LECCIÓN 4: SEÑALES

LECCIÓN 4: SEÑALES

4.1 Introducción

Es un mecanismo de comunicación mínima entre procesos o entre el núcleo y un proceso o entre el teclado y un proceso, por el cual se envía un número (señal). Cada señal tiene un nombre SIGXXX con un significado específico. La comunicación es unidireccional. Es una comunicación rápida.

Se genera una señal para
Avisar a un proceso padre que un proceso hijo termina con un exit, SIGCHLD
Abortar un proceso pulsando las teclas ctrl C, SIGINT
Para matar a un proceso con el kill, SIGKILL
Para avisar a un proceso que ha finalizado una alarma, SIGALRM
Para despertar a un proceso que estaba en pausa

Cuando un proceso recibe una señal, se interrumpe su ejecución, se almacena su estado para posteriormente reanudar su ejecución, se pasa a ejecutar la función que atiende esa señal, esta función esta definida en el proceso receptor, una vez finalizada esta función se reanuda la ejecución del proceso en el punto que se interrumpió.

1) El estado del proceso se guarda en su stack.
2) Se ejecuta el manejador de la señal.
3) Se recupera el estado del proceso y se continúa.

Las señales son atendidas en modo usuario, si el proceso está en modo núcleo, la señal se añade al conjunto de señales pendientes y se atiende cuando se regresa a modo usuario, esto puede causar un pequeño retraso.

Cuando un proceso recibe una señal, si el proceso no se ha preparado para recibirla, el resultado es la muerte del proceso.

Los tipos de señales se encuentran definidos en **include/asm-i386/signal.h**, se pueden clasificar en dos tipos:

señales no tiempo real.
: Son la clásicas, son las primeras 31 señales, cuando se envían solo se envía su número de señal.

señales tiempo real.
: Definidas por la norma POSIX 1003, son configurables por los procesos, cuando se envían se manda información extra a través de la estructura info, si se reciben mas señales cuando se está atendiendo la primera se encolan.

Veamos el listado de señales no tiempo real:

1	SIGHUP	El modem a detectado línea telefónica rota o ha terminado el proceso líder de la sesión
2	SIGINT	Las teclas Ctrl C han sido pulsadas
3	SIGQUIT	Las teclas Ctr \ han sido pulsadas, terminación de terminal

4	SIGILL	Instrucción ilegal
5	SIGTRAP	Traza de los programas
6	SIGIOT, SIGABORT	Instrucción IOT (I/O TRAP), Terminación anormal
7	SIGBUS	Error de bus
8	SIGFPE	Rebosamiento de coma flotante, error aritmético
9	SIGKILL	Matar un proceso, no puede ser desviada a una función
10	SIGUSR1	Señal definida por el usuario
11	SIGSEGV	Violación de segmentación
12	SIGUSR2	Señal definida por el usuario
13	SIGPIPE	Escritura en pipe sin lectores
14	SIGALRM	Señal enviada por el núcleo cuando fin del reloj ITIMER_REAL
15	SIGTERM	Software genera una señal de terminación
16	SIGSTKFLT	Desbordamiento coprocesador matemático
17	SIGCHLD	Señal enviada por el núcleo a un padre cuando este hace un wait, para avisarle que un hijo ha terminado con un exit.
18	SIGCONT	El proceso se lleva a primer o segundo plano
19	SIGSTOP	Suspensión de un proceso, por ejemplo por el debuger
20	SIGTSTP	Suspensión del proceso debido a Ctrl Z del terminal
21	SIGTTIN	Suspensión de un proceso en segundo plano que trata de leer del terminal
22	SIGTTOU	Suspensión de un proceso en segundo plano que trata de escribir en el Terminal
23	SIGURG	Datos urgentes para los sockets
24	SIGXCPU	Sobrepasado limite de tiempo de CPU
25	SIGXFSZ	Sobrepasado tamaño de fichero
26	SIGVTALRM	Fin del temporizador ITIMER_VIRTUAL
27	SIGPROF	Fin del temporizador ITIMER_PROF
28	SIGWINCH	Cambio del tamaño de una ventana, usado por X11
29	SIGIO, SIGPOLL, SIGSLOT	Datos disponibles para una entrada salida
30	SIGPWR	Fallo de alimentación
31	SIGUNUSED	Argumento erróneo en una llamada
32	SIGRTMIN	Marca el limite de señales en tiempo real, < 32 no tiempo real

4.2 Llamadas al sistema para señales

s = kill (pid, sig) - envía una señal a un proceso.

s = signal (sig, &función) – recibe una señal y define la función que la atiende.

s = sigaction (sig, &act, &oldact) - define la acción a realizar cuando se recibe una señal, es la versión POSIX de signal.

residual = alarm (segundos) - planifica una señal SIGALRM después de un cierto tiempo.

s = pause() - suspende al proceso solicitante hasta recibir una señal.

s = sigreturn (&context) - regresa de una señal.

Llamadas al sistema que afectan a grupos de señales:

s = sigemptyset (sigset_t *set) – crea un conjunto de señales vacio.

s = sigfillset (sigset_t *set) – crea un conjunto con todas las señales.

s = sigaddset (sigset_t *set) – añade una señal a un conjunto de señales.

s = sigdelset (sigset_t *set) – borra una señal de un conjunto de señales.

s = sigismemberset (sigset_t *set) – comprueba si una señal pertenece a un conjunto.

s = sigprocmask (how, &set, &old) - examina o cambia la mascara de las señales que se van a bloquear para que queden pendientes.

s = sigpending (set) – un proceso obtiene el conjunto de señales pendientes de ser atendidas.

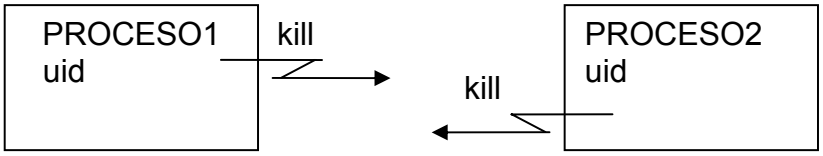
s = sigsuspend(sigmask) - sustituye la mascara de señales y suspende el proceso.

ENVIO DE SEÑALES

La llamada kill, permite enviar señales entre procesos que tienen el mismo identificador real uid o igual identificador efectivo euid, la forma es:

KILL (pid, sig)

Pid -> identificador del proceso
Sig -> tipo de señal a enviar



Sig = 9 es la señal SIGKILL, que mata cualquier proceso, aún en modo tanda, esta

señal no puede ser ignorada.

los posibles valores de pid y la acción de kill asociada son:

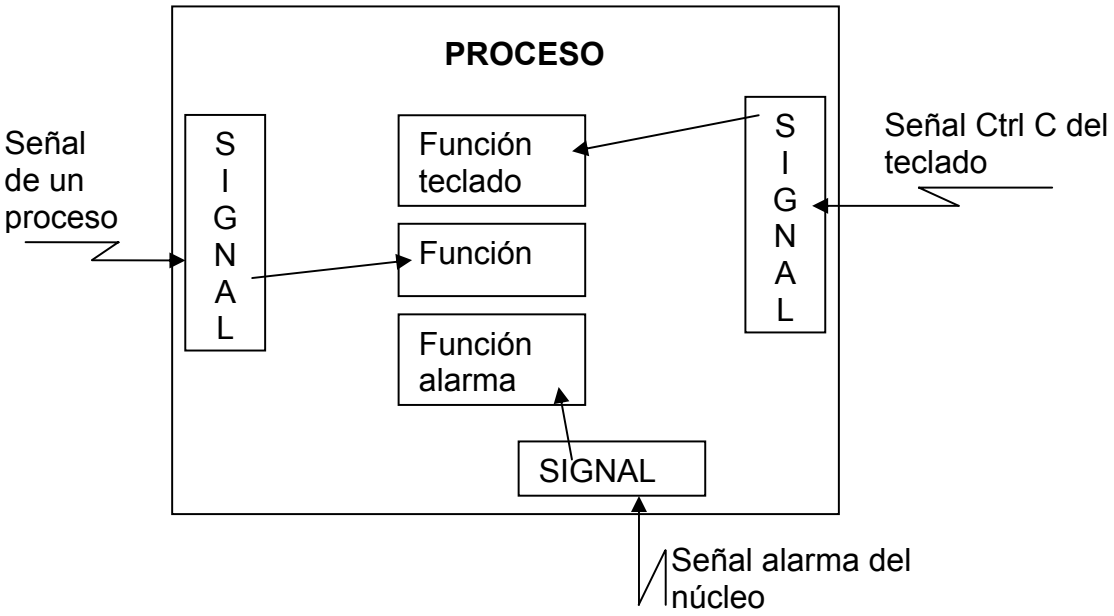
pid = 0 Se envía la señal a todos los procesos del mismo grupo **pgrp** que el proceso emisor.
pid = -1 Se envía la señal a todos los proceso, salvo 0 idle y 1 init. Necesario ser root.
pid > 0 Se envía la señal al proceso que tiene este pid.
pid < -1 Se envía la señal a todos los procesos del grupo **pgrp = - pid**.

