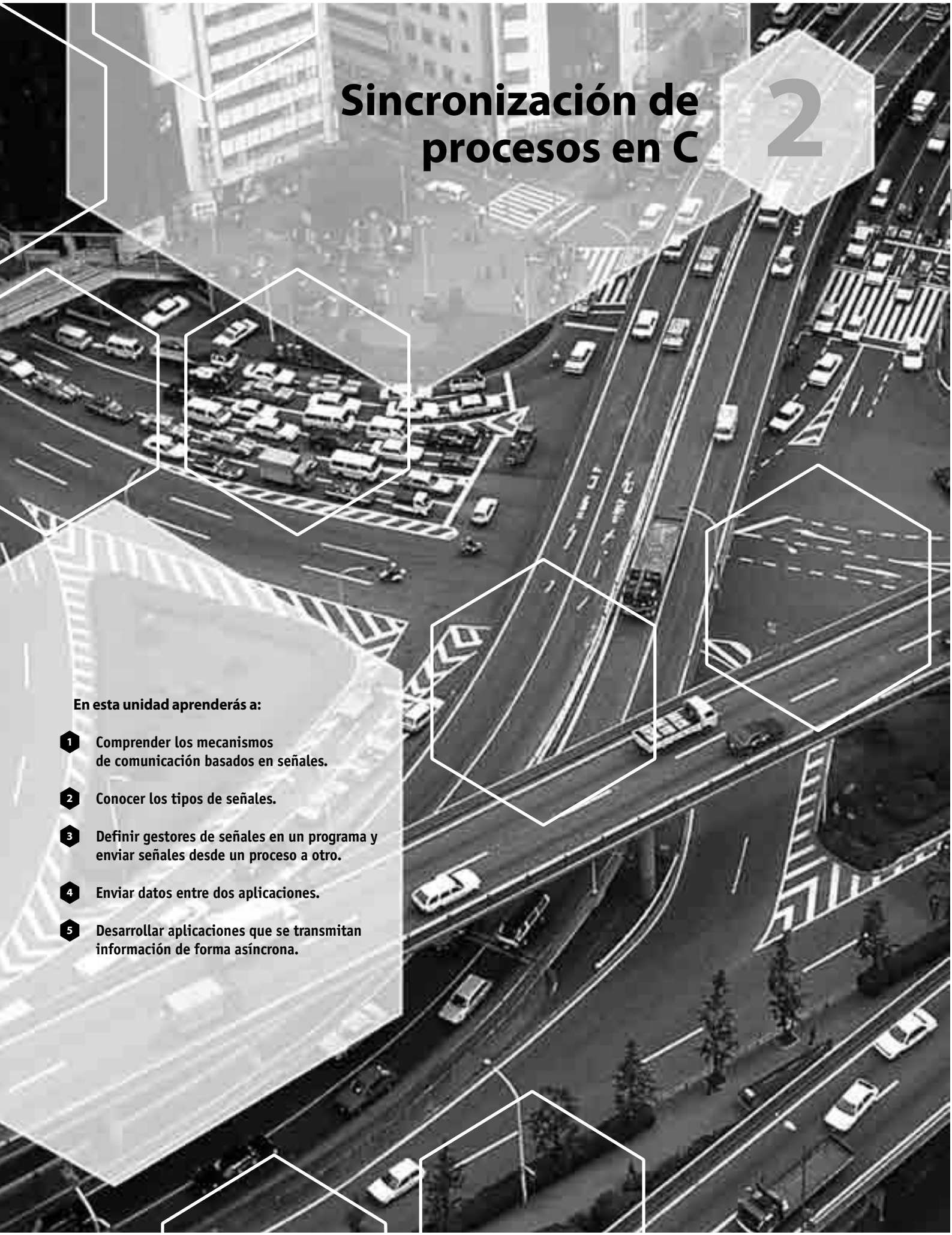
Sincronización de procesos en C

2



En esta unidad aprenderás a:

1. **Comprender los mecanismos de comunicación basados en señales.**
2. **Conocer los tipos de señales.**
3. **Definir gestores de señales en un programa y enviar señales desde un proceso a otro.**
4. **Enviar datos entre dos aplicaciones.**
5. **Desarrollar aplicaciones que se transmitan información de forma asíncrona.**

2.1 Concepto de señal

En un sistema, los procesos que se ejecutan simultáneamente interactúan entre sí. Esta interacción se produce incluso en el caso de los procesos independientes, esto es, los que no necesitan cooperar para completar sus tareas. Esto ocurre cuando varios procesos quieren acceder a los mismos recursos del sistema operativo, por ejemplo un dispositivo de entrada y salida. Para resolver esta situación, el sistema operativo dispone de un gestor de procesos que determina el orden de acceso a esos recursos; adicionalmente, cuenta con un mecanismo para poder enviar mensajes a los procesos. La necesidad de intercambiar información entre procesos es muy clara en las arquitecturas colaborativas, por ejemplo, la llamada de *cliente-servidor*, donde un proceso suministra información o servicios a otros procesos.

Las señales pueden considerarse un tipo de mensajes, aunque, si se comparan con otros medios de comunicación de procesos (*sockets*, *pipes*, etc.), resultan un mecanismo más pobre, ya que no permiten transmitir datos. A pesar de ello, es importante conocerlas y saber manejarlas, pues proporcionan dos servicios fundamentales:

- La defensa del proceso establecido frente a incidencias comunicadas por el *kernel*, que envía señales al proceso cuando se ha producido alguna eventualidad. Si éstas no son gestionadas (bien ignoradas, bien capturadas) por el proceso al que van dirigidas, dan lugar a su inmediata conclusión, lo que puede redundar en una pérdida irrecuperable de datos. Es el caso del proceso que se da cuando se están guardando datos en un fichero y, al mismo tiempo, se recibe una señal del *kernel*: la conclusión del programa debe aplazarse hasta la finalización de la transferencia para que no se produzca una pérdida de datos.

- El mecanismo de comunicación entre dos procesos. Dicho mecanismo resulta útil y sencillo para avisar a un proceso de la aparición de eventos excepcionales, si bien no debe emplearse como forma habitual de comunicación entre procesos. Por ejemplo, puede utilizarse en el caso de que el usuario desee interrumpir el proceso de impresión de un documento porque se ha dado cuenta de que ha mandado imprimir una versión antigua; o en el caso de un proceso principal que recibe una señal de conclusión, ya que puede enviar una señal a los procesos que dependen de él, a fin de que éstos puedan actualizar sus datos y escribir en disco, antes de finalizar también ellos su ejecución. Estas necesidades de comunicación con y entre procesos quedan plenamente satisfechas mediante el empleo de las señales, al menos en aquellas en que el proceso pide al *kernel* que envíe una señal a otro proceso.

Así pues, el uso de señales es un método sencillo de aviso de incidencias (por circunstancias del propio proceso o por la intervención de otro proceso) entre el *kernel* y los procesos, y su aparición en el tiempo no puede ser prefijada, ya que suelen referirse a hechos imprevistos. Dicha comunicación de incidencias debe realizarse de forma estructurada y bien definida. Las señales pueden aparecer en cualquier instante, por lo que los procesos no pueden limitarse a verificar una variable para comprobar si ha llegado una señal, sino que deben lanzar una rutina de tratamiento de la señal con la que se gestione de manera automática su recepción en el momento en que aparezca.

2.2 Definición, generación y tratamiento de señales

Las señales son interrupciones de software que pueden enviarse a un proceso para proporcionar un método con el que tratar eventos asíncronos. Las interrupciones de software enviadas a un proceso pueden ser generadas por el *kernel* del sistema operativo, por el usuario desde la línea de comandos, o por cualquier otro proceso que conozca el *pid* del proceso.

Las señales generadas por el *kernel* se producen cuando se detecta un error de software o de hardware en la ejecución del proceso. Un error de software sobreviene, por ejemplo, cuando un proceso intenta acceder a una zona de memoria que no le corresponde; en ese instante, se genera una señal de violación de segmentación que puede causar la finalización de ese proceso. Asimismo, las señales se emplean para informar de ciertos fallos detectados por el hardware, como divisiones por cero, referencias inválidas de memoria, etc.

