal en cuestión.

- Si se va a enviar una señal a un proceso que no tiene espacio suficiente en su pila, dicho proceso será abortado.

- La señal SIGKILL no puede ser ignorada ni bloqueada por un proceso.

5 Llamadas al Sistema

A continuación pasamos a comentar las distintas llamadas al sistema que MINIX soporta.

5.1 sigprocmask (sigprocmask.c)

Esta llamada se encarga de examinar o manipular la máscara de señal. Dicha mácara es el conjunto de señales que están actualmente bloqueadas.

```
PUBLIC int sigprocmask(how, set, oset)
int how;
_CONST sigset_t *set;
sigset_t *oset;
{
  message m;

  if (set == (sigset_t *) NULL) {
      m.m2_i1 = SIG_INQUIRE;
      m.m2_l1 = 0;
  } else {
      m.m2_i1 = how;
      m.m2_l1 = (long) *set;
  }
  if (_syscall(MM, SIGPROCMASK, &m) < 0) return(-1);
  if (oset != (sigset_t *) NULL) *oset = (sigset_t) (m.m2_l1);
  return(m.m_type);
}
```

5.2 kill (kill.c)

Esta llamada envía la señal sig al proceso cuyo pid es proc. La señal debe ser una de las especificadas en el apartado 1, ó 0.

```
PUBLIC int kill(proc, sig)
int proc;         /* proceso a enviar a la señal */
int sig;          /* número de señal */
{
  message m;
  m.m1_i1 = proc;
  m.m1_i2 = sig;
  return(_syscall(MM, KILL, &m));
}
```

5.3 alarm (alarm.c)

Esta llamada provoca la señal SIGALRM que debe ser enviada al proceso que la invoca en un número determinado de segundos.

```
PUBLIC unsigned int alarm(sec)
unsigned int sec;
{
  message m;

  m.m1_i1 = (int) sec;
  return( (unsigned) _syscall(MM, ALARM, &m));
}
```

5.4 pause (pause.c)

Produce una pausa hasta que se recibe una señal.

```
PUBLIC int pause()
{
  message m;

  return(_syscall(MM, PAUSE, &m));
}
```

5.5 sigsuspend (sigsuspend.c)

Activa la máscara de señal indicada por set y suspende el proceso señalado. La señal es tratada, y se restaura el valor de la máscara a su valor anterior a la llamada sigsuspend cuando se retorna.

```
PUBLIC int sigsuspend(set)
_CONST sigset_t *set;
{
  message m;

  m.m2_l1 = (long) *set;
  return(_syscall(MM, SIGSUSPEND, &m));
}
```

5.6 sigpending (sigpending.c)

Esta llamada al sistema devuelve el conjunto de señales que están esperando a ser liberadas. Estas están actualmente bloqueadas por la máscara de señal.

```
PUBLIC int sigpending(set)
sigset_t *set;
{
  message m;

  if (_syscall(MM, SIGPENDING, &m) < 0) return(-1);
  *set = (sigset_t) m.m2_l1;
  return(m.m_type);
}
```

5.7 reboot (reboot.c)

Se usa para parar el sistema. Permite diferentes formas de apagado, que dependerá de how.

```
int reboot(int how, ...)
{
  message m;
  va_list ap;

  va_start(ap, how);
  if ((m.m1_i1 = how) == RBT_MONITOR) {
      m.m1_p1 = va_arg(ap, char *);
      m.m1_i2 = va_arg(ap, size_t);
  }
  va_end(ap);

  return _syscall(MM, REBOOT, &m);
}
```

5.8 sigreturn (sigreturn.c)

Invoca a la tarea del sistema para que recupere el contexto del proceso. Esta llamada tiene lugar tras el manejo de una señal.

```
PUBLIC int sigreturn(scp)
register struct sigcontext *scp;
{
  sigset_t set;

  static message m;

  sigfillset(&set);
  sigprocmask(SIG_SETMASK, &set, (sigset_t *) NULL);

  m.m2_l1 = scp->sc_mask;
  m.m2_i2 = scp->sc_flags;
  m.m2_p1 = (char *) scp;
  return(_syscall(MM, SIGRETURN, &m));
}
```

5.9 sigset

El conjunto de llamadas sigset es utilizado para manipular un conjunto de señales. Estas llamadas son:

Sigaddset, se utiliza para añadir una señal a un conjunto.

```
int sigaddset(sigset_t *set, int sig)
```