RECEPCIÓN DE SEÑALES

Esta llamada se utiliza para recibir algún tipo de señal y suministrar la dirección de la función que va a atender la señal.

SIGNAL (señal, funcion)

int señal; señal a recibir
int (*función)(); función que se ejecuta cuando se recibe la señal



Después de ejecutar un SIGNAL, si se recibe una señal se efectúa:

1. Se interrumpe la ejecución del proceso
2. Se almacena el estado del proceso en su stack
3. Se ejecuta la función manejadora de la señal
4. Se recupera el estado del proceso

Después de recibir una señal, es necesario volver a prepararse para recibir otra vez la señal con otro SIGNAL, ya que si no, se puede recibir la señal y mata al proceso

(acción por defecto que se carga por el núcleo después de atenderse la señal).

Existen unas funciones predefinidas por el sistema y de uso público para atender una señal:
La función **SIG_IGN**, para que todas las señales se ignoren (excepto SIGKILL)
La función **SIG_DFL** para ejecutar la acción por defecto de matar al proceso.

Ejemplo:
Supongamos que lanzamos un comando en segundo plano.
Command &
Es indeseable que la señal DEL, del teclado pueda afectar a ese proceso, así, el Shell después de ejecutar el FORK, pero antes del EXEC deberá hacer

Signal (SIGINT, SIG_IGN);
Signal (SIGQUIT, SIG_IGN);

que inhiben las señales DEL (Ctrl C), y QUIT (Ctrl \).

Existe otra llamada al sistema para recibir una señal y definir una función manejadora según la norma POSIX.

sigaction (int **signum,** const struct sigaction ***act,** struct sigaction ***oldact);**

signum, es la señal a recibir.

act y **oldact** son las estructuras que definen el nuevo y viejo comportamiento para tratar la señal.

ALARMA

s = alarm (segundos)

Esta llamada se utiliza por un proceso para decirle al núcleo que le envíe la señal SIGALRM al cabo de un cierto tiempo, su parámetro especifica el tiempo en segundos reales, después del cual el núcleo envía al proceso la señal SIGALRM.

Un proceso solo puede tener una alarma pendiente, para anular una alarma, basta hacer una llamada de alarma con parámetro cero.

Existen varios contadores.

ITIMER_REAL – Contador que se decrementa en tiempo real, cuando llega a cero se envía la señal SIGALRM.
ITIMER_VIRTUAL – Contador que se decrementa cuando el proceso se ejecuta, cuando llega a cero se envía la señal SIGVTALRM.
ITIMER_PROF – Contador que se decrementa cuando se ejecuta el proceso y cuando el sistema operativo se ejecuta a cuenta del proceso, cuando llega a cero se envía la

señal SIGPORF.

PAUSA

pause ( )

Esta llamada le dice al S.O. que suspenda al proceso hasta que se reciba una señal.

4.3 Estructuras de datos

Veamos las estructuras de datos y variables que tiene LINUX, que permiten el uso de las señales.

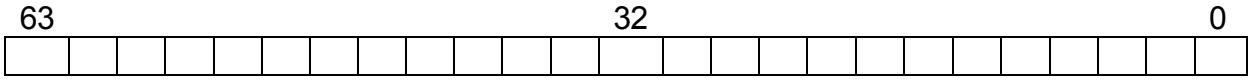
La información y gestión de señales de un proceso se lleva a cabo en unos campos específicos dentro de la estructura **task_struct** del proceso que son.

►sigset_t **signal** conjunto de señales pendientes de atender por este proceso.

►sigset_t **blocked** conjunto de señales que se quieren bloquear para ese proceso.

La estructura **sigset_t** está definida en **include/asm-i386/signal.h**

Es un vector de unsigned long que representa un conjunto de señales, donde cada bit corresponde a una señal.




```
typedef struct {
    unsigned long sig [_NSIG_WORDS];
} sigset_t
```

►int **sigpending** utilizado como un flag, a uno indica que el proceso tiene señales pendientes.

►int **exit_signal** tipo de señal que se envía al proceso padre cuando termina el proceso.

►int **exit_code** valor de retorno a devolver al proceso padre cuando el hijo termina. El byte menos significativo contiene la señal que ha causado la finalización del hijo, y el byte mas significativo contiene el código de retorno devuelto por el proceso hijo cuando realiza el exit.

►spinlock_t **sigmask_lock** protege y bloquea las señales.

►struct signal_struct **sig** es un vector de estructuras que contiene las direcciones de las funciones que van a atender a las señales.

La estructura **signal_struct** se define en el fichero **linux/sched.h**

struct **signal_struct**

int **count** – contador con el número de procesos creados con clone que referencian a esta estructura.

struct sigaction **action**[_NSIG] – vector con las funciones manejadoras.

struct sigaction contiene la acción a tomar por un proceso cuando se recibe una señal, existe una estructura por cada señal, se encuentra en **incluye/asm-i386/signal.h**, un proceso tiene una estructura sigaction por cada señal pendiente, sus campos son:

_sighandler_t **sa_handler** - dirección de la función manejadora de la señal, o SIG_DFL = NULL, o SIG_IGN.

unsigned long **sa_flags** – directrices para el proceso cuando recibe la señal.

void (***sa_restorer**) **(void)** – no se utiliza, puntero a la pila de la función manejadora.

sigset_t **sa_mask** – para cada señal, mascara de señales a bloquear cuando se ejecuta la función manejadora de esta señal, se añade la señal que está siendo atendida.

valores para **sa_flags**

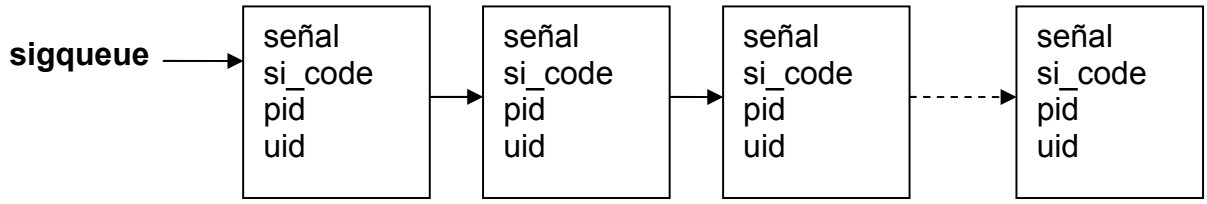
SA_NOCLDSTOP – se pone a cero si la señal recibida es SIGHLD, el hijo termina.
SA_ONESHOT, SA_RESETHAND – reinstalar la función manejadora por defecto SIG_DFL del sistema cuando se termine de atender la señal.
SA_RESTART – ejecutar la llamada al sistema interrumpida por la recepción de la señal.
SA_NOMASK, SA_NODEFER – no bloquear la recepción de esta señal cuando se esta atendiendo por su función manejadora.

►struct signal_queue ***sigqueue, **sigqueue_tail,** lista con las señales pendientes de tiempo real de un proceso, para asegurar que todas son atendidas.

La estructura **signal_queue** está definida en **include/linux/signal.h**

```
struct signal_queue
{
   struct signal_queue *next;
```

```
    siginfo_t    info;
};
```



struct siginfo se encuentra definida en **incluye/asm-i386/siginfo.h**

Estructura con información adicional relativa a la señal pendiente de tiempo real.

```
typedef struct siginfo {
  int si_signo;    /* la señal enviada */
  int si_errno;    /* valor de errno en el momento de hacer la llamada */
  int si_code;     /* quien envió la señal */

  union {      /* depende del valor de si_code */
        int _pad[SI_PAD_SIZE]; /* ajusta el tamaño de siginfo_t a 128*sizeof(int) */

   /* kill ( )  */
        struct {
           pid_t _pid;        /* pid del proceso que hace la llamada */
           uid_t _uid;        /* uid del proceso que hace la llamada */
    } _kill;
```

valores para **si_code** Digital reserva valores positivos para las señales generadas por el núcleo.

```
#define SI_USER    0          /* enviada por un usuario con kill o sigsend o raise */
#define SI_KERNEL  0x80     /* enviada por el núcleo */
#define SI_QUEUE   -1         /* enviada por sigqueue */
#define SI_TIMER   -2        /* enviada al finalizar el timer */
#define SI_MESGQ   -3        /* enviada por RT mesq state chg */
#define SI_ASYNCIO -4        /* enviada por AIO */
#define SI_SIGIO   -5        /* enviada por queued SIGIO */
```

4.4 Procedimientos para el manejo de señales

PROCEDIMIENTOS PARA SIGSET

Son procedimientos de apoyo que tratan con la estructura **sigset_t** para tener acceso a bits.