En el caso de las señales generadas por el usuario, éstas se pueden producir al pulsar ciertas teclas (por ejemplo, **Supr**) en un terminal, y también a través de la línea de comando: el comando *kill* permite que el usuario envíe señales a un proceso o grupo de procesos.

Para la generación de señales en un proceso se utilizan las funciones *kill* y *killpg*, con las que un proceso puede enviar señales a otro proceso o grupo de procesos. Estas funciones son equivalentes al comando *kill* y se verán con más detalle a lo largo de esta unidad.

Como las señales pueden aparecer en cualquier instante, el proceso debe indicar al *kernel* qué es lo que ha de hacer cuando recibe una señal determinada. El *kernel* puede actuar de tres formas diferentes: ignorar la señal, capturarla o aplicar la rutina por defecto.

A. Señales ignoradas

Mediante el establecimiento de la constante *SIG_IGN*, el proceso ignorará la recepción de la señal. Esto implica que el proceso no realizará ninguna acción al recibirla; su ejecución continuará exactamente en el mismo punto donde se encontraba. Cuando se ignora una señal, suele decirse que el proceso se hace inmune a ella. Esta inmunidad no significa que el proceso no quede afectado por la causa que generó la señal, sino que simplemente la ignora.

Cuando se ignoran las señales generadas por fallos de hardware, el comportamiento del proceso es incierto. Por ejemplo, si una sentencia del código del proceso está produciendo una división por cero, la sentencia dará como resultado algo que no tendrá sentido y, sin embargo, el proceso continuará ejecutándose. Por otra parte, un proceso que ignorara todas las señales no podría ser gestionado por el administrador de la máquina, ya que ésta no respondería a sus instrucciones: por ejemplo, si la máquina se instalara en un bucle infinito y su ejecución no terminara nunca, el administrador no

podría concluir su ejecución con el comando *kill-9*. Por este motivo, existen ciertas señales que no pueden ser ignoradas, a fin de permitir al administrador de la máquina detener o eliminar cualquier proceso.

B. Señales capturadas

El programador del proceso puede preparar una rutina que se ejecute al recibirse una señal determinada. El código de la rutina debe desarrollarse teniendo en cuenta que esta rutina no se va a ejecutar en el momento deseado por el programador, sino en cualquier instante en que se produzca un evento. El código deberá, asimismo, responder a lo que ocurre en el sistema. Por ejemplo, en el caso de que se haya producido una división por cero, debe ser capaz de conocer cuál es la variable cuyo valor es cero, cambiar su valor por un valor por defecto que no sea cero, y seguir ejecutando el proceso.

Una vez programada, se debe especificar a qué señal responderá la rutina propia que gestiona una señal concreta. Para ello, existe una función que asigna la dirección de la rutina a la señal. El *kernel* llamará a esa función cada vez que se reciba la señal. La acción o acciones a las que dé lugar dicha rutina dependerán del programador. Cuando se termina la ejecución de esta rutina, el desarrollo del proceso puede continuar en el punto en que se había interrumpido, finalizar o volver a un punto del programa establecido con anterioridad por el programador. En la figura 2.1 se representan de manera gráfica estos tres comportamientos.

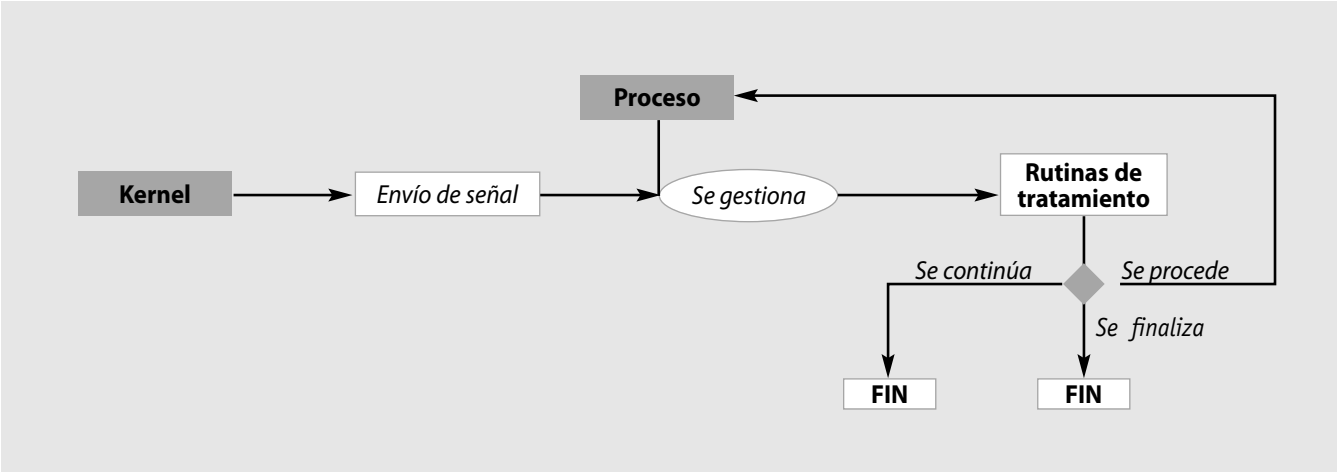


Fig. 2.1. *Acciones que se pueden producir al recibir una señal y capturarla.*

C. Señales tratadas por defecto

Mediante la especificación de la constante *SIG_DFL*, el *kernel* llama a una función determinada cada vez que la señal reaparece. Cada señal tiene asignada una acción por

defecto (representadas en la figura 2.2), que suelen ser: *exit*, por la que el proceso receptor de la señal finaliza; *core*, por la que el proceso receptor finaliza y deja un fichero en el directorio actual con una imagen de memoria del contexto del proceso, y *stop*, por la que el proceso receptor se detiene.

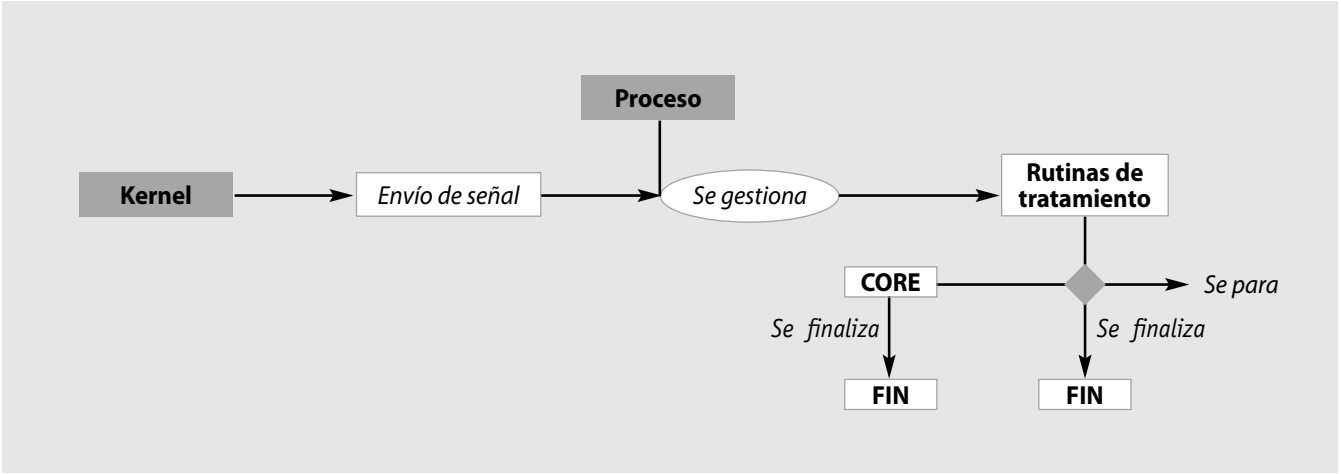
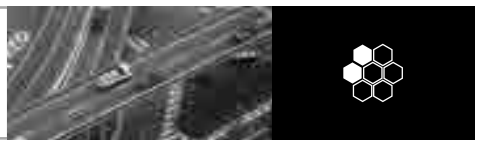


Fig. 2.2. *Acciones que se pueden producir al recibir una señal y tratarla por defecto.*

2.3 Tipos de señales en Linux

Cada señal posee un nombre que comienza por *SIG*, mientras que el resto de los caracteres se relacionan con el tipo de evento que representa. Asimismo, cada señal lleva asociado un número entero positivo, que es el que intercambia con el proceso cuando éste recibe la señal. Ninguna señal lleva asociado el número 0, ya que se emplea con la función *kill* en algún caso especial, que en Linux es una comprobación de errores: por ejemplo, en la comprobación del *pid* de un proceso. Las señales que pueden ser manejadas por el sistema, junto con sus nombres, se definen en el fichero de cabecera </linux/signal.h>, que a su vez se halla incluido en <signal.h>.