Sigdelset, borra una señal del conjunto.

```
int sigdelset(sigset_t *set, int sig)
```

Sigemptyset, inicializa a vacío el conjunto de señales.

```
int sigemptyset(sigset_t *set)
```

Sigfillset, inicializa el conjunto de señales a lleno, por ejemplo, todas las señales están en el conjunto.

```
int sigfillset(sigset_t *set)
```

Sigismember, devuelve un 1 si la señal se encuentra en el conjunto indicado, en caso contrario devuelve un 0.

```
int sigismember(const sigset_t *set, int sig)
```

5.10 sigaction (estandar POSIX)

Esta llamada al sistema es usada para examinar, modificar o rellenar los atributos de una señal. El argumento sig es la señal con la que se va a trabajar, mientras que el puntero act apunta a una estructura que contiene los nuevos atributos de la señal.

```
int sigaction(sig, act, oact)
int sig;
  _CONST struct sigaction *act;
  struct sigaction *oact;
  {
    message m;

    m.m1_i2 = sig;

    m.m1_p1 = (char *) act;
    m.m1_p2 = (char *) oact;
    m.m1_p3 = (char *) __sigreturn;

    return(_syscall(MM, SIGACTION, &m));
  }
```

5.11 signal (C estandar)

Al igual que sigaction se encarga de examinar, modificar o rellenar los atributos de una señal. Esta función existe por compatibilidad con ANSI C.

```
PUBLIC sighandler_t signal(sig, disp)
int sig;                /* signal number */
sighandler_t disp;      /* signal handler, or SIG_DFL, or SIG_IGN */
{
  struct sigaction sa, osa;

  if (sig <= 0 || sig > _NSIG || sig == SIGKILL) {
      errno = EINVAL;
      return(SIG_ERR);
  }
  sigemptyset(&sa.sa_mask);

#ifdef WANT_UNRELIABLE_SIGNALS
  sa.sa_flags = SA_NODEFER;

  if (sig != SIGILL && sig != SIGTRAP) sa.sa_flags |= SA_RESETHAND;
#else
  sa.sa_flags = 0;
#endif
```

```
  sa.sa_handler = disp;
  if (sigaction(sig, &sa, &osa) < 0) return(SIG_ERR);
  return(osa.sa_handler);
}
```

5.12 _syscall (syscall.c)

Esta función es invocada por todas las llamadas al sistema para enviar el mensaje propio de cada una al MM. Se puede ver como _syscall indica cual es el tipo de mensaje en función del número que identifica a la llamada al sistema syscallnr. _syscall envía el mensaje al MM gracias a la función _sendrec.

```
PUBLIC int _syscall(who, syscallnr, msgptr)
int who;
int syscallnr;
register message *msgptr;
{
  int status;

  msgptr->m_type = syscallnr;
  status = _sendrec(who, msgptr);
  if (status != 0) {
      /* 'sendrec' itself failed. */
      /* XXX - strerror doesn't know all the codes */
      msgptr->m_type = status;
  }
  if (msgptr->m_type < 0) {
      errno = -msgptr->m_type;
      return(-1);
  }
  return(msgptr->m_type);
}
```

6 Variables y defines a tener en cuenta

int dont_reply: indica si se debe o no responder al proceso.

struct mproc *mp: apunta a la ranura *mproc* del proceso actual.

int who: número de proceso del invocador.

message mm_in: mensaje de entrada al MM.

message mm_out: mensaje de salida del MM.

sig_nr: número de señal del mensaje mm_in.

sig_nsa: conjunto nuevo de atributos del mensaje mm_in.

sig_osa: conjunto antiguo de atributos del mensaje mm_in.

y otras como: sig_set, sig_context, sig_flags, ret_mask.

7 Código de signal

Es la parte del manejador de memoria que se encarga de atender las llamadas al sistema realacionadas con las señales.

7.1 do_sigaction

La función do_sigaction lleva a cabo la llamada al sistema SIGACTION.

Lo primero que hace es comprobar si se pretende modificar la respuesta ante la señal SIGKILL, en cuyo caso retornará inmediatamente ya que no es posible ni ignorar ni interceptar dicha señal. En caso contrario se verifica que el número de señal sea correcto.

La variable mp apunta a la entrada del proceso que realizó la llamada al sistema, en dicha entrada existe un vector llamado mp_sigact con un elemento de tipo sigaction por cada señal, donde se guardan los atributos de la misma.