Veamos los siguientes procedimientos, en una versión independiente de la plataforma hardware, **en lenguaje C** definidos en **include/linux/signal.h**

sigaddset

Añade una señal a un conjunto, poniendo a uno el bit correspondiente.

```
 extern inline void sigaddset (sigset_t  *set, int _sig)
 {
/*  resta uno a la señal, el indice del vector comienza en cero */
       unsigned long sig  = _sig  -  1;

       if   (_NSIG_WORDS == 1) /* tamaño de la palabra  unsigned long*/
            set->sig[0] |= 1UL << sig;
       else

/* ajustar el indice dividiendo por NSIG_BPW bits por palabra */
            set->sig[sig/_NSIG_BPW] |= 1UL << (sig % _NSIG_BPW);
 }
```

sigdelset

Quita una señal de un conjunto, poniendo a cero el bit correspondiente.

```
 extern inline void sigdelset  (sigset_t *set, int _sig)
{
  unsigned long sig = _sig - 1;
  if   (_NSIG_WORDS == 1)
       set->sig[0] &= ~ (1UL << sig); /* pone un cero */
  else
       set->sig[sig/_NSIG_BPW] &= ~(1UL << (sig%_NSIG_BPW));
}
```

sgismember
Pregunta si una señal esta en el conjunto devolviendo su bit.

```
extern inline int sigismember  (sigset_t *set,  int _sig)
{
  unsigned long sig = _sig - 1;
  if (_NSIG_WORDS == 1)
        return 1 & (set->sig[0] >> sig);
  else
        return 1 & (set->sig[sig/_NSIG_BPW]>>(sig%_NSIG_BPW));
}
```

sigfindinword
Esta function encuentra la posición del primer bit que es uno en el parámetro word se apoya de la función ffz que busca ceros.

```
extern inline int sigfindinword (unsigned long word)
{
  return ffz (~ word);
}
```

sigmask
Es una macro que convierte un número de señal en un mapa de bits con un solo uno en ka correspondiente posición.

```
#define sigmask(sig)    (1UL << ((sig) - 1))
```

sigemptyset
Pone a cero todo el conjunto de señales. En el caso general utiliza la instrucción **memset** para poner ceros en set. Cuando **_NSIG_WORDS** vale 1 (palabras de 2 bytes) directamente pone a uno.

```
extern inline void sigemptyset  ( sigset_t  *set )
{
  switch (_NSIG_WORDS) {
      default:
        memset(set, 0, sizeof(sigset_t));
        break;
      case 2:  set -> sig [1] = 0;
      case 1: set -> sig [0] = 0;
        break;
   }
}
```

sigfillset
Pone a uno todo el conjunto de señales, función análoga a la anterior.

sigaddsetmask
Coloca los bits a uno y a cero según los bits de mask.

```
extern inline void sigaddsetmask  (sigset_t *set, unsigned long mask)
 {
   set->sig[0]  |=  mask;
 }
```

siginitset
Coloca los 32 primeros bits según la mascara y el resto a cero.

```
extern inline void siginitset  (sigset_t *set,  unsigned long mask)
 {
   set->sig[0] = mask;
   switch (_NSIG_WORDS) {
      default:
         memset ( &set->sig[1], 0, sizeof(long)*(_NSIG_WORDS -1 ) );
         break;
      case 2: set->sig[1] = 0;
      case 1:
      }
 }
```

siginitsetinv
Analoga a la anterior coloca los 32 primeros bits contrario a la mascara y el resto a uno.

Veamos estos mismos procedimientos, en **lenguaje ensamblador,** dependiente de la plataforma, definidos en **include/asm-i386/signal.h**

sigaddset
Utiliza la instrucción **btsl** para poner un bit a uno en su operando.

```
 extern __inline__ void sigaddset (sigset_t *set,  int _sig)
{
   __asm__( " btsl %1, %0 "  :  " =m " (*set)  : " ir " (_sig - 1)
                 : "cc");
}
```

sigdelset
Utiliza la instrucción **btrl** para poner un bit a cero en su operando.

```
 extern __inline__ void sigdelset  (sigset_t *set,  int _sig)
{
  __asm__( " btrl %1,  %0  " : " =m " (*set) : " ir " (_sig - 1)
          : "cc");
}
```

sgismember
Elije una implementación preguntando si su argumento **sig** es una constante de compilación, si lo es la función sera reemplazada por la función **_const_sigsmember** para realizar su trabajo, si no se remplaza por la función mas general **_gen_sigismember**, esta utiliza la instrucción **btl** que testea un bit de su operando.

```
#define sigismember  ( set, sig )                    \
  (__builtin_constant_p ( sig )  ?                  \
   __const_sigismember ( (set) , ( sig ) ) :          \
   __gen_sigismember  ( (set), ( sig ) ) )


extern __inline__ int  __const_sigismember ( sigset_t *set, int _sig)
{
  unsigned long sig = _sig - 1;
  return 1 & ( set->sig [sig / _NSIG_BPW]  >> (sig % _NSIG_BPW));
}

extern __inline__ int  __gen_sigismember (sigset_t *set, int _sig)
{
  int ret;
  __asm__( " btl  %2, %1 \ n \ tsbbl  %0, %0 "
          : " =r " (ret) :  " m " ( *set ),  " ir "(_sig-1) : "cc");
  return ret;
}
```

sigmask
Esta versión es igual que la versión independiente de la plataforma.

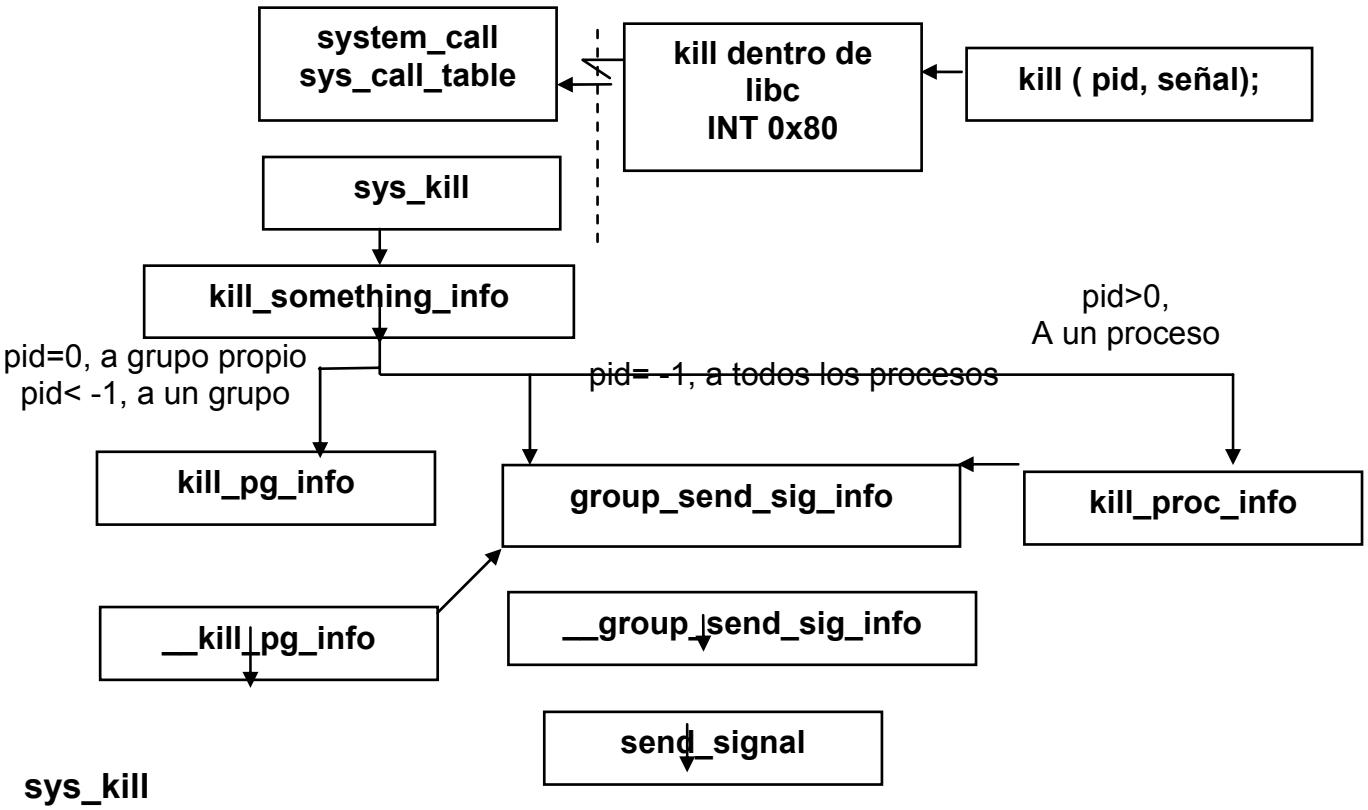
```
#define sigmask ( sig )      (1UL << ((sig) - 1))
```

sigfindinword
Utiliza la instrucción **bsfl** para buscar un bit a uno en su operando.

```
extern __inline  __ int sigfindinword ( unsigned long word )
{
  __asm__( " bsfl %1,%0 " : " =r " ( word ) : " rm "( word ) : "cc");
        return   word;
 }
```

PROCEDIMIENTOS PARA EL ENVIO DE SEÑALES

La llamada al sistema **kill (pid, señal)**; permite enviar una señal a un proceso, cuando llega al núcleo a través de **system_call**, invoca a la función **sys_kill**. Los fuentes se encuentran en el fichero **kernel/signal.**



sys_kill

Rellena campos de la estructura **info** y llama a **kill_something_info** para realizar el trabajo.

```
 sys_kill ( int pid, int sig )
{
  struct siginfo info;

  info.si_signo = sig; /* señal a enviar */
  info.si_errno = 0;
  info.si_code = SI_USER; /* es un usuario quien envía la señal */
  info.si_pid = current -> pid; /* pid del proceso emisor */
  info.si_uid = current -> uid; /* identificador de usuario del proceso emisor */

  return  kill_something_info  (sig,  &info,  pid);
 }
```

kill_something_info

Los parámetros son el pid, la señal sig y la estructura info, consiste en una serie de if encadenados que dependiendo del valor de pid llaman a una determinada función.