Tipo	N.º	Significado	Acción por defecto
SIGHUP	1	**Colgado**. Se envía al controlador de terminales cuando se detecta la desconexión de algún terminal, o cuando el proceso principal de un grupo termina. En este caso, se envía a todos los procesos miembros del grupo. Se suele emplear para notificar a procesos demonio que deben realizar una relectura de sus ficheros de configuración; se emplea esta señal, ya que este tipo de procesos no suele tener asociado ningún terminal, y no es probable que reciban esta señal por ningún otro motivo.	Fin del proceso que la recibe
SIGINT	2	**Interrupción**. Se envía a todos los procesos asociados con un terminal cuando se pulsa la tecla de interrupción (**Ctrl + C**). Se suele emplear para terminar la ejecución de programas que están generando mucha información no deseada por pantalla.	Fin del proceso que la recibe



Tipo	N.º	Significado	Acción por defecto
SIGQUIT	3	**Abandonar**. Se envía a todos los procesos asociados con un terminal cuando se pulsa la tecla de salida. Es muy similar a *SIGINT*, ya que su acción por defecto es la misma, pero, además, genera un fichero *core*.	Fin del proceso que la recibe y generación de fichero *core*
SIGILL	4	**Instrucción ilegal**. Se envía a un proceso cuando el hardware detecta que éste ha ejecutado una instrucción ilegal.	Fin del proceso que la recibe y generación de fichero *core*
SIGTRAP	5	**Punto de ruptura**. Se genera por un punto de ruptura cuando se está depurando o ejecutando un proceso paso a paso.	Fin del proceso que la recibe y generación de fichero *core*
SIGABRT *SIGIOT*	6	**Abortar**. *SIGABRT* se genera cuando se produce la terminación anormal de un proceso por acción de la función *abort*. *SIGIOT* se envía a un proceso cuando se produce un fallo de hardware.	Fin del proceso que la recibe y generación de fichero *core*
SIGBUS	7	**Error de bus**. Se genera cuando se produce un error de acceso a memoria, por ejemplo, cuando se intenta acceder a una dirección que físicamente no existe. También se produce cuando se accede a una dirección impar, violando así las reglas de alineación que impone el hardware.	Fin del proceso que la recibe y generación de fichero *core*
SIGPFE	8	**Excepción de coma flotante**. Se envía a un proceso cuando el hardware detecta un error de coma flotante, como un desbordamiento o un formato desconocido	Fin del proceso que la recibe y generación de fichero *core*
SIGKILL	9	**Eliminar**. Es una de las señales que no puede ser ignorada. Proporciona al administrador del sistema un medio seguro de terminar con cualquier proceso.	Fin del proceso que la recibe y generación de fichero *core*
SIGUSR1	10	**Señal de usuario 1**. Es una de las dos señales reservadas al usuario para su empleo en programas de aplicación. El significado es el que le quiera conferir el programador.	Fin del proceso que la recibe
SIGSEGV	11	**Fallo de segmentación**. Se envía a un proceso cuando éste intenta acceder a datos que están fuera de su segmento de datos.	Fin del proceso que la recibe y generación de fichero *core*
SIGUSR2	12	**Señal de usuario 2**. Es la segunda de las señales reservadas al usuario. Su acción y significado son similares a los de *SIGUSR1*.	Fin del proceso que la recibe
SIGPIPE	13	***Pipe*** o ***socket*** **roto**. Se envía a un proceso cuando éste intenta escribir en un *pipe* del que no hay nadie leyendo. También se genera cuando un proceso intenta escribir en un *socket* del que no hay nadie leyendo.	Fin del proceso que la recibe
SIGALRM	14	**Alarma de reloj**. Se envía a un proceso cuando se sobrepasa el tiempo (en segundos) fijado mediante la función *alarm*.	Fin del proceso que la recibe
SIGTERM	15	**Terminar**. Se genera por defecto por la función *kill*. Es menos drástica que *SIGKILL* y puede ser ignorada. Se envía a todos los procesos durante la parada del sistema o *shutdown*.	Fin del proceso que la recibe
SIGSTKFLT	16	**Fallo pila coprocesador**. Se genera por un fallo de pila en el coprocesador.	Fin del proceso que la recibe
SIGCHLD	17	**Proceso hijo detenido** o **terminado**. Se genera cuando un proceso hijo ha parado de ejecutarse o ha terminado.	Ignorar la señal
SIGCONT	18	**Continuar**. Indica que la ejecución de un proceso debe continuar si se había detenido; en cualquier otro caso, la señal se ignora.	Ignorar la señal
SIGSTOP	19	**Detener**. Indica a un proceso que debe detenerse. Es otra de las señales que no puede ser ignorada.	Proceso detenido
SIGSTP	20	**Detener interactiva**. Señal de stop interactiva. Se genera al pulsar la tecla detener, que suele ser **Ctrl + Z** y es enviada a todos los procesos del grupo de procesos que están listos para ser ejecutados.	Proceso detenido
SIGTTIN	21	**Lectura de terminal**. Lectura de un terminal de control por un miembro de un grupo de procesos que se encuentra en *background*.	Detener el proceso
SIGTTOU	22	**Escritura en terminal**. Escritura en un terminal de control por un miembro de un grupo de procesos en *background*.	Detener el proceso
SIGURG	23	**E/S dato urgente**. Se genera cuando llega un dato urgente a través de un canal de entrada/salida.	Ignorar la señal

Tipo	N.º	Significado	Acción por defecto
SIGXCPU	24	**Tiempo CPU excedido**. Permite indicar al proceso que la recibe que ha superado el tiempo de CPU que tenía asignado.	Fin del proceso que la recibe
SIGXFSZ	25	**Excedido límite tamaño fichero**. Permite indicar al proceso que la recibe que ha superado el tamaño máximo de fichero que puede manejar.	Fin del proceso que la recibe
SIGVTALRM	26	**Alarma generada por temporizador virtual**. Se genera por un temporizador en tiempo virtual, cuando éste ha llegado a cero.	Fin del proceso que la recibe
SIGPROF	27	**Temporizador expirado**. Se genera por un temporizador tanto en tiempo real como en virtual, cuando éste ha llegado a cero.	Fin del proceso que la recibe
SIGWINCH	28	**Cambio tamaño ventana**. Se genera por el comando *ioctl* y permite cambiar el tamaño de la ventana cuando se está trabajando con interfaces gráficas.	Ignorar la señal
SIGIO	29	**E/S asíncrona**. Esta señal indica un evento de entrada/salida asíncrono. Normalmente se emplea para indicar que un dispositivo está listo para una operación de entrada/salida.	Ignorar la señal

Tabla 2.1. *Resumen de las señales con sus características más destacables.*

2.4 Señales suspendidas y bloqueadas

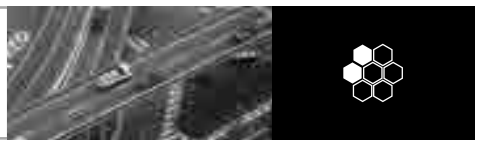
Analizaremos ahora lo que ocurre con las señales desde que se generan hasta que son recibidas por un proceso, así como la capacidad de los procesos para inhibir la recepción de ciertas señales.