Se realiza una copia de dichos atributos sobre la variable sig_osa, guardando así los atributos que la señal tenía hasta ese momento. Si existen nuevos atributos especificados en la variable sig_nsa, estos se asignan al elemento correspondiente de mp_sigact, convirtiéndose estos en los nuevos atributos de la señal.

Según el manejador que se especifique en los nuevos atributos (SIG_IGN, SIG_DFL, o un puntero a una función) se asignan valores a los campos mp_ignore, mp_catch y mp_sippending en consecuencia.

```
PUBLIC int do_sigaction()
{
  int r;
  struct sigaction svec;
  struct sigaction *svp;

  /* No se permite alterar la respuesta ante la señal SIGKILL */
  if (sig_nr == SIGKILL) return(OK);
  /* Número de señal fuera de rango? */
  if (sig_nr < 1 || sig_nr > _NSIG) return (EINVAL);
  /* Guarda copia de los atributos actuales de la señal */
  svp = &mp->mp_sigact[sig_nr];
  if ((struct sigaction *) sig_osa != (struct sigaction *) NULL) {
      r = sys_copy(MM_PROC_NR,D, (phys_bytes) svp, who, D,
                   (phys_bytes)sig_osa, (phys_bytes)sizeof(svec));
      if (r != OK) return(r);
  }
  /* nuevos atributos == NULL? No hay nada que hacer -> retornar */
  if ((struct sigaction *) sig_nsa == (struct sigaction *) NULL)
    return(OK);

  /* Actualiza los atributos para la señal */
```

```
  r = sys_copy(who, D, (phys_bytes) sig_nsa,MM_PROC_NR, D,
              (phys_bytes)&svec, (phys_bytes)sizeof(svec));

  if (r != OK) return(r);

  if (svec.sa_handler == SIG_IGN) {
      sigaddset(&mp->mp_ignore, sig_nr);
      sigdelset(&mp->mp_sigpending, sig_nr);
      sigdelset(&mp->mp_catch, sig_nr);
  } else {
      sigdelset(&mp->mp_ignore, sig_nr);
      if (svec.sa_handler == SIG_DFL)
            sigdelset(&mp->mp_catch, sig_nr);
      else
            sigaddset(&mp->mp_catch, sig_nr);
  }
  mp->mp_sigact[sig_nr].sa_handler = svec.sa_handler;
  sigdelset(&svec.sa_mask, SIGKILL);
  mp->mp_sigact[sig_nr].sa_mask = svec.sa_mask;
  mp->mp_sigact[sig_nr].sa_flags = svec.sa_flags;
  mp->mp_sigreturn = (vir_bytes) sig_ret;
  return(OK);
}
```

7.2 do_sigpending

Encargada de dar soporte a la llamada al sistema SIGPENDING. Devuelve la máscara de señales pendientes del proceso, de esta forma un proceso puede determinar si tiene señales pendientes de ser tratadas.

```
PUBLIC int do_sigpending()
{
  ret_mask = (long) mp->mp_sigpending;
  return OK;
}
```

7.3 do_sigprocmask

Encargada de dar soporte a la llamada al sistema SIGPROCMASK. Se utiliza tanto para obtener el conjunto de señales que están bloqueadas como para cambiar el estado de una señal (o de todas). Cuando se desbloquea una señal se comprueba si hay alguna pendiente del mismo tipo.

```
PUBLIC int do_sigprocmask()
{
  int i;
  /* Recupera la máscara actual de señales */
  ret_mask = (long) mp->mp_sigmask;
  switch (sig_how) {
```

```
      /* Si la acción es de bloquear, primero se elimina la señal
        SIGKILL (si es que está en el conjunto) y luego se actualiza
        la máscara */
      case SIG_BLOCK:
      sigdelset((sigset_t *)&sig_set, SIGKILL);
      for (i = 1; i < _NSIG; i++) {
            if (sigismember((sigset_t *)&sig_set, i))
                  sigaddset(&mp->mp_sigmask, i);
      }
      break;
     /* Si la acción es de desbloquear se actualiza la máscara y se
        comprueba si hay señales pendientes */
      case SIG_UNBLOCK:
      for (i = 1; i < _NSIG; i++) {
            if (sigismember((sigset_t *)&sig_set, i))
                  sigdelset(&mp->mp_sigmask, i);
      }
      check_pending();
      break;
     /* Si la acción es cambiar la máscara completamente primero se
elimina  la señal SIGKILL (si es que está en el conjunto),  luego
se actualiza la máscara y por último se comprueba si hay señales
        pendientes */
     case SIG_SETMASK:
      sigdelset((sigset_t *)&sig_set, SIGKILL);
      mp->mp_sigmask = (sigset_t)sig_set;
      check_pending();
      break;
     /* Se usa cuando el argumento parámetro es nulo */
     case SIG_INQUIRE:
      break;
      /* En cualqueir otro caso se produce un error */
     default:
      return(EINVAL);
      break;
  }
  return OK;
}
```