- si pid es cero → kill_pg_info
- si pid es -1 → group_send_sig_info
- si pid es negativo → kill_pg_info
- si pid es positivo → kill_proc_info

```
kill_something_info ( int sig, struct siginfo *info, int  pid)
{
  /* si pid es cero se quiere enviar la señal a todos los procesos
   * del mismo grupo, hacerlo con kill_pg_info */
  if (!pid) {
     return  kill_pg_info  (sig, info, current -> pgrp);

  /* si pid es -1 se quiere enviar a todos los procesos en el sistema,
   * excepto a  los procesos 0 idle , 1 init y el proceso que envia*/
  } else if (pid == -1) {
     int retval = 0, count = 0;
     struct task_struct * p;
     read_lock(&tasklist_lock);

     for_each_task (p) {         /* macro para barrer todos los procesos */
        if  (p->pid > 1 && p != current) {
           /* si no es el proceso idle, init o el actual, enviar la señal */
           int err = group_send_sig_info (sig, info, p);
           ++count;    /* se incrementa cada vez que se encuentra un proceso */
           if (err != -EPERM)
              /* se devuelve 0, error, o el nº del proceso que ha recibido la señal */
              retval = err;
     }
    }
    read_unlock(&tasklist_lock);
    /* si no se ha encontrado ningún proceso error, sino cero es éxito */
    return count ? retval : -ESRCH;

    /* si pid es negativo con otro valor distinto de -1, el valor absoluto
     * del pid especifica el grupo de procesos al que se envia la señal */
  } else if (pid < 0) {
        return kill_pg_info (sig, info, -pid);
  } else {

        /* pid es positivo enviar la señal a un proceso particular, */
        return   kill_proc_info (sig, info, pid);
  }
}
```

kill_pg_info

Bloquea y desbloquea la lista de procesos y llama a __kil_pg_info

```
1153 kill_pg_info(int sig, struct siginfo *info, pid_t pgrp)
1154 {
1155        int retval;
1156
1157        read_lock(&tasklist_lock);
1158        retval = __kill_pg_info(sig, info, pgrp);
1159        read_unlock(&tasklist_lock);
1160
1161        return retval;
1162 }
```

__kill_pg_info

Esta function evia una señal y la estructura info a un grupo de procesos, si pid = 0 al grupo del proceso que envía, si pid < 0 al grupo que tienen pgrp = -pid.

```
kill_pg_info (int sig, struct siginfo *info, pid_t pgrp)
{
  int retval = -EINVAL;
  if (pgrp > 0) {
     struct task_struct *p;
     int found = 0;

     retval = -ESRCH;
     read_lock (&tasklist_lock);
     for_each_task(p) {  /* macro para barrer todos los procesos */
        if (p->pgrp == pgrp) {
           /* si pertenece al grupo enviar señal */
           int err = group_send_sig_info (sig, info, p);
           if (err != 0)
              retval = err;
        else
        found++; /* se incrementa con cada proceso encontrado */
        }
     }
     read_unlock (&tasklist_lock);
     if (found)
       retval = 0; /* se envio señal a algún proceso */
  }
  return retval;
}
```

kill_proc_info

Esta function evia una señal y la estructura info a un proceso determinado por su pid. Se apoya en la función group_send_sig_info.

```
kill_proc_info  (int sig,  struct siginfo *info,  pid_t  pid)
{
   int error;
   struct task_struct *p;

   read_lock (&tasklist_lock);

   /*busca al proceso por su pid */
   p = find_task_by_pid (pid);
   error = -ESRCH;

   /* envia la señal y la estructura info*/
   if (p)
       error = group_send_sig_info (sig, info, p);
   read_unlock(&tasklist_lock);
   return error;
}
```

group_send_sig_info

Chequeo de permisos, bloqueo de IRQ y llama a __group_send_sig_info.

```
int group_send_sig_info(int sig, struct siginfo *info, struct task_struct *p)
1115 {
1116        unsigned long flags;
1117        int ret;
1118
1119        ret = check_kill_permission(sig, info, p);
1120        if (!ret && sig && p->sighand) {
1121                spin_lock_irqsave(&p->sighand->siglock, flags);
1122                ret = __group_send_sig_info(sig, info, p);
1123                spin_unlock_irqrestore(&p->sighand->siglock, flags);
1124        }
1125
1126        return ret;
1127 }
```

__group_send_sig_info

Función de paso que llama a send_signal

```
static int__group_send_sig_info(int sig, struct siginfo *info, struct task_struct *p)
1041 {
1042        int ret = 0;
1043
1044        assert_spin_locked(&p->sighand->siglock);
1045        handle_stop_signal(sig, p);
1046
1047        if (((unsigned long)info > 2) && (info->si_code == SI_TIMER))
1048                /*
```

```
1049                * Set up a return to indicate that we dropped the signal.
1050                */
1051               ret = info->si_sys_private;
1052
1053        /* Short-circuit ignored signals.  */
1054        if (sig_ignored(p, sig))
1055                return ret;
1056
1057        if (LEGACY_QUEUE(&p->signal->shared_pending, sig))
1058                /* This is a non-RT signal and we already have one queued.  */
1059                return ret;
1060
1061        /*
1062         * Put this signal on the shared-pending queue, or fail with EAGAIN.
1063         * We always use the shared queue for process-wide signals,
1064         * to avoid several races.
1065         */
1066        ret = send_signal(sig, info, p, &p->signal->shared_pending);
1067        if (unlikely(ret))
1068                return ret;
1069
1070        __group_complete_signal(sig, p);
1071        return 0;
1072 }
```

send_signal

Es la function que realmente realiza el envío de la señal a un proceso, se le pasa como parámetros, la **señal** a enviar, la estructura **info** y un puntero ***t** a la estructura task_struct del **proceso receptor**.