Se considera que una señal se ha entregado a un proceso cuando éste realiza la acción asociada a ella. Durante el tiempo que transcurre entre la generación de la señal y su entrega a un proceso, se suele decir que la señal está *pendiente*. En consecuencia, es preciso disponer de algún mecanismo —en este caso una función— que permita averiguar si en un determinado instante hay señales pendientes, que aún no se han entregado al proceso correspondiente.

Por otro lado, hay que apuntar que los procesos tienen la posibilidad de bloquear la recepción de una determinada señal. En el transcurso de un proceso, si el programador ha definido con anterioridad que una señal concreta sea capturada o tratada por defecto, se supone que, en principio, se desea responder a dicha señal. Por otra parte, si el programador bloquea una señal cuya acción asociada es la captura de la señal o su tratamiento por defecto, la acción queda pendiente. Es decir, el *kernel* genera la señal y la envía al proceso que, en principio, tiene definido cómo responder a ella; al descubrir que la señal está bloqueada, el *kernel* la guarda hasta que el proceso o bien la desbloquea, o bien modifica la acción que lleva asociada, pasando entonces a ignorarla. El *kernel* determina qué debe hacerse con una señal bloqueada cuando se entrega, y no cuando se genera, lo que permite que los procesos puedan modificar la acción que van a realizar ante ella antes de su recepción.

A. Función *sigprocmask*

Cada proceso dispone de una máscara de señales que le permite definir qué señales se encuentran bloqueadas. Puede decirse que esta máscara está formada por un conjunto



de bits, cada uno de los cuales se halla asociado a una posible señal. Si el bit tiene valor 1 para una determinada señal, por ejemplo, significa que dicha señal está bloqueada. Para que la máscara represente todas las señales, se utiliza un número entero sin signo; cada uno de sus bits indica, según la posición que ocupa, el estado de cada señal.

Existen varias funciones relacionadas con el establecimiento y la comprobación de la máscara de señales de un proceso. Estas funciones son *sigsuspend*, *sigsetmask*, *sigblock*, *siggetmask* y *sigmask*. No analizaremos la sintaxis ni la utilidad de cada una de ellas, ya que su funcionalidad se puede resumir en la función *sigprocmask*, que es la que realmente se emplea en las últimas versiones de Linux, pese a lo cual todas las demás también pueden utilizarse a criterio del programador.

La función *sigprocmask* se emplea para modificar la lista de señales bloqueadas en un momento dado. La sintaxis de la función es:

```
include <signal.h>
int sigprocmask (int how, const sigset_t *set, sigset_t
*oldset);
```

donde el entero que retorna esta función es 0, cuando se ejecuta con éxito, y –1, en caso de error.

El comportamiento de esta función es distinto según cuál sea el valor de *how*:

- *SIG_BLOCK*: el conjunto de señales bloqueadas es la unión del actual y el del argumento *set*.
- *SIG_UNBLOCK*: las señales incluidas en *set* se eliminan del conjunto de señales bloqueadas en la actualidad. Se considera permitido el intento de desbloquear una señal que no se halla bloqueada.
- *SIG_SETMASK*: el conjunto de señales bloqueadas pasa a ser el contenido en el argumento *set*.

Si el contenido del argumento *oldset* no es nulo, el valor previo de la máscara de señales se almacena en *oldset*.

Mediante esta función se puede bloquear la señal *SIGPIPE* a través de la llamada:

```
sigset_t  *antiguo = 0;
.........
sigprocmask (SIG_BLOCK, 8192, antiguo );
```

donde *8192* es el número entero que, en binario, posee un 1 en la posición 13, siendo 13 el número entero asociado a *SIGPIPE*. Es necesario recordar que existen dos señales que no es posible bloquear: *SIGKILL* y *SIGSTOP*. Cualquier intento de bloqueo de estas señales, simplemente, se ignorará.

B. Función *sigpending*

La función *sigpending* permite conocer qué señales se encuentran pendientes, es decir, están bloqueadas y se han generado para un proceso. Su sintaxis es la siguiente:

```
include <signal.h>
int sigpending (sigset_t *set);
```

Las señales pendientes se almacenan en *set*, que es una máscara de formato similar al visto con anterioridad.

Pueden plantearse algunas preguntas relacionadas con las señales bloqueadas:

- ¿Qué ocurre cuando una señal bloqueada por un proceso es generada para/por ese proceso una o varias veces antes de que el proceso desbloquee la señal?

Si el sistema entregara las señales a un proceso tantas veces como son generadas, se diría que las señales se ponen en cola; no obstante, en la mayoría de los sistemas Unix las señales no se comportan así, pues el *kernel* entrega la señal una sola vez.

- ¿Qué ocurre si las señales que están esperando para ser entregadas a un proceso son diferentes?

En la mayoría de los sistemas Unix, parece imperar el criterio de que cierto tipo de señales, por ejemplo, las relacionadas con el estado actual del proceso, como *SIGSEGV*, se entregarán antes que otras.

La máscara de señales bloqueadas viene definida por el programador, mientras que la máscara que devuelve la función *sigpending* representa las señales que, habiéndose definido como bloqueadas, se hallan pendientes; en este último caso, es el *kernel* el que indica de qué señales se trata. La función que devuelve las señales bloqueadas es *sigprocmask*, mientras que *sigpending* devuelve las pendientes.

La función *sigpending* se podría utilizar para conocer si existe alguna señal pendiente referente a *SIGPFE*, y, si es así, seguir bloqueando dicha señal y enviar un mensaje al usuario que le advierta de que ha ocurrido un error en la ejecución; de que, aunque el proceso no se va a cancelar, puede que los resultados no sean satisfactorios, y de que, en consecuencia, termine el proceso, si así lo desea:

```
/* Definimos la máscara para la señal SIGFPE. Como es la
señal 8, el bit 8 debe estar a 1, que en hexadecimal es
0080 */
#define mascara_SIGFPE 0x0080

long mascara_original;
sigset_t senales_suspendidas;

/* Máscara que bloquea la señal SIGINT */
sigsetmask(sigmask(SIGINT) );

.........................................

/* Inclusión de SIGFPE */
mascara_original = sigblok( sigmask(SIGFPE) );

.........................................
```

```
/* Cuando se termina el conjunto de divisiones, se verifi-
ca si hay alguna señal pendiente de SIGPFE; si es así, no
desbloqueamos la señal de error de coma flotante y restau-
ramos la máscara que había antes de incluirla */

sigpending( &senales_suspendidas );

/* Para ver el valor del bit 8, se realiza el AND con los
bits de la máscara de SIGFPE; si este AND tiene un 1 en la
octava posición es que está pendiente. Al hacer el XOR con
la máscara, el 1 se convierte en 0 y todos los bits que-
dan a cero, con lo que queda if(0) y no se restaura la más-
cara */

if(((senales_suspendidas & mascara_SIGFPE)^ mascara_SIGF-
PE))

      sigsetmask( mascara_original );

.........................................
```

2.5 Gestores de señales

La consecuencia de enviar una señal a un proceso, si éste no está preparado para aceptar dicha señal, es la finalización de su ejecución; en la terminología más habitual, se dice que el proceso "muere". Si el proceso dispone de un gestor de señales, el proceso responderá a la señal ejecutando la rutina asignada por el programador a las señales. Esta rutina podrá ser desarrollada por el propio programador, o bien será la asociada por defecto a la señal.

El gestor de señales anuncia que el proceso está preparado para aceptar cierto tipo de señales y proporciona la dirección de la rutina de tratamiento de señales, en caso de ser ésta de rango especial. Se dice entonces que el proceso está preparado para capturar esas señales.

Se asigna una señal a su rutina mediante una llamada a una función de Linux. Si se genera una señal de carácter relevante, por ejemplo, la pulsación de la tecla **Supr**, se realiza una llamada al gestor de señales y el proceso interrumpe su ejecución. El gestor puede estar funcionando el tiempo que sea necesario, y realizar o responder a cualquier llamada del sistema. En la práctica, el tiempo de funcionamiento suele ser pequeño; cuando finaliza, restaura el estado del proceso, que continúa su ejecución en el punto en que había sido interrumpido. Después de capturar una señal se debe restablecer el gestor de señales mediante otra llamada a la función que asigna la señal al procedimiento (si bien algunas señales se restablecen por sí mismas).