7.4 do_sigsuspend

Encargada de dar soporte a la llamada al sistema SIGSUSPEND. Se utiliza para suspender la ejecución de un proceso hasta que reciba una señal determinada.

```
PUBLIC int do_sigsuspend()
{
  /* Guarda la máscara original */
```

```
  mp->mp_sigmask2 = mp->mp_sigmask;

  /* Cambia la máscara de señales por la nueva */
  mp->mp_sigmask = (sigset_t) sig_set;
  sigdelset(&mp->mp_sigmask, SIGKILL);

  /* Coloca el flag de suspensión y comprueba si hay señales pendientes
     para el proceso */
  mp->mp_flags |= SIGSUSPENDED;
  dont_reply = TRUE;
  check_pending();
  return OK;
}
```

7.5 do_sigreturn

Se encarga de dar soporte a la llamada al sistema SIGRETURN. Se emplea para retornar de un manejador de señales. Se recupera la máscara de señales y se invoca a la tarea del sistema para que recupere el contexto del proceso. Antes de retornar comprueba si el proceso tiene más señales pendientes.

```
PUBLIC int do_sigreturn()
{
  int r;

  mp->mp_sigmask = (sigset_t) sig_set;
  sigdelset(&mp->mp_sigmask, SIGKILL);
  r = sys_sigreturn(who, (vir_bytes)sig_context, sig_flags);
  /* Se comprueba si el proceso tiene más señales pendientes */
  check_pending();
  return(r);
}
```

7.6 do_kill

Las señales generadas a través del kernel y las generadas por un proceso mediante la llamada al sistema KILL, se manejan en do_kill. Dado que una llamada a KILL puede provocar que se envíen señales a más de un proceso, do_kill simplemente llama a check_sig para comprobar que procesos están preparados para recibir señales.

```
PUBLIC int do_kill()
{
   return check_sig(pid, sig_nr);
}
```

7.7 do_ksig

Ciertas señales, como la violación de segmento y DEL, se originan en el kernel. Cuando el kernel detecta tales señales las codifica en un mapa de bits. Tan pronto como el manejador de memoria se encuentre ocioso el kernel le envía un mensaje que contiene la entrada correspondiente al proceso que provocó la señal y dicho mapa de bits. Esos mensajes acaban siendo tratados por esta función. El sistema de ficheros utiliza también este mecanismo para señalizar la escritura en una tubería (pipe) rota (SIGPIPE).

```
PUBLIC int do_ksig()
{

  register struct mproc *rmp;
  int i, proc_nr;
  pid_t proc_id, id;
  sigset_t sig_map;

  /* si no es el kernel -> error */
  if (who != HARDWARE)
    return(EPERM);

  /* no hay que responder al kernel*/
  dont_reply = TRUE;

  /*SIG_PROC dice si el proceso señalado se interrumpe o termina */
  proc_nr = mm_in.SIG_PROC;

  /* entrada en la tabla de procesos del MM */
  rmp = &mproc[proc_nr];

  /* entrada no en uso ó el proceso está colgado -> retornar */
  if((rmp->mp_flags & IN_USE) == 0 || (rmp->mp_flags & HANGING) )
    return(OK);

  /* obtenemos el pid del proceso señalado */
  proc_id = rmp->mp_pid;

  /* extraemos el mapa de bits del mensaje */
  sig_map = (sigset_t) mm_in.SIG_MAP;
  mp = &mproc[0];

  /* asignamos el grupo de procesos */
  mp->mp_procgrp = rmp->mp_procgrp;

  /* Chequea cada bit en sig_map para ver si una señal debe ser enviada
     al MM.*/
  for (i = 1; i <= _NSIG; i++) {
      if (!sigismember(&sig_map, i)) continue;
      switch (i) {
          case SIGINT:
```

```
          case SIGQUIT:
            /* enviamos la señal a todos los  procesos del  grupo.
            id = 0;
            break;
          case SIGKILL:
            /* terminación forzosa que no puede interceptarse ni
            ignorarse */
            id = -1;
/* se envía la señal a todos los procesos menos al INIT */
            break;
          case SIGALRM:
            /* Producida por una alarma de reloj. */
            /* Ignoramos SIGALRM si el proceso destino no ha solicitado
   dicha alarma */
            if ((rmp->mp_flags & ALARM_ON) == 0) continue;
            rmp->mp_flags &= ~ALARM_ON;
            /* sigue recto y sin frenos */
          default:
            id = proc_id;
            break;
      }
      /* Se envía, si es posible, la señal al proceso señalado. */
      check_sig(id, i);
      /* Indicamos al Kernel que ya finalizó la señalización. */
      sys_endsig(proc_nr);
  }
  return(OK);
}
```