1. Comprueba todos los **permisos**.
2. Encola la señal en la lista **sigqueue**.
3. Pone la señal en el conjunto de señales pendientes **signal**.
4. Pone a uno el flag de señales pendientes **sigpending**.

```
group_send_sig_info (int sig,  struct siginfo *info,  struct task_struct *t)
{
  unsigned long flags;
  int ret;

  #if DEBUG_SIG
  printk("SIG queue (%s:%d): %d ",  t->comm,  t->pid,  sig);
  #endif

  ret = -EINVAL;
  /* testea el rango de la señal */
  if (sig < 0 || sig > _NSIG)
     goto out_nolock; /* salta al final de esta función */
  ret = -EPERM;
```

```
/* (1) hacer comprobaciones de permisos */
/* comprueba si la señal tiene todos los permisos */
 * pregunta si existe información adicional y si quien envía la señal es un
 * usuario del mismo grupo, solo el núcleo o el root pueden enviar a
 * otros usuarios */
if (( !info || (( unsigned long) info != 1 && SI_FROMUSER ( info ) ) )

/* pregunta si la señal es SIGCONT (el proceso se lleva a primer o segundo
 * plano), única señal que un usuario puede enviar a otros usuarios, pero
 * de la misma sesión */
   && ( (sig != SIGCONT ) ||  ( current->session != t->session ) )
 /* comprueba con xor que el identificador de usuario efectivo (euid) y real (uid)
 * del proceso que envia son iguales al identificador de usuario guardado (suid)
 * y real (uid) del proceso que recibe (t)*/
   && (current->euid ^ t->suid)
   && (current->euid ^ t->uid)
   && (current->uid ^ t->suid)
   && (current->uid ^ t->uid)
   && !capable(CAP_SYS_ADMIN))
/* en este punto si el usuario no tiene los permisos adecuados la señal
 * no se envía saltar el código */
goto   out_nolock;

ret = 0;
/* no se envía la señal cero, ni se envía señales a procesos zombis
 * (la estructura de funciones esta a cero*/
if (!sig || !t->sig)
  goto out_nolock;

spin_lock_irqsave (&t->sigmask_lock, flags);

 /* dependiendo del tipo de señal se realiza un trabajo extra antes de
  * enviar la señal*/
switch (sig) {

case SIGKILL: case SIGCONT:     /* Despertar al proceso si esta parado  */
   if (t->state == TASK_STOPPED)   wake_up_process(t);

   /* pone a cero el código de salida, si el proceso fue parado con la señal
    * SIGSTOP, el campo exit_code se utilizó para comunicarle al padre la
    * señal de stop */
   t->exit_code = 0;

   /* coloca una mascara para borrar las señales pendientes SIGSTOP,
    * SIGTSTP, SIGTTIN, o SIGTTOU, que paran un proceso, en
    * respuesta a SIGCOUNT o SIGKILL*/
   sigdelsetmask(&t->signal,
            (sigmask(SIGSTOP) | sigmask(SIGTSTP) |
             sigmask(SIGTTOU) | sigmask(SIGTTIN)));
```

```
      /* comprueba si el proceso receptor todavía tiene señales pendientes,
       * después de las anteriormente borradas */
      recalc_sigpending(t);
      break;

    case SIGSTOP: case SIGTSTP:
    case SIGTTIN: case SIGTTOU:
      /* en el caso de recibir estas señales que paran a un proceso, borra
       * la señal pendiente SIGCONT , por el contrario SIGKILL nunca
       * debe bloquearse*/
      sigdelset (&t->signal, SIGCONT);
      /* Comprobar si existen señales pendientes */
      recalc_sigpending(t);
      break;
  }

  /* pregunta si el proceso receptor puede ignorar la señal */
  if (ignored_signal (sig, t))
      goto out;

  /* pregunta si la señal es del grupo "no tiempo real", estas no se ponen
   * en cola y si es un una segunda instancia de la misma señal cuando
   * se está atendiendo la primera no se atiende */
  if ( sig < SIGRTMIN ) {
      /* pregunta si la señal es una segunda instancia comprobando si pertenece
       * al conjunto de señales pendientes */
      if ( sigismember ( &t->signal, sig ) )
          goto out;

  } else {

      /* señales del grupo "tiempo real" deben ponerse en cola */
      struct signal_queue *q = 0;

      /* el número de señales en cola no debe superar el máximo permitido
       * por razones de seguridad ya que se puede consumir la memoria del núcleo*/
      if (nr_queued_signals < max_queued_signals) {

          /* el núcleo busca memoria para el nodo */
          q = ( struct signal_queue * )
          kmem_cache_alloc ( signal_queue_cachep, GFP_KERNEL);
      }

      /* (2) poner en la cola la señal */

       /* si es asignado espacio para el nodo de la señal, colocar información
        * de la señal  */
      if (q) {
          nr_queued_signals++;     /* incrementa el nº total de señales pendientes */
          q->next = NULL;
```

```
        /* añade el nuevo nodo a la cola de señales pendientes del proceso
         * receptor */
        *t->sigqueue_tail = q;
        t->sigqueue_tail = &q->next;
        /* llena los distintos campos del nodo con la información de la
         * estructura info */
        switch ((unsigned long) info) {
        case 0:      /* señal enviada por el usuario */
           /* se rellena la estructura info del receptor */
           q->info.si_signo = sig;
           q->info.si_errno = 0;
           q->info.si_code = SI_USER;
           q->info.si_pid = current->pid;
           q->info.si_uid = current->uid;
           break;

        case 1: /* señal enviada por el núcleo */
           /* se rellena la estructura info del receptor */
           q->info.si_signo = sig;
           q->info.si_errno = 0;
           q->info.si_code = SI_KERNEL;
           q->info.si_pid = 0;
           q->info.si_uid = 0;
           break;

        default:     /* el caso normal */
        /* se copia la estructura info del receptor con el info del emisor */
           q->info = *info;
           break;
        }

      /* en el caso en que no se pueda asignar memoria para el nodo */
      } else {
        /* si la señal fue enviada por medio de un kill, avisa al emisor
         * con EAGAIN para que lo intente de nuevo.*/
        if ( info->si_code < 0)  {
           ret = -EAGAIN;
           goto out; /* abandonar el envío de la señal */
        }
        /* En otro caso mencionar que la señal esta pendiente pero no colocar
         * en la cola la información  */
        }
      }

 /* (3) añadir la señal en el conjunto de señales pendientes del receptor */
      /* ya se está listo para enviar la señal */
      sigaddset  ( &t->signal,  sig );
      /* si la señal no esta bloqueada  el flag sigpending se pone a uno  */
      if (!sigismember ( &t ->blocked, sig ) ) {
```

```
/* (4) poner el flag señales pendientes a uno */

      t ->sigpending = 1;

      /* código para varias cpu's */
      #ifdef __SMP__
      spin_lock(&runqueue_lock);
      if (t->has_cpu && t->processor != smp_processor_id())
         smp_send_reschedule(t->processor);
      spin_unlock(&runqueue_lock);
      #endif /* __SMP__ */

  }

  /* punto de salto para no enviar la señal */
  out:

  spin_unlock_irqrestore(&t->sigmask_lock, flags);
  /* si el proceso estaba esperando por una señal, despertarlo para que
   * pueda manejar la señal */
  if(t->state == TASK_INTERRUPTIBLE && signal_pending(t))
     wake_up_process(t);

  /* punto de salto para no enviar la señal */
  out_nolock:

  #if DEBUG_SIG
  printk(" %d -> %d\n", signal_pending(t), ret);
  #endif

  return ret;
}
```

recalc_sigpending

Es utilizada por **group_send_sig_info** para conocer si un proceso tiene señales pendientes. Se llama cuando varían los campos "**signal**" señales pendientes o "**blocked**" señales a bloquear de un proceso. Se realiza la operación lógica **and** con los campos signal y el inverso de blocked, si hay señales pendientes se pone a uno el flag **sigpending**.

```
 static inline void recalc_sigpending ( struct task_struct *t)
 {
   unsigned long ready;
   long i;

   switch (_NSIG_WORDS) { /* para varios tamaños de palabra */
```

```
/* realizar la operación logica and entre signal y el inverso de blocked */
   default:
     for (i = _NSIG_WORDS, ready = 0; --i >= 0 ;)
       ready |= t->signal.sig[i] &~ t->blocked.sig[i];
     break;

   case 4: ready  = t->signal.sig[3] &~ t->blocked.sig[3];
     ready |= t->signal.sig[2] &~ t->blocked.sig[2];
     ready |= t->signal.sig[1] &~ t->blocked.sig[1];
     ready |= t->signal.sig[0] &~ t->blocked.sig[0];
     break;

   case 2: ready  = t->signal.sig[1] &~ t->blocked.sig[1];
     ready |= t->signal.sig[0] &~ t->blocked.sig[0];
     break;

   case 1: ready  = t->signal.sig[0] &~ t->blocked.sig[0];
   }

/* si el resultado de la operación lógica and ha dejado algún bit a uno hay señales
pendientes lo que se refleja en ready y se pone a uno el flag sigpending */
   t->sigpending = (ready != 0);
 }
```

ignored_signal

Es utilizada por **group_send_sig_info** para decidir si se envía una señal a un proceso, devuelve un cero si la señal debe ser atendida o un uno si la señal debe ser ignorada.

```
 static int  ignored_signal (int sig, struct task_struct *t)
 {
   struct signal_struct *signals;
   struct k_sigaction *ka;

 /* no ignorar la señal si el proceso está en modo traza o la señal pertenece al
conjunto de bloqueadas */
   if ((t->flags & PF_PTRACED) ||
       sigismember (&t->blocked, sig))
     return 0;

   signals = t->sig;
/* ignorar la señal si el proceso esta en estado zombi */
   if (!signals)
     return 1;
```

```
   ka = &signals->action[sig-1];
   switch ((unsigned long) ka->sa.sa_handler) {

/* la función definida por defecto SIG_DFL ignora las señales SIGCONT,
SIGWINCH, SIGCHLD y SIGUR, las otras señales las atiende  */
   case (unsigned long) SIG_DFL:
     if (sig == SIGCONT ||
         sig == SIGWINCH ||
         sig == SIGCHLD ||
         sig == SIGURG)
       break;
     return 0;

/* la función definida SIG_IGN, ignora todas las señales excepto SIGCHLD */
   case (unsigned long) SIG_IGN:
     if (sig != SIGCHLD)
       break;
/* en el caso por defecto devuelve un cero indicando que la señal no sera
 * ignorada ya que ignored_signal tiene un puntero a la función definida
 * por el usuario que atiende a la señal */
   default:
     return 0;
   }
   return 1;
 }
```

PROCEDIMIENTOS PARA LA RECEPCIÓN DE SEÑALES

Un proceso se prepara para recibir una señal con la llamada **signal**, donde se especifica el tipo de señal a recibir y la función que manejara a la señal.
Dentro del núcleo se llega a la función **do_signal**.