Si otra señal del mismo tipo llega antes de que el gestor de señales se haya restablecido, se ejecuta la acción por defecto de la señal recibida, por lo general, la finalización del proceso. En vez de proporcionar una función especial para capturar la señal,

es posible, como ya se ha dicho, ignorar la señal o restaurar la acción por defecto mientras ésta se está gestionando.

Un problema habitual en el uso de gestores de señales es que la rutina de tratamiento de la señal no conozca el estado del programa cuando aparece la señal. El programa puede estar solicitando memoria mediante *malloc*, liberándola mediante *free*, realizando una llamada a una función que modifique el valor de una variable estática o de una variable global externa, o esperando la conclusión de un proceso hijo mediante *wait*. En estos casos, el programador debe poner especial cuidado en la rutina de tratamiento: por ejemplo, si se va a modificar una variable global dentro del procedimiento de atención a la señal (bien directamente, bien porque se llame a una función que lo modifique), se debe guardar en una variable temporal el valor original y, al finalizar la rutina, restaurar dicho valor. Un ejemplo clásico de esta situación aparece al gestionar la señal *SIGCHLD*, ya que por regla general la rutina de atención utilizará alguna de las funciones *wait,* y esta llamada producirá una modificación de la variable externa *errno*, que se encuentra definida en <*errno.h*>.

Cuando se trabaja con procesos hijo, el gestor de señales puede interferir en el uso de la función *wait*, porque cuando el proceso padre está esperando la terminación del proceso hijo, si el usuario envía una señal a ambos procesos, la gestión de ésta saca al proceso padre de su estado de espera. Es decir, mientras el padre espera, las señales deben ser gestionadas en el proceso hijo; por tanto, deben activarse los gestores de señales en el padre para que las ignore.

A. Función *signal*

La función *signal* es el gestor de señales por excelencia. Permite especificar la acción que debe realizarse cuando un proceso recibe una señal y lo prepara para recibir cierto tipo de señales. Esto supone que será preciso añadir una llamada a *signal* para cada tipo de señal que se desee que el proceso reciba sin que cause su finalización. Su definición es:

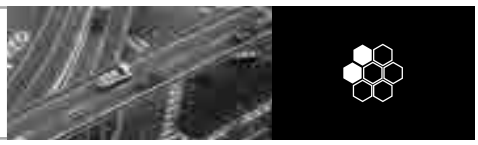
```
include <signal.h>
void (*signal (int signum, void (*func) (int))) (int);
```

La definición de la función en ANSI C establece que ésta precisa dos parámetros —el número de la señal y el tipo de acción que debe realizar— y devuelve un puntero a una función que no restablece nada.

Mediante una llamada con la función *signal* se establece un gestor de señales para la señal *signum* (número de la señal). *Func* permite elegir una de las tres acciones que se pueden realizar cuando se recibe la señal:

- *SIG_IGN*: ignora la señal.
- *SIG_DFL*: devuelve a la señal su comportamiento por defecto.
- La dirección de la función especificada por el usuario. En este caso, la definición de la función será:

```
include <signal.h>
void func (int sig);
```



donde *sig* es el número de la señal cuya acción se desea programar. Después de ser programada, la función deberá contener el código correspondiente para que la acción sea la deseada.

La llamada a *signal* devuelve, si todo va bien, el valor que tenía *func*, es decir, la disposición previa de la señal. Ésta sirve para restaurar dicho valor en cualquier momento; dicho de otro modo, la función *signal* devuelve un puntero a una función. En cambio, si se produce algún error, la función *signal* devuelve *SIG_ERR;* el código del error resultante se encuentra en la variable externa *errno*.

Como argumento *func* se pueden utilizar dos valores especiales: *SIG_DFL*, que hace referencia a la función que maneja las señales por defecto, y *SIG_IGN*, que ignora la señal recibida. En el fichero <signal.h> se puede encontrar una definición especial de las constantes *SIG_DFL*, *SIG_IGN* y *SIG_ERR*, de forma que el argumento *func*, en la llamada a *signal*, puede sustituirse por un número, 0, 1 y –1, respectivamente. La definición de estos valores enteros se realiza mediante un *cast* del siguiente modo:

```
#define SIG_DFL    ((void (*) () 0)
/* mantiene la acción por defecto */
#define SIG_IGN    ((void (*) () 1)
/* ignora la señal */
#define SIG_ERR    ((void (*) () -1)
/* error devuelto por la señal */
```

Cuando una señal llega a un proceso y es capturada, suceden dos cosas, en el orden en que se exponen: en primer lugar, se restablece a la señal su acción por defecto (generalmente, finalizar el proceso), y en segundo lugar, se llama a la función programada al efecto con un número entero como argumento, que coincide con el número asociado con la señal que se está capturando. Cuando la función retorna, el proceso continúa ejecutándose en el punto donde fue interrumpido.

B. Función *sigaction*

Se trata de otro gestor de señales. La función *sigaction* permite modificar o examinar la acción asociada con una señal determinada. Resulta de gran utilidad, puesto que permite averiguar la disposición respecto a una determinada señal sin modificarla: por ejemplo, si una señal va a ser ignorada, puede programarse una acción para capturarla, si no está siendo ignorada en un determinado instante, lo que resulta de gran utilidad en los procesos interactivos. La función *sigaction* sustituye a la función *signal* de las primeras versiones de Unix. Su definición es la siguiente:

```
include <signal.h>
int sigaction (int signum, const struct sigaction *act,
      struct sigaction *oldact);
```

donde el argumento *signum* es el número de la señal cuya acción se desea examinar o modificar; el puntero **act* representa la función con la que se desea gestionar la señal, y el puntero **oldact* representa la función que estaba gestionando la señal.

Si el puntero **act* no es nulo, entonces se está modificando la acción, que pasa a ser la contenida en la dirección de **act*. Si el puntero **oldact* no es nulo, la acción que va a ser modificada se almacena en **oldact*.

La estructura empleada por esta función es:

```
struct sigaction
{
      void (*sa_handler)();
/* dirección del gestor de señales o SIG_IGN, SIG_DFL,
según corresponda */
      sigset_t    sa_mask;
      /* señales adicionales al bloque */
      int sa_flags;
      /* flags opcionales para la gestión de cada señal*/
};
```

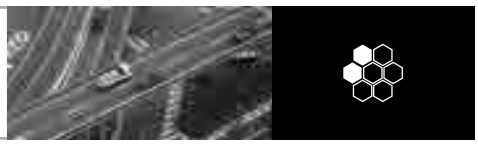
donde:

- *sa_handler* será bien la constante *SIG_DFL*, para la acción por defecto, o bien *SIG_IGN*, para ignorar la señal o la dirección a una función que maneje esa señal. Su significado, por tanto, es el mismo que el del parámetro *func* en la función *signal*.
- *sa_mask* proporciona una máscara para las señales que deben ser bloqueadas durante la ejecución del gestor de señales.
- *sa_flags* es la combinación *or* de ninguna o alguna de las siguientes *flags*:

flag	Descripción
SA_NOCLDSTOP	Si *signum* es *SIGCHLD*, no generar esta señal cuando un proceso hijo se para (control de trabajos).
SA_ONESHOT	Restablece la acción correspondiente a la señal una vez que se ha llamado al gestor de señales.
SA_RESTART	Opuesta a *SA_ONESHOT*. No restablece la acción de la señal.
SA_NOMASK	No previene a la señal de ser recibida dentro de su propio gestor de señales.

Esta función devuelve un 0, cuando se ejecuta sin problemas, y un –1, en caso de error. Los errores se deben a intentos de modificación de las acciones asociadas a las señales *SIGKILL* o *SIGSTOP*, las cuales, como ya se ha comentado, no pueden ser capturadas. También se pueden deber a que alguno de los punteros empleados por la función remite a posiciones de memoria que no forman parte del espacio de direcciones disponible para el proceso.