7.8 do_alarm

Se ocupa de la llamada al sistema ALARM mediante una llamada a set_alarm ( ), que envía un mensaje a la tarea de reloj indicando que ha de establecer una nueva alarma para el proceso que realizó la llamada.

```
PUBLIC int do_alarm()
   {
       return(set_alarm(who, seconds));
}
```

7.9 set_alarm

Esta función es utilizada por do_alarm para establecer la alarma solicitada. También es usada para desactivar aquellas alarmas aún pendientes tras la finalización del proceso.

```
PUBLIC int set_alarm(proc_nr, sec)
int proc_nr;      /* proceso que quiere la alarma */
int sec;          /* segundos a esperar antes de la señal */
{
```

```
  message m_sig;
  int remaining;
  /* Activa la alarma si el número de segundos es distinto de 0 */
  if (sec != 0)
      mproc[proc_nr].mp_flags |= ALARM_ON;
  else
  /* en caso contrario desactiva la alarma */
      mproc[proc_nr].mp_flags &= ~ALARM_ON;

  /* preparamos el mensaje para la tarea de reloj */
  m_sig.m_type = SET_ALARM;   /* tipo de mensaje */
  m_sig.CLOCK_PROC_NR = proc_nr; /*número de proceso */

  /* Se usan varios cast para solucionar problemas de conversión de tipos
que no nos ocupan */
  m_sig.DELTA_TICKS = (clock_t) (HZ * (unsigned long) (unsigned) sec);
  if ( (unsigned long) m_sig.DELTA_TICKS / HZ != (unsigned) sec)
      m_sig.DELTA_TICKS = LONG_MAX; /* eternidad (realmente CLOCK_T_MAX)*/

   /* enviamos el mensaje a la tarea de reloj */
  if (sendrec(CLOCK, &m_sig) != OK) panic("alarm er", NO_NUM);
  remaining = (int) m_sig.SECONDS_LEFT;
  if (remaining != m_sig.SECONDS_LEFT || remaining < 0)
      remaining = INT_MAX;
  return(remaining);
}
```

7.10 do_pause

Realiza una llamada al sistema PAUSE. El proceso permanecerá en pausa hasta que se le mande una señal.

```
PUBLIC int do_pause()
{
  /* activa el bit PAUSE para el proceso de usuario */
  mp->mp_flags |= PAUSED;
  /* no hay que enviarle un mensaje de respuesta al proceso */
  dont_reply = TRUE;
  return(OK);
}
```

7.11 do_reboot

Sólo se permite su uso a programas con el uid del superusuario. Asegura que todos los procesos terminan de manera ordenada y que el FS es actualizado antes de que el kernel cierre el sistema.

```
PUBLIC int do_reboot()
{
  register struct mproc *rmp = mp;
  char monitor_code[64];

  /* si no es superusuario -> salir */
  if (rmp->mp_effuid != SUPER_USER)   return EPERM;

  switch (reboot_flag) {
```

```
  case RBT_HALT:
  case RBT_REBOOT:
  case RBT_PANIC:
  case RBT_RESET:
      break;
  case RBT_MONITOR:
      /* copia el código de retorno al monitor que se especifica en
reboot_code a la variable monitor_code */
      if (reboot_size > sizeof(monitor_code)) return EINVAL;
      memset(monitor_code, 0, sizeof(monitor_code));
      if (sys_copy(who, D, (phys_bytes) reboot_code,
            MM_PROC_NR, D, (phys_bytes) monitor_code,
            (phys_bytes) reboot_size) != OK) return EFAULT;
      if (monitor_code[sizeof(monitor_code)-1] != 0) return EINVAL;
      break;
  default:
      return EINVAL;
  }
  /* Mata a todos los procesos excepto a init */
  check_sig(-1, SIGKILL);


  /* Notifica al sistema de ficheros la terminación del INIT */
  tell_fs(EXIT, INIT_PROC_NR, 0, 0);

  /* Guarda todo antes de salir */
  tell_fs(SYNC,0,0,0);

  sys_abort(reboot_flag, monitor_code);  /* aborta la ejecución y retorna
al monitor */
  /* nunca pasará por aquí */
}
```