También un **manejador de interrupción** llama a **signal_return** quien llama a **do_signal**.
También se llama desde sys_wait4.
Todos estos casos tienen en común que quieren que el proceso en curso maneje una señal si la hay.

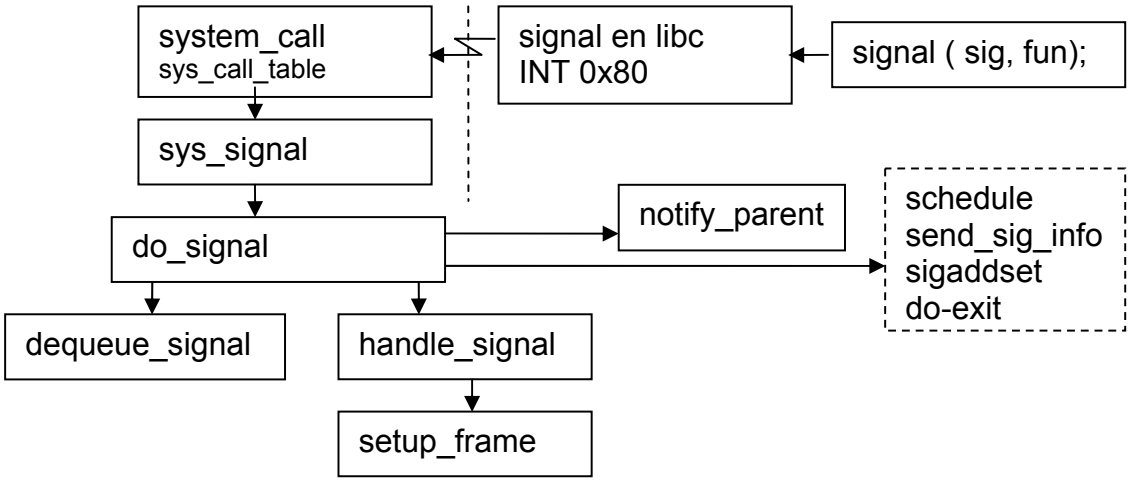
do_signal

Busca todas las señales pendientes de un proceso para atenderlas. Se encuentra definida en **arch/i386/kernel/signal.c**

Los pasos principales son:

La función do_signal, es la que para cada una de las señales pendientes, realiza los siguientes puntos:

1. Sacar de la cola de señales pendientes una señal.
2. Comprobar si el proceso está en modo traza, acción especial.
3. Obtener la función para tratar la señal.
4. Comprobar si es una señal que requiere una acción especial.
5. Si no ejecutó la función en el caso 3, se trata la señal con la función definida.
6. Volver.




```
int do_signal (struct pt_regs *regs, sigset_t *oldset)
{
  /* regs los registros de la cpu */
  /* oldset el conjunto de señales bloqueadas del proceso en curso */
  siginfo_t info;
  struct k_sigaction *ka;  //aquí irá la función definida.

  /* si llegamos desde el núcleo, salir sin hacer nada */
  if  ( ( regs->xcs & 3 ) != 3)
        return 1;

  /* si el proceso en curso tiene conjunto de señales bloqueadas, tomarlo */
  if ( !oldset )
      oldset = &current->blocked;

/* 1. Sacar de la cola de señales pendientes una señal. */
    /* bucle principal de la función, se testea si es un caso especial, se sale del
     * bucle por no haber señal que atender o bien por tener que  ejecutar la
     * función manejadora de una señal */
    for ( ; ; )  {
```

```
unsigned long signr;
spin_lock_irq ( &current->sigmask_lock );

/* saca de la cola sigqueue una señal, devuelve un cero si la cola está vacia
 * o el nº de la señal a manejar, llena la estructura info con información de
 * la señal */
signr = dequeue_signal ( &current->blocked, &info );
spin_unlock_irq ( &current->sigmask_lock );

/* si no hay señales pendientes salir, esto no ocurre en la primera ejecución
 * del bucle */
if (!signr)
   break;

/* 2. Comprobar si el proceso está en modo traza, acción especial. */
   /* si el proceso actual está en modo traza, y la señal no es SIGKILL,
    * el proceso padre, (el debugger), debe ser avisado de la llegada de la señal */
   if ((current->flags & PF_PTRACED) &&
        signr != SIGKILL) {
      /* hacer que el debugger se ejecute  */
      /* la señal que llega al proceso se comunica al padre, el debuger por medio
       * del campo exit_code, el padre la estará esperando en un wait. */
      current->exit_code = signr;
      /* se para al proceso actual */
      current->state = TASK_STOPPED;
      /* se avisa al padre enviándole la señal SIGCHLD */
      notify_parent ( current, SIGCHLD );
      /* llama a schedule para darle cpu al padre (debugger) o a otro
       * proceso prioritario */
      schedule();

      /* si el debugger cancela la señal, continuar con el bucle */
      if ( !( signr = current->exit_code ) )
            continue;
      current->exit_code = 0;

       /* si el proceso está en modo traza y ha recibido la señal SIGSTOP
        * del debugger, no tratar esta señal, continuar con el bucle */
      if (signr == SIGSTOP)
         continue;

      /* si el debugger cambió el numero de señal que se iba a tratar, llenar
       * la estructura info con la nueva información de la señal */
      if (signr != info.si_signo) {
         info.si_signo = signr;
         info.si_errno = 0;
         info.si_code = SI_USER;
         info.si_pid = current->p_pptr->pid;
         info.si_uid = current->p_pptr->uid;
      }
```

```
        /* si la nueva señal está en el conjunto de señales bloqueadas, ponerla en
         * la cola y continuar el bucle, si no seguir */
        if (sigismember(&current->blocked, signr)) {
            group_send_sig_info(signr, &info, current);
            continue;
        }
    } /* fin del if PF_TRACED */

/* 3. Obtener la función para tratar la señal. */
    /* en este punto hay una señal a tratar, el proceso no está en modo traza, o
     * si está en modo traza ha recibido la señal SIGKILL. */
    /* comienza leyendo la estructura sig->action que le dice la función y modo
     * de tratar la señal */
    ka = &current ->sig ->action [ signr-1 ]; /*Se carga en ka, la función que tratará
la señal*/

    /* tratar los casos SIG_IGN y SIG_DFL */
    /* si la función para tratar la señal es ignorarla SIG_IGN, y no es la
     * señal SIGCHLD, continuar con el bucle */
    if ( ka->sa.sa_handler == SIG_IGN ) {
        if (signr != SIGCHLD)
            continue;
        /* la función ignorar SIG_IGN para la señal SIGCHLD, es especial y
         * significa esperar por un proceso hijo sin suspenderse y quitar el
         * proceso hijo de la tabla de procesos*/
        while ( sys_wait4 ( -1, NULL, WNOHANG, NULL) > 0);
        continue;
    }
    /* pregunta si es la función por defecto SIG_DFL */
    if ( ka->sa.sa_handler == SIG_DFL )  {
         int exit_code = signr;

        /* si es el proceso es el INIT (1), independientemente de la señal recibida,
         * la acción por defecto es ignorarla no podemos matar SIGKILL al
         * proceso INIT ni por error */
        if (current->pid == 1)
            continue;

/* 4. Comprobar si es una señal que requiere una acción especial. */
        /* si no se trata del proceso INIT, ni de las funciones predefinidas hay
         * que tratar dependiendo del tipo de señal */
        switch (signr) {

        case SIGCONT: case SIGCHLD: case SIGWINCH:  /* nada */
            continue;

        case SIGTSTP: case SIGTTIN: case SIGTTOU:
            /* pregunta si el grupo de procesos al que pertenece el proceso son
             * huérfanos, no están asignados a ningún terminal tty, no hacer nada */
```

```
            if ( is_orphaned_pgrp ( current->pgrp ) )
            continue;      /* si no están huérfanos sigue el código y para al proceso */

        /* caso de recibirse la seña SIGSTOP, parar el proceso y notificar al padre */
        case SIGSTOP:
            current->state = TASK_STOPPED;
            current->exit_code = signr;
            /* si el padre no ha inhibido responder, notificar al padre */
            if ( !(current->p_pptr->sig->action[SIGCHLD-1].
                        sa.sa_flags & SA_NOCLDSTOP))
            notify_parent (current, SIGCHLD);
            /* se llama al manejador de la cpu */
            schedule();
            /* cuando se vuelve a dar cpu a este proceso se continua para atender
             * otra señal */
            continue;

        /* para otro grupo de señales la función es terminar el proceso con un exit,
         * pero para algunas primero hay que hacer un volcado de la memoria */
        case SIGQUIT: case SIGILL: case SIGTRAP:
        case SIGABRT: case SIGFPE: case SIGSEGV:
        lock_kernel();
        if (current->binfmt
            && current->binfmt->core_dump
            && current->binfmt->core_dump (signr, regs) )
             exit_code |= 0x80;
        unlock_kernel();

        /* seguir en el caso por defecto */
        /* terminar el proceso */
        default:
        lock_kernel();
        /* añadir la señal al conjunto de señales pendientes */
        sigaddset (&current->signal, signr);
        current->flags |= PF_SIGNALED;
        do_exit  (exit_code); /* termina el proceso */
        /* de do_exit no se retorna por lo que este punto nunca se alcanza */
      } /* fin del switch */
    } /* fin del if SIG_DFL */

/* 5. Tratar la señal con la función. */
    /* en este punto se ha quitado una señal de la cola de señales pendientes,
     * el proceso no está en modo traza y no es un caso especial y va a ser
     * tratada por la función definida por el usuario. */
    handle_signal ( signr, ka, &info, oldset, regs );

/* 6. Volver */
    return 1; /*indica al llamador que la señal va a ser tratada */
  } /* fin del bucle principal for */
```

```
  /* si se viene cuando se estaba tratando una llamada al sistema y se interrumpió */
  if (regs->orig_eax >= 0) {
       /* reanudar la llamada al sistema, arreglando registros,
        * no hay función para tratar la señal */
       if (regs->eax == -ERESTARTNOHAND ||
          regs->eax == -ERESTARTSYS ||
          regs->eax == -ERESTARTNOINTR) {
          regs->eax = regs->orig_eax;
          regs->eip -= 2;
       }
  }
  return 0; /* indica al llamador que no se ha tratado ninguna señal */
} /* fin do_signal */
```