El ejemplo desarrollado para la función *signal* puede reproducirse ahora de forma más sencilla para la función *sigaction*. En principio, la función que gestiona la señal no debe preocuparse de reasignar la función ni de ignorar las señales, si el programador establece en la estructura *sigaction* únicamente el *flag SA_ONESHOT*, ya que este *flag* hace que la función que gestiona la señal sea siempre la definida en *sigaction*,



y bloquea la recepción de señales mientras se esté atendiendo la señal. Esto permite que el código de la rutina de tratamiento sea mucho más sencillo:

```
void gestor( int señal )
{
      printf("Señal SIGINT recibida");
}
```

Lógicamente, la reducción de la complejidad en la rutina de gestión implica su aumento a la hora de describir, mediante la estructura *sigaction*, cómo se va a comportar el gestor de señales.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
main(void)
{
      int codigo_error=0;
      struct sigaction gestion;
      gestion.sa_handler = gestor;
      gestion.sa_mask = 0;
      gestion.sa_flags = SA_ONESHOT;
      codigo_error = sigaction  ( SIGINT, gestion, 0);
      if( codigo_error == SIG_ERR )
      {
            perror("Error al definir el gestor de SIGINT");
            exit(-1);
      }
      /** Código del programa ***/
      while(1);
}
```

Del mismo modo que se ha asignado la función *gestor* a la señal *SIGINT*, se pueden asignar las constantes *SIG_IGN* y *SIG_DFL*. En el primer caso, el proceso ignoraría la señal:

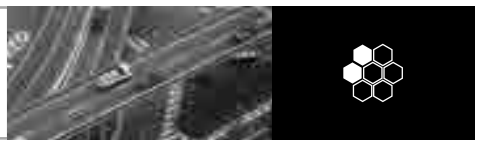
```
main(void)
{
      int codigo_error=0;
      struct sigaction gestion;
      gestion.sa_handler = SIG_IGN;
      gestion.sa_mask = 0;
      gestion.sa_flags = SA_ONESHOT;
      codigo_error = sigaction  ( SIGINT, gestion, 0);
      if( codigo_error == SIG_ERR )
      {
            perror("Error al definir el gestor de SIGINT");
            exit(-1);
      }
      /** Código del programa ***/
      while(1);
}
```

Por otra parte, si se asignase la constante *SIG_DFL*, tendría lugar el tratamiento por defecto:

```
main(void)
{
      int codigo_error=0;
      struct sigaction gestion;
      gestion.sa_handler = SIG_DFL;
      gestion.sa_mask = 0;
      gestion.sa_flags = SA_ONESHOT;
      codigo_error = sigaction  ( SIGINT, gestion, 0);
      if( codigo_error == SIG_ERR )
      {
            perror("Error al definir el gestor de SIGINT");
            exit(-1);
      }
      /** Código del programa ***/
      while(1);
}
```

Para modificar el gestor de señales, es necesario realizar una nueva llamada a la función *sigaction*. Por ejemplo, si en el programa anterior se incluye al principio un conjunto de instrucciones que se desea ejecutar sin que la señal *SIGINT* pueda interrumpirlas, y posteriormente se desea establecer la rutina de tratamiento *gestor*, el código del programa quedará como sigue:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

main(void)
{
      int codigo_error=0;
      struct sigaction gestion;
      gestion.sa_handler = SIG_IGN;
      gestion.sa_mask = 0;
      gestion.sa_flags = SA_ONESHOT;

      codigo_error = sigaction  ( SIGINT, gestion, 0);
      if( codigo_error == SIG_ERR )
      {
            perror("Error al definir el gestor de SIGINT");
            exit(-1);
      }

      /* Parte del código que ignoraría la señal SIGINT */

      ..................

      gestion.sa_handler = gestor;
```



```
      codigo_error = sigaction  ( SIGINT, gestion, 0);
      if( codigo_error == SIG_ERR )
      {
      perror("Error al definir el segundo gestor de SIGINT");
            exit(-1);
      }
/** Código del programa en el que se respondería a la señal
SIGINT con gestor ***/
      while(1);
}
```

C. Otras funciones: *pause, sigpause, setjmp* y *longjmp*

La función *pause* permite suspender la ejecución de un proceso hasta la recepción de una señal. Se dice entonces que el proceso queda en espera; como la llegada de una señal interrumpe este estado, se dice que *pause* espera la llegada de una señal. La definición de esta función es la siguiente:

```
include <unistd.h>
int pause (void);
```

Cuando por fin llega una señal, en primer lugar se ejecuta la rutina de tratamiento de la señal; después, la función *pause* devuelve un –1 y en *errno* se sitúa el valor *EINTR*, que indica que la señal ha sido recibida y que la función *pause* se ha ejecutado de forma correcta.

Con frecuencia, *pause* espera a una alarma de reloj; es decir, la espera finaliza con una señal *SIGALRM*. Ahora bien, hay que destacar que la función *pause* no es selectiva respecto al tipo de señal que puede terminar con su espera, lo que no sucede con la función *sigpause*.

Como ya se ha explicado, es posible detener un proceso hasta la recepción de una o varias señales concretas, lo que se puede fijar mediante la función *sigpause*. La definición de esta función es:

```
include <signal.h>
long sigpause (long mask);
```

Así, la función *sigpause* suspende el proceso hasta que llega una de las señales que no están bloqueadas en *mask*. Una vez que el proceso deja de estar detenido, la función *sigpause* devuelve el valor original a la máscara de señales. Al igual que ocurre en la función *pause*, devuelve un –1 y en *errno* se sitúa el valor *EINTR*, que indica que la señal ha sido recibida y que la función *sigpause* se ha ejecutado correctamente.

En C no es posible ir con un *goto* a una etiqueta que se encuentra en otra función. Para llevar a cabo este tipo de ramificaciones es preciso emplear las funciones *setjmp* y *longjmp* (se encuentran en <setjmp.h>). No obstante, su uso más relevante es el destinado a la captura de condiciones de error que aparecen en llamadas a funciones profundamente anidadas. La función *longjmp* se suele llamar desde un gestor de señales para retornar al bucle principal de un programa, en lugar de retornar directamente

desde el gestor. En realidad, el estándar ANSI C establece que un gestor de señales puede retornar a la función principal o llamar a *abort*, *exit* o *longjmp*.

Al ser llamada, la función *setjmp* guarda en una variable el entorno de pila. La definición de *setjmp* es:

```
include <setjmp.h>
int setjmp (jmp_buf env);
```

donde *env* es la variable donde se guarda el entorno de pila. El tipo *jmp_buf* se define en <setjmp.h>. El valor retornado es 0 la primera vez que se llama a la función.

Cuando se llama a la función *longjmp*, el programa salta a la línea donde se encuentra *setjmp*. Su definición es:

```
include <setjmp.h>
void longjmp (jmp_buf env, int val);
```

donde *env* es la variable donde se guardó el entorno de pila con *setjmp*, y *val*, el valor que retornará *setjmp* cuando el flujo de la ejecución salte desde *longjmp* a *setjmp*. Es decir, con una llamada a *setjmp*, el programador marca el punto al que saltará el flujo de ejecución, cuando se ejecute *longjmp*. Mientras se esté ejecutando el flujo normal del programa, *setjmp* devolverá el valor 0, y cuando se salte hasta *setjmp* haciendo uso de *longjmp,* el valor de retorno de la función será el especificado en *longjmp* en la variable *val*. La variable de entorno de pila sirve, en el flujo normal del programa, para restaurar el proceso al estado en el que se encontraba la última vez que se ejecutó *setjmp*.

El uso de estas dos funciones no está limitado a la gestión de señales, pero es aquí donde su empleo es más frecuente. Mediante *setjmp* se fija el punto del programa al que se desea volver después de recibir una señal. La función que gestiona la señal utiliza *longjmp* para permitir que, una vez gestionada la señal, el proceso continúe en un punto prefijado por el programador mediante *setjmp*.