7.12 sig_proc

Es la función que se encarga realmente de enviar las señales.

Envía una señal a un proceso. Comprueba si la señal debe ser capturada, ignorada o bloqueada. Si tiene que capturarse, tenemos que coordinarnos con el núcleo para que guarde el contexto del proceso en la pila. Además, tiene que modificar el contador de programa y el puntero de pila para que, al seguir ejecutándose el proceso, se ejecute el manejador de la señal. Cuando éste retorne, se llamará a do_sigreturn para que el núcleo restaure el contexto original. Si no hay suficiente espacio en la pila, matamos al proceso.

```
PUBLIC void sig_proc(rmp, signo)
register struct mproc *rmp;   /* puntero al proceso a ser señalizado */
int signo;                    /* señal a enviar (de 1 a _NSIG) */
{
  vir_bytes new_sp;
  int slot;
  int sigflags;
  struct sigmsg sm;
```

```
  slot = (int) (rmp - mproc); /* índice a entrada de proceso en la tabla
*/

  /* si el proceso no está en uso -> panic! */
  if (!(rmp->mp_flags & IN_USE)) {
      printf("MM: signal %d sent to dead process %d\n", signo, slot);
      panic("", NO_NUM);
  }
  /* si el proceso está colgado -> panic! */
  if (rmp->mp_flags & HANGING) {
      printf("MM: signal %d sent to HANGING process %d\n", signo, slot);
      panic("", NO_NUM);
  }
  /* El proceso está en modo traza? */
  if (rmp->mp_flags & TRACED && signo != SIGKILL) {
      /* Siempre es una terminación forzosa */
      unpause(slot);
stop_proc(rmp, signo);  /* una señal detiene el proceso (por estar
  en modo traza */
      return;     /* termina aquí para estos procesos */
  }
  /* Si la señal se ignora -> retornar */
  if (sigismember(&rmp->mp_ignore, signo)) return;

  /* la señal se debe bloquear? */
  if (sigismember(&rmp->mp_sigmask, signo)) {
      sigaddset(&rmp->mp_sigpending, signo);   /* bloquear la señal */
      return;
  }
  sigflags = rmp->mp_sigact[signo].sa_flags;
  /* la señal debe capturarse? */
  if (sigismember(&rmp->mp_catch, signo)) {
      /* si el proceso está suspendido, por usar do_sigsuspend */
      if (rmp->mp_flags & SIGSUSPENDED)
            sm.sm_mask = rmp->mp_sigmask2;
      else
            sm.sm_mask = rmp->mp_sigmask;
      sm.sm_signo = signo;
      sm.sm_sighandler = (vir_bytes) rmp->mp_sigact[signo].sa_handler;
      sm.sm_sigreturn = rmp->mp_sigreturn;
      /* Obtenemos el puntero de pila */
      sys_getsp(slot, &new_sp);
      sm.sm_stkptr = new_sp;
      /* Reservamos espacio para el contexto del proceso */
      new_sp -= sizeof(struct sigcontext)
                        + 3 * sizeof(char *) + 2 * sizeof(int);
      /* No hay suficiente espacio */
```

```
      if (adjust(rmp, rmp->mp_seg[D].mem_len, new_sp) != OK)
            goto doterminate; /* no hay espacio suficiente en pila,
                                 salta a la etiqueta para terminar */
      rmp->mp_sigmask |= rmp->mp_sigact[signo].sa_mask;
      /* Proceso bloqueado por señal anterior del mismo tipo cuando
         manejábamos la señal */
      if (sigflags & SA_NODEFER)
            sigdelset(&rmp->mp_sigmask, signo);
      else
            sigaddset(&rmp->mp_sigmask, signo);
      /* Si se ha especificado en los flags se resetea el manejador de la
señal al recibirla */
      if (sigflags & SA_RESETHAND) {
            sigdelset(&rmp->mp_catch, signo);
            rmp->mp_sigact[signo].sa_handler = SIG_DFL;
      }
      /* notificamos al kernel */
      sys_sendsig(slot, &sm);
      /* La señal ya no está pendiente */
      sigdelset(&rmp->mp_sigpending, signo);
      /* termina cualquier llamada al sistema que el proceso pueda estar
esperando */
      unpause(slot);
      return;
  }
doterminate:
  /* La señal no puede o no debería capturarse. Matamos al proceso. */
  rmp->mp_sigstatus = (char) signo;
  if (sigismember(&core_sset, signo)) {
      /* Cambiamos al entorno de FS del usuario y generamos un core */
      tell_fs(CHDIR, slot, FALSE, 0);
      dump_core(rmp);
  }
  mm_exit(rmp, 0); /* se elimina el proceso */
}
```