dequeue_signal

Esta función quita una señal pendiente de la cola **sigqueue** y del conjunto de señales pendientes **signal** que no este bloqueada en **mask**, devuelve el número de la señal e información de la señal en la estructura **info**. Utiliza la función **kmem_cache_free** para liberar un nodo de la cola, y la función **sigdelset** para quitar la señal del conjunto de señales pendientes. Se encuentra definida en **/kernel/signal.c**

```
dequeue_signal (sigset_t *mask, siginfo_t *info)
{
  unsigned long i, *s, *m, x;
  int sig = 0;

  /* si esta en modo debugger print */
  #if DEBUG_SIG
  printk  ("SIG dequeue (%s:%d): %d ", current->comm,
         current->pid, signal_pending(current));
  #endif

  /* encontrar la primera señal que esta pendiente */
  /* para mayor rapidez, almacena el conjunto de señales pendientes en s y
   * en m el conjunto de señales bloqueadas */
  s = current->signal.sig;
  m = mask->sig;
  /*busca y coloca en sig la primera señal pendiente */
  switch (_NSIG_WORDS) {
  default:
     for (i = 0; i < _NSIG_WORDS; ++i, ++s, ++m)
        if ( (x = *s &~ *m) != 0) {      /si hay señal pendiente y no está bloqueada */
            /* x contiene el conjunto de señales pendientes no bloqueadas */
             sig = ffz (~x) + i*_NSIG_BPW + 1;   /* sig contiene la primera señal */
             break;
        }
```

```
    break;

case 2: if ( (x = s[0] &~ m[0] ) != 0)
            sig = 1;
        else if ((x = s[1] &~ m[1]) != 0)
                sig = _NSIG_BPW + 1;
             else
                break;
        sig += ffz (~x);
        break;

case 1: if ( ( x = *s &~ *m ) != 0)
            sig = ffz (~x) + 1;
        break;
}

/* si sig es distinto de cero hay señal pendiente, quitarla de la cola sigqueue*/
if (sig) {
    int reset = 1;  /* reset indica si la señal debe borrarse de la cola */

    /* si la señal no es de tiempo real  */
    if (sig < SIGRTMIN) {
        /* colocar solo el número de la señal en la estructura info */
        info->si_signo = sig;
        info->si_errno = 0;
        info->si_code = 0;
        info->si_pid = 0;
        info->si_uid = 0;
    } else {
        /* si la señal es de tiempo real extraerla de la cola */
        struct signal_queue  *q,  **pp;
        pp = &current->sigqueue;
        q = current->sigqueue;

        /* Encontrar una señal en la lista*/
        for ( ; q ; pp = &q->next, q = q->next)
            if ( q->info.si_signo == sig )
                break;
            if (q) {
            / si ha sido encontrada la señal quitarla de la lista */
                if ( ( *pp = q->next) == NULL )
                    current->sigqueue_tail = pp;
                *info = q->info;
                /* liberar el nodo que ocuapa la señal en la lista */
                kmem_cache_free (signal_queue_cachep, q );
                nr_queued_signals--; /* decrementar el número de señales en cola */

                /* mirar si la misma señal tiene otro nodo en la lista de
                 * señales pendientes. */
                q = *pp;
```

```
                while (q) {
                    /* si hay otra ocurrencia de la misma señal en la cola, reset el flag */
                    if (q->info.si_signo == sig) {
                        reset = 0;
                    break;
                    }
                   q = q->next;
                }
            } else {

                /* la señal que se busca es de tiempo real, pero no esta en la cola
                 * puede que no tuviese espacio en la cola o señal generada
                 * por un mecanismo de no tiempo real, rellenar la estructura info como
                 * si fuese una señal de no tiempo real, guardar solo el nº de la señal */
                info->si_signo = sig;
                info->si_errno = 0;
                info->si_code = 0;
                info->si_pid = 0;
                info->si_uid = 0;
            }
        }

        /* si la señal ha sido quitada de la cola para ser atendida, quitarla
         * del conjunto de señales pendientes */
        if (reset)
        sigdelset (&current->signal, sig);
        recalc_sigpending (current); /* volver a calcular el flag sigpendig */

        } else {

            /*no se ha encontrado señal en el switch, no hay señales pendientes */
            /* volver hacer una comprobación para asegurarse */
            if ( mask == &current->blocked &&
            signal_pending (current)) {
            printk ( KERN_CRIT "SIG: sigpending lied\n" );
            current->sigpending = 0;
        }
    }

  #if DEBUG_SIG
  printk(" %d -> %d\n", signal_pending(current), sig);
  #endif

  /* devolver el numero de señal quitada de la lista o cero si no se quito */
  return sig;
}
```

notify_parent

Se notifica a un proceso padre que el estado de un hijo ha cambiado, el hijo ha sido parado o matado.

1. Rellena la estructura info con información relativa a la señal.
2. Envia la señal al padre con la función group_send_sig_info.
3. Despierta a cualquier proceso que este esperando por este hijo.

```
void  notify_parent ( struct task_struct *tsk, int sig )
{
  struct siginfo info;
  int why;

  /*  almacenar en info el contexto en el que ha ocurrido la señal */
  info.si_signo = sig;
  info.si_errno = 0;
  info.si_pid = tsk->pid;
  info.si_utime = tsk->times.tms_utime;
  info.si_stime = tsk->times.tms_stime;

  why = SI_KERNEL;        /* no debería ocurrir */
  switch ( tsk->state ) {
  /* si el hijo es zombi examinar la condición de salida y rellenar  la variable why
   * con valores para hacer un volcado de memoria, o reflejar un kill o un exit */
  case TASK_ZOMBIE:   /* en el caso proceso zombi */
     if (tsk->exit_code & 0x80)
        why = CLD_DUMPED;
     else if (tsk->exit_code & 0x7f)
              why = CLD_KILLED;
           else
              why = CLD_EXITED;
     break;
   /* si el proceso fue parado por una señal de stop, why refleja este hecho */
  case TASK_STOPPED:
     why = CLD_STOPPED;
     break;

   /* en este caso hay un error why esta con SI_KERNEL, escribir el error y
    * continuar */
  default:
     printk (KERN_DEBUG  "eh? notify_parent with state %ld?\n",
         tsk->state);
     break;
  } /* fin del switch */
  info.si_code = why; /* fin de rellenar la estructura info */

  /* enviar la señal al proceso padre */
  group_send_sig_info ( sig, &info, tsk->p_pptr );
```

```
  /* despertar a cualquier proceso que este esperando por este hijo, dándole cpu */
  wake_up_interruptible ( &tsk->p_pptr->wait_chldexit );
}
```