Existe un problema cuando se llama a *longjmp*: la máscara de señales que se restaura en el punto al que se salta definido por *setjmp* es distinta de la máscara de señales que existía en el momento en que *setjmp* fue invocada. Evidentemente, en la variable de entorno se ha almacenado la información necesaria para que el proceso se siga ejecutando desde ese punto, pero esa variable de entorno no almacena la máscara definida para las señales. Cuando se captura una señal, la función que lo consigue comienza a ejecutarse automáticamente en cuanto ésta llega. Como se ha visto, al comienzo de la rutina, se desactiva la señal que se está tratando, para evitar la ejecución de la acción por defecto, que puede romper la propia ejecución. Para evitar responder a la señal correspondiente, la rutina añade la señal a la máscara de señales del proceso, lo que previene sobre apariciones de una señal procedente de la interrupción del gestor de señales. Esta nueva máscara es la que existe en el punto definido por *setjmp*, que no es la misma que estaba definida cuando se llamó a la sentencia *setjmp*. Para evitar este efecto indeseable se utilizan las funciones:

```
include <setjmp.h>
int sigsetjmp (sigjmp_buf env, int savemask);
void siglongjmp (sigjmp_buf env, int val);
```

donde la única variación respecto a lo comentado es la variable *savemask*, que, si tiene un valor no nulo, forzará a *sigsetjmp* a salvar la máscara de señales para, posteriormente, ser restaurada cuando el flujo del programa salte hasta *sigsetjmp* mediante una llamada a *siglongjmp*.

El uso de estas funciones dota al desarrollo de programas de otras funciones que los hacen más robustos frente a posibles errores en la ejecución. El programa debe, en primer lugar, instalar la rutina de tratamiento y, posteriormente, definir el punto en donde se va a producir el salto.

2.6 Comunicación entre procesos

En un sistema, los procesos pueden ejecutarse independientemente o cooperando entre sí. Los intérpretes de comandos son ejemplos típicos de procesos que no precisan la cooperación de otros para realizar sus funciones. En cambio, los procesos que sí cooperan necesitan comunicarse entre sí para poder completar sus tareas. La interacción entre procesos puede estar motivada por la competencia o el uso de recursos compartidos: por ejemplo, en los puertos de comunicaciones pueden concurrir varios procesos simultáneamente, de manera que es necesario utilizar un proceso que planifique el orden de acceso. Otro tipo de interacción entre procesos es aquel en el que deben ejecutarse sincronizadamente para completar una tarea, como un generador de escenas gráficas: por un lado, trabaja el proceso que interpreta la escena descrita en un lenguaje de alto nivel; por otro, el proceso con los algoritmos que representan el comportamiento físico de la luz. Para que puedan realizarse ambos tipos de interacciones, es necesario que el sistema operativo provea de servicios para posibilitar la comunicación entre procesos.

A continuación, se examinan algunas de las señales que pueden comunicarse a los procesos, así como su utilidad. Se estudia en profundidad el uso de las funciones de temporización para procesos que requieren ejecuciones en instantes de tiempo determinados, y, finalmente, se trata la sincronización entre procesos.

2.7 Comunicación de señales entre procesos

En este apartado estudiaremos las funciones que permiten enviar señales a un proceso, de proceso a proceso, o a un grupo de procesos. Esto es ligeramente distinto de lo estudiado en los apartados anteriores, donde se explicó que el *kernel* envía las señales cuando detecta algún tipo de evento excepcional. Las principales funciones son *kill*, *killpg* y *raise*.

A. Función *kill*

La función *kill* permite enviar una señal a un proceso; si no se especifica cuál, será, por defecto, la señal de terminación. Ésta pondrá fin a aquellos procesos que no la

capturen; para los que sí lo hagan y con los que se desee terminar, será necesario enviarles *kill* con la señal 9, ya que ésta no puede ser ni capturada ni ignorada. Con el mismo fin puede emplearse el comando *kill*.

La lista completa de señales puede encontrarse en el fichero </usr/include/linux/signal.h>. La principal diferencia entre la función y el comando homónimos reside en que el comando no se utiliza dentro de los programas de aplicación, y sí en la mayoría de las llamadas realizadas desde la línea de comandos.

La definición de la función *kill* es:

```
include <signal.h>
int kill (int pid, int sig);
```

donde *pid* es el número del proceso al que se quiere enviar la señal. En función del valor de *pid*, la tarea realizada será diferente:

- Si *pid* > 0, la señal *sig* se envía al proceso con identificador *pid*.
- Si *pid* = –1, la señal *sig* se envía a cualquier proceso de la tabla de procesos, excepto al primero.
- Si *pid* < –1, la señal *sig* se envía a cualquier proceso del mismo grupo del *pid*.

Si la operación se realiza con éxito, la función devuelve 0; en caso contrario, el valor devuelto es –1. Se debe tener en cuenta que, cuando se envía la señal a varios procesos, la operación tiene éxito cuando al menos uno de ellos recibe la señal. En este caso, no puede determinarse qué procesos capturaron la señal o si todos lo hicieron.

Los errores más frecuentes se deben, entre otros, al envío de una señal no válida, a que el *pid* indicado no existe o a que el proceso o procesos a los que se envía la señal no pertenecen a quien intenta enviarles la señal, es decir, que el propietario del proceso no es el de los procesos o el superusuario.

En la práctica, los usos más frecuentes de *kill* suelen tener uno de estos dos propósitos:

- Terminar con uno o más procesos, para lo que se envía *SIGTERM*, aunque a veces se emplea también *SIGQUIT* o *SIGIO*, si se desea obtener un fichero *core*.
- Probar el código de gestión de errores de un nuevo programa mediante la simulación de señales; por ejemplo, *SIGFPE*.

B. Función *killpg*

La función *killpg* permite a un proceso enviar una señal a un grupo de procesos del mismo modo que con la función *kill*. Su definición es:

```
include <signal.h>
int killpg (int pgrp, int sig);
```

donde *sig* es la señal que se desea enviar y *pgrp* es el grupo de procesos. Según el valor que toma *pgrp*, se puede encontrar:

- Si *pgrp* es 0, la función envía la señal al grupo de procesos al que pertenece el proceso que está enviando la señal.
- Si *pgrp* es positivo, la señal *sig* se manda al proceso cuyo *pid* es *pgrp*. En este caso, la función devuelve un 0, si todo ha ido bien, o un –1, en caso de error, el cual tiene asociado un código que se encuentra en *errno*.
- Si *pgrp* es igual a –1, la señal *sig* se envía a todos los procesos, excepto al primero, desde los valores más altos en la tabla de procesos a los más bajos. En este caso, la función devuelve un 0, si la acción ha tenido éxito, o la última condición de error de la *sig* enviada, si ha tenido lugar algún error.
- Si *pgrp* es menor que –1, la señal *sig* se envía a todos los procesos que pertenecen al mismo grupo que el proceso que envía la señal. En este caso, si hay un error, se devuelve un valor negativo y, si todo va bien, se devuelve el número de los procesos a los que se ha enviado la señal.

C. Función *raise*

La función *raise* se emplea cuando es necesario que un proceso se envíe señales a sí mismo. La definición es:

```
include <signal.h>
int raise (int sig);
```

donde *sig* es el número de la señal que se quiere enviar. El valor devuelto es 0, cuando se ha ejecutado con éxito, y un número distinto de 0, en caso de error. La función *raise* se puede implementar a partir de *kill* del siguiente modo:

```
kill (getpid(), sig);
```

donde *getpid* es una función que devuelve el *id* del proceso actual, y *sig* es el número de la señal que se desea enviar.

2.8 Mensajes entre procesos

Una arquitectura de procesos muy utilizada es la llamada *cliente-servidor*, que se basa, como su nombre indica, en que unos procesos *servidores* ofrecen servicios a los procesos *clientes*, como es el caso del sistema de correo electrónico diseñado por esta arquitectura, o de ciertas partes de algunos sistemas operativos (AMOEBA, MACA) implementados también con este tipo de arquitectura de procesos. Los procesos clientes emiten un mensaje de petición de servicio al proceso o procesos servidores, y éstos les devuelven un mensaje de respuesta.