7.13 check_sig

Comprueba si se puede enviar una señal al proceso o grupo de procesos indicado(s).

```
PUBLIC int check_sig(proc_id, signo)
pid_t proc_id;    /* pid del proceso a señalizar, o 0 o -1, o -pgrp */
int signo;        /* señal a enviar al proceso (0 a _NSIG) */
{
  register struct mproc *rmp;
  int count;                  /* contador de señales enviadas */
  int error_code;
```

```
  /* si el número de señal no es correcto -> error */
  if (signo < 0 || signo > _NSIG) return(EINVAL);
  /* si se intenta enviar SIGKILL a init de forma aislada */
  if (proc_id == INIT_PID && signo == SIGKILL) return(EINVAL);
  /* Busca los procesos a los que se debe enviar la señal en la tabla de
procesos */
  count = 0;
  error_code = ESRCH;

  /* Para cada entrada en la tabla del MM. */
  for (rmp = &mproc[INIT_PROC_NR]; rmp < &mproc[NR_PROCS]; rmp++) {
      /* si esta entrada no está en uso, pasa */
      if ( (rmp->mp_flags & IN_USE) == 0) continue;
      if (rmp->mp_flags & HANGING && signo != 0) continue;
      /* si el identificador de la entrada no es correcto, pasa */
      if (proc_id > 0 && proc_id != rmp->mp_pid) continue;
      if (proc_id == 0 && mp->mp_procgrp != rmp->mp_procgrp) continue;
      if (proc_id == -1 && rmp->mp_pid == INIT_PID) continue;
      if (proc_id < -1 && rmp->mp_procgrp != -proc_id) continue;
      /* Comprueba el uid real y efectivo si el proceso origen no
pertenece al super usuario */
      if (mp->mp_effuid != SUPER_USER
          && mp->mp_realuid != rmp->mp_realuid
          && mp->mp_effuid != rmp->mp_realuid
          && mp->mp_realuid != rmp->mp_effuid
          && mp->mp_effuid != rmp->mp_effuid) {
            error_code = EPERM;
            continue;
      }
      count++;
      if (signo == 0) continue;
      /* llamamos a sig_proc para tratar la señal */
      sig_proc(rmp, signo);
      if (proc_id > 0) break; /* solo un proceso ha sido señalizado */
  }

  /* si proceso llamado no existe o está colgado, no enviamos respuesta*/
  if ((mp->mp_flags & IN_USE) == 0 || (mp->mp_flags & HANGING))
      dont_reply = TRUE;
  return(count > 0 ? OK : error_code);
}
```

7.14 check_pending

Comprueba si una señal pendiente ha sido desbloqueada. La primera que se encuentre en estas condiciones se libera. Si se encuentran múltiples señales pendientes no enmascaradas se liberarán secuencialmente.

```
PRIVATE void check_pending()
{
  int i;
  /* recorremos las señales */
  for (i = 1; i < _NSIG; i++) {
      /* mp_sigpending contiene el mapa de bits del proceso referido
         mediante do_sigmask, do_sigreturn, o do_sigsuspend para
   comprobar si la señal ha sido bloqueada */
      /* si la señal es pendiente y no está enmascarada */
      if (sigismember(&mp->mp_sigpending, i) &&
            !sigismember(&mp->mp_sigmask, i)) {
            /* liberamos la señal */
            sigdelset(&mp->mp_sigpending, i);
            /* enviamos la señal */
            sig_proc(mp, i);
            break;
      }
  }
}
```