handle_signal

Es invocado desde **do_signal**, cuando un usuario define un manejador para una señal. Comprueba si el proceso realizaba una llamada al sistema y llama **setup_frame** para crear una pila para el proceso y ejecutar la función. El código fuente se encuentra en **arch/i386/kernel/signal.c**

```
static void handle_signal ( unsigned long sig, struct k_sigaction *ka,
                            siginfo_t *info, sigset_t *oldset, struct pt_regs * regs)
{
  /* preguntamos si venimos de una llamada al sistema */
  if (regs->orig_eax >= 0) {
     /* si si, ver si estamos reiniciando la llamada al sistema */
     switch ( regs->eax ) {
        case -ERESTARTNOHAND:
           regs->eax = -EINTR;
           break;

        case -ERESTARTSYS:
           if (!(ka->sa.sa_flags & SA_RESTART)) {
              regs->eax = -EINTR;
              break;
           }
        /* seguir en este punto */
        case -ERESTARTNOINTR:
           regs->eax = regs->orig_eax;
           regs->eip -= 2;
     } /* fin del switch */
  } /* fin del if eax > 0 */

  /* crear un stack para el manejador de la señal del usuario*/
  /* si el usuario solicita información de la señal y su contexto, crear un stack
   * con setup_rt_frame si no crearlo con setup_frame, después el control pasa
   * al manejador de la señal y luego retorna al código que se estaba
   * ejecutando cuando llego la señal */
  if (ka->sa.sa_flags & SA_SIGINFO)
     setup_rt_frame ( sig, ka, info, oldset, regs );
  else
     setup_frame ( sig, ka, oldset, regs );

  /* si esta activado el flag SA_ONESHOT, (sys_signal lo tiene), el manejador
   * debe ejecutarse una sola vez y el manejador se carga con la función por
   * defecto SIG_DFL.  */
  if (ka->sa.sa_flags & SA_ONESHOT)
     ka->sa.sa_handler = SIG_DFL;

  /* El flag SA_NODEFER  indica que no otras señales se deben bloquear
   * cuando se esta ejecutando el manejador de esta señal, si el flag es
   * cero se colocan bits adicionales se añaden al conjunto de señales bloqueadas */
```

```
  if ( !( ka->sa.sa_flags & SA_NODEFER ) ) {
     spin_lock_irq (&current->sigmask_lock);
     sigorsets (&current->blocked,&current->blocked, &ka->sa.sa_mask);
     sigaddset (&current->blocked,sig);
     recalc_sigpending(current);
     spin_unlock_irq(&current->sigmask_lock);
  }
}
```

setup_frame

Esta función es llamada por **handle_signal**, para construir una pila para que el proceso guarde su estado y posteriormente pueda recuperarse, y ejecuta la función manejadora del proceso. Tiene como parámetros, **sig** la señal a tratar, ***ka** puntero a la función manejadora, ***set** el conjunto de señales pendientes, y * **regs** el conjunto de registros de la cpu.

```
static void setup_frame (int sig, struct k_sigaction *ka,
                         sigset_t *set, struct pt_regs * regs)
{
  struct sigframe *frame; /* estructura con todos los registros de la cpu */
  int err = 0;
  /* obtiene una estructrua frame con el estado de la cpu*/
  frame = get_sigframe (ka, regs, sizeof(*frame));

  /* comprueba permisos de acceso */
  if (!access_ok(VERIFY_WRITE, frame, sizeof(*frame)))
     goto give_sigsegv;

  /* escribe en el espacio de usuario la pila con el estado de los registros de la
   * cpu, el número de interrupción de la llamada al sistema en curso, el código
   * de error asociado y la mascara de señales*/
  err |= __put_user((current->exec_domain
           && current->exec_domain->signal_invmap
           && sig < 32
           ? current->exec_domain->signal_invmap[sig]
           : sig),
         &frame->sig);

  err |= setup_sigcontext(&frame->sc, &frame->fpstate, regs, set->sig[0]);

  if (_NSIG_WORDS > 1) {
     err |= __copy_to_user(frame->extramask, &set->sig[1],
                           sizeof(frame->extramask));
  }

  /* Se prepara para retornar cuando termine la función del espacio del usuario.
   * Coloca en la pila una llamada al sistema que se ejecutara cuando termine la
```

```
    * función con el fin de recuperar los datos de la pila y reinstalar la mascara de
    * señales  Si estaba previsto retornar, SA_RESTORER, usa información ya
    * situada en el espacio de usuario.  */
   if (ka->sa.sa_flags & SA_RESTORER) {
   err |= __put_user(ka->sa.sa_restorer, &frame->pretcode);
   } else { /* almacena información en el espacio de usuario para poder retornar */
      err |= __put_user(frame->retcode, &frame->pretcode);
      /* Esto hace popl %eax ; movl $,%eax ; int $0x80 */
      err |= __put_user(0xb858, (short *)(frame->retcode+0));
      err |= __put_user(__NR_sigreturn, (int *)(frame->retcode+2));
      err |= __put_user(0x80cd, (short *)(frame->retcode+6));
      }

   if (err) /* salir de la función */
   goto  give_sigsegv;

   /* preparar los registros para que se pueda ejecutar la función manejadora*/
   /* inicializa el estack pointer a la estructura frame */
   /* inicializa el contador de programa a la función manejadora */
   regs->esp = (unsigned long) frame;
   regs->eip = (unsigned long) ka->sa.sa_handler;

   set_fs(USER_DS);
   regs->xds = __USER_DS;
   regs->xes = __USER_DS;
   regs->xss = __USER_DS;
   regs->xcs = __USER_CS;
   regs->eflags &= ~TF_MASK;

   #if DEBUG_SIG
      printk("SIG deliver (%s:%d): sp=%p pc=%p ra=%p\n",
               current->comm, current->pid, frame, regs->eip,
               frame->pretcode);
   #endif

   return;

   give_sigsegv:
   /* comprueba si se trata de la señal SIGSEGV, violación de segmento */
   if (sig == SIGSEGV)  ka->sa.sa_handler = SIG_DFL;
      force_sig(SIGSEGV, current);
}
```

do_sigaction

Su código fuente se encuentra en el fichero **kernel/signal.c**
Se ejecuta en respuesta a la llamada al sistema **sys_sigaction**, la versión POSIX de la llamada **action,** análogamente se prepara para **recibir una señal** y asocia una **función manejadora** para cuando llegue la **señal.**
La función **sigactión** no tiene instalado el flag **SA_ONESHOT**, asi que **no** se necesita **reistalar** la función manejadora cada vez que se recibe la señal como en signal.

```
int do_sigaction ( int sig, const struct k_sigaction *act, struct k_sigaction *oact)
{
  struct k_sigaction *k;

  /* comprueba el rango de la señal  y que el proceso no esta tratando de
   * asociar una acción a las señales SIGKILL o SIGSTOP */
  if ( sig < 1 ||  sig > _NSIG  || (act && (sig == SIGKILL || sig == SIGSTOP)))
     return -EINVAL;

  spin_lock_irq ( &current->sigmask_lock );
  /* toma la function manejadora del usuario a través de la estructura sig de la
   * tabla de procesos */
  k = &current->sig->action [sig-1];

  /* sigaction puede retornar la acción a través de oact, cuando se guardan
   * en el estack manejadores para ser recuperados */
  if (oact) *oact = *k;

  /* si el proceso de usuario ha asociado una acción para la señal  */
  if (act) {
     *k = *act;
     /* borra del conjunto de señales bloqueadas a SIGKILL y SIGSTOP
      * para asegurarse que estas señales no pueden ser bloqueadas o
      * asignarles un manejador*/
     sigdelsetmask (&k->sa.sa_mask, sigmask (SIGKILL) | sigmask (SIGSTOP) );

     /* POSIX 3.3.1.3: Colocar el manejador por defecto SIG_IGN para una
      * señal que esta pendiente, causa que la señal sea ignorada independiente
      * de si está o no bloqueada */

     /* Colocar el manejador por defecto SIG_DFL para una señal que
      * esta pendiente y cuya acción  por defecto es ignorar la señal (por
      * ejemplo, SIG_CHLD), causa que la señal sea ignorada independientemente
      * si está o no bloqueada */

     /* Note el comportamiento tonto de SIGCHLD: SIG_IGN significa que la señal
      * no sea ignorada, pero hace que el hijo sea quitado, mientras que la función
      * SIG_DFL según la norma POSIX, fuerza que la señal sea ignorada */
     /* no se quita de la cola si las señales son SIGCONT, SIGCHLD, o
      * SIGWINCH  o la función es SIG_DFL */
```

```
        if (k->sa.sa_handler == SIG_IGN || (k->sa.sa_handler == SIG_DFL
           && (sig == SIGCONT || sig == SIGCHLD || sig == SIGWINCH))) {

           /*quitar de la cola cualquier señal pendiente de tiempo real */
           if ( sig >= SIGRTMIN && sigismember (&current->signal, sig) ) {
              struct signal_queue *q, **pp;
              pp = &current->sigqueue;
              q = current->sigqueue;
              while (q) {
                 if (q->info.si_signo != sig)
                    pp = &q->next;
                 else {
                    *pp = q->next;
                    /* liberar el espacio ocupado por el nodo de señal pendiente */
                    kmem_cache_free (signal_queue_cachep, q);
                    nr_queued_signals --;
                 }
                 q = *pp;
              } /* fin del while */
           } /* fin del if sig*/

        /* borrar la señal del conjunto de señales pendientes */
        sigdelset (&current->signal, sig);
        recalc_sigpending(current);
        }

     }/* fin del if act*/
     spin_unlock_irq(&current->sigmask_lock);

     return 0;
}
```