7.15 unpause

Termina las llamadas al sistema PAUSE, WAITING ó SUSPENDED.
Responde al proceso con EINTR.
Notifica al FS que se ha realizado una llamada UNPAUSE.

```
PRIVATE void unpause(pro)
int pro;               /* número de proceso */
{
  register struct mproc *rmp;
  rmp = &mproc[pro];
  /* Si el proceso está bloqueado por una llamada a PAUSE */
  if ( (rmp->mp_flags & PAUSED) && (rmp->mp_flags & HANGING) == 0) {
      rmp->mp_flags &= ~PAUSED;
      reply(pro, EINTR, 0, NIL_PTR);
      return;
  }
 /* Si el proceso está bloqueado por una llamada a WAITING */
  if ( (rmp->mp_flags & WAITING) && (rmp->mp_flags & HANGING) == 0) {
      rmp->mp_flags &= ~WAITING;
      reply(pro, EINTR, 0, NIL_PTR);
      return;
  }
  /* Si el proceso está bloqueado por una llamada SIGSUSPEND */
  if ((rmp->mp_flags & SIGSUSPENDED) && (rmp->mp_flags & HANGING) == 0) {
      rmp->mp_flags &= ~SIGSUSPENDED;
```

```
      reply(pro, EINTR, 0, NIL_PTR);
      return;
  }
  /* Notificar al FS */
  tell_fs(UNPAUSE, pro, 0, 0);
}
```

7.16 dump_core

Hace una copia del núcleo de un proceso , si es posible, sobre el fichero core. El núcleo de un proceso que va a morir consta de su espacio de direcciones y de su entrada en la tabla de procesos.

```
PRIVATE void dump_core(rmp)
register struct mproc *rmp;   /* el core que será volcado */
{
  int fd, fake_fd, nr_written, seg, slot;
  char *buf;
  vir_bytes current_sp;
  phys_bytes left;      /* cuidado; 64K puede producir desbordamiento
   de vir_bytes */
  unsigned nr_to_write;       /* es sin signo por ser argumento a la
                                 función write. Sin embargo, debe ser
                                 menor que INT_MAX */
  long trace_data, trace_off;

  slot = (int) (rmp - mproc); /* entrada en la tabla de procesos de MM */

  /* Podemos escribir el fichero core? */
  if (rmp->mp_realuid != rmp->mp_effuid) return;
  if ( (fd = creat(core_name, CORE_MODE)) < 0) return;
  rmp->mp_sigstatus |= DUMPED;

  /* Nos aseguramos que el segmento de pila está actualizado. No queremos
     que adjust() falle a menos que current_sp sea absurdo, pero podría
     fallar por verificación de seguridad. Tampoco queremos realmente que
     el adjust() falle al enviar una señal por verificación de seguridad.
     Tal vez hacer que SAFETY_BYTES sea un parámetro */

  sys_getsp(slot, &current_sp);
  adjust(rmp, rmp->mp_seg[D].mem_len, current_sp);
  /* escribimos el mapa de memoria de todos los segmentos para comenzar el
fichero core */
  if (write(fd, (char *) rmp->mp_seg, (unsigned) sizeof rmp->mp_seg)
      != (unsigned) sizeof rmp->mp_seg) {
      close(fd);
      return;
  }
```

```
  /* Copiamos la entrada en la tabla de procesos del kernel para obtener
     los registros */
  trace_off = 0;
  while (sys_trace(3, slot, trace_off, &trace_data) == OK) {
      if (write(fd, (char *) &trace_data, (unsigned) sizeof (long))
          != (unsigned) sizeof (long)) {
            close(fd);
            return;
      }
      trace_off += sizeof (long);
  }
  /* Bucle para recorrer todos los segmentos y escribirlos */
  for (seg = 0; seg < NR_SEGS; seg++) {
      buf   =   (char   *)   ((vir_bytes)   rmp->mp_seg[seg].mem_vir   <<
CLICK_SHIFT);
      left = (phys_bytes) rmp->mp_seg[seg].mem_len << CLICK_SHIFT;
      fake_fd = (slot << 8) | (seg << 6) | fd;
      /* realizamos la escritura física de los registros en el
         fichero core */
      while (left != 0) {
            nr_to_write = (unsigned) MIN(left, DUMP_SIZE);
            if ( (nr_written = write(fake_fd, buf, nr_to_write)) < 0) {
                  close(fd);
                  return;
            }
            buf += nr_written;
            left -= nr_written;
      }
  }
  close(fd);
}
```