A continuación, analizaremos cómo se puede crear una arquitectura cliente-servidor de tal manera que la ejecución del proceso cliente tenga sentido únicamente si existe el proceso servidor que le proporciona los datos que necesita; es decir, de forma que, si se establece una conexión entre un proceso servidor y otro cliente, la muerte del servidor provoque que el cliente se quede a la espera de los datos que el servidor debía enviarle.

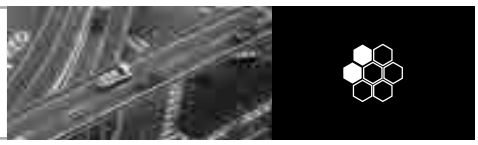
En estos casos, el programador del cliente suele proteger su programa de manera que, si pasado un cierto tiempo no se establece una comunicación entre el cliente y el servidor, el proceso cliente envía un mensaje al usuario en el que le indica la imposibilidad de comunicar con el servidor y finaliza la ejecución. Evidentemente, otra posibilidad es que el propio proceso servidor comunique al cliente que va a finalizar, para que éste actúe en consecuencia. Ahora bien, esta posibilidad sólo es viable mediante señales, ya que el servidor no sabe en qué instante recibirá la señal cuya recepción causará su muerte. En la rutina de tratamiento de esa señal, el servidor puede enviar, mediante las funciones expuestas, una señal al cliente y posteriormente morir. El cliente recibirá esa señal, avisará al usuario de la caída del servidor en la rutina de tratamiento y luego morirá.

Con lo visto hasta ahora, se puede rebatir la dependencia de un proceso respecto a otro mediante la generación de un proceso hijo.

Ejemplo práctico

1 En el programa que se muestra a continuación, el proceso padre vive hasta que le llega la señal *SIGINT*, bloqueada para el hijo. En ese momento entra en la rutina de tratamiento y envía una señal al proceso hijo, que éste captura, y posteriormente muere.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

/* Variable global con el pid del proceso hijo */

int pid = 0;

/* Rutina de tratamiento del mensaje SIGINT desde donde se va a mandar el mensa-
je al proceso hijo para que éste muera */

void gestor_SIGINT( int señal )
{
   printf("Rutina de tratamiento de SIGINT\n");
   if( pid != 0 )
   {
         kill( pid, SIGUSR1 );
         exit( 0 );
   }
}

void gestor_hijo( int señal )
{
   exit(0);
}
main( void )
{
   int codigo_error=0;
```



```
struct sigaction gestion;
   /* Primero se instala el gestor de la señal SIGINT */
   gestion.sa_handler = gestor_SIGINT;
   gestion.sa_mask = 0;
   gestion.sa_flags = SA_ONESHOT;
   codigo_error = sigaction  ( SIGINT, gestion, 0);
   if( codigo_error == SIG_ERR )
   {
         perror("Error al definir el gestor de SIGINT");
         exit(-1);
   }
   if( (pid = fork()) == -1 )
   {
         printf("Error en la creación del hijo\n");
         exit( 1 );
   }
   else if( pid == 0 )
{
   /* Código del hijo */

   /* Primero se desactiva la respuesta a SIGINT */
   gestion.sa_handler = SIG_IGN;
   gestion.sa_mask = 0;
   gestion.sa_flags = SA_ONESHOT;
   codigo_error = sigaction  ( SIGINT, gestion, 0);
   if( codigo_error == SIG_ERR )
   {
   perror("Error al definir el gestor de SIGINT");
         exit(-1);
   }
   /* Segundo se instala el gestor de SIGUSR1 */
   gestion.sa_handler = gestor_hijo;
   gestion.sa_mask = 0;
   gestion.sa_flags = SA_ONESHOT;
   codigo_error = sigaction  ( SIGUSR1, gestion, 0);
   if( codigo_error == SIG_ERR )
   {
   perror("Error al definir el gestor de SIGUSR1");
         exit(-1);
   }
         /* Código */
         while(1)
         }
   else
   {
   /* Código del padre */
         while(1);
         }
   }
```

2.9 Temporización de procesos

La temporización de procesos se relaciona con aquellos procesos en los que es necesario que la ejecución se lleve a cabo en unos espacios de tiempo determinados; por ejemplo, que cada dos segundos se realice cierta operación. Para este tipo de procesos es necesario disponer de una función que permita controlar el tiempo y enviar al proceso una señal de aviso cuando se haya cumplido el tiempo establecido.

En otros casos, puede ser necesario tener uno o varios procesos detenidos hasta el momento en que se quiera que continúen ejecutándose. Estas cuestiones pueden resolverse con el uso de las funciones *alarm* y *pause*, que generan señales que, enviadas al proceso correspondiente, consiguen avisarle de que ha transcurrido cierto tiempo o "despertarlo".

En los sistemas operativos de tipo multiproceso, los procesos se ejecutan en paralelo, por lo que el tiempo real transcurrido en la ejecución de un proceso no es exactamente el tiempo que ha tardado el proceso en realizar un conjunto de instrucciones. El tiempo medido como la diferencia entre el instante de comienzo de ejecución y el instante de fin de ejecución representa, en concreto, el tiempo que ocupa la ejecución del código entre los dos puntos, más el tiempo de todos los procesos que en ese momento se encuentren en la máquina, ya que al ser un sistema operativo multiproceso todos los procesos se ejecutan compartiendo los recursos. Si no existe más que nuestro proceso y la CPU es utilizada de manera permanente por él, ese tiempo sí que coincidiría con el tiempo de ejecución del proceso. Las medidas de tiempo transcurrido, por tanto, pueden ser relativas al tiempo de la máquina o al tiempo de ejecución. Para examinar estas diferencias, vamos a estudiar, en primer lugar, las funciones de C que trabajan con unidades de tiempo, antes de especificar las funciones que permiten instalar temporizadores para un proceso.

A. Tiempo de sistema

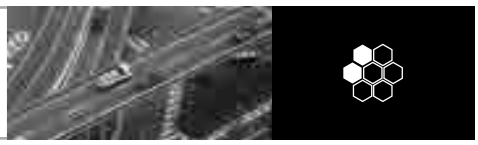
Para trabajar con el tiempo del sistema se utiliza una medida de tiempo que representa los segundos transcurridos desde las 00:00:00 GMT *(Greenwich Mean Time)* del día 1 de enero de 1970. El origen de tiempo, por tanto, se fija en esa hora, de manera que toda la información temporal tiene como referencia ese preciso instante. Para fijar la hora del sistema se utiliza:

```
#include <time.h>
int stime (long *tloc);
```

donde *tloc* es un puntero que contiene la hora actual respecto al origen de tiempos. Devuelve 0, si se ejecuta satisfactoriamente, y –1, en caso contrario.

Para leer la hora actual se utiliza la función:

```
#include <time.h>
time_t time (time_t *tloc);
```



donde *tloc* es un puntero que, si no vale 0, la función rellenará con la hora actual respecto al origen de tiempos.

Cuando no es suficiente con la resolución en segundos de las dos funciones anteriores, se pueden utilizar:

```
#include <time.h>
int gettimeofday (struct timeval *tloc, struct timezone
*tzp);
int settimeofday (struct timeval *tloc, struct timezone
*tzp);
```

Estas funciones permiten una resolución en microsegundos y, además, posibilitan la definición del tiempo para una zona horaria distinta de la del meridiano de Greenwich. El tiempo que retorna *gettimeofday* o que se le pasa a *settimeofday* se define a través de la estructura *timeval*, y la zona horaria, a través de la estructura *timezone*.

La definición de estas dos estructuras es la siguiente:

```
struct timeval
{
      unsigned long tv_sec;
      /*Tiempo respecto al tiempo de referencia*/
long tv_usec; /* Microsegundos */
};
struct timezone
{
      int tz_minuteswest;
      /* Variación en minutos de la hora local respecto de
la referencia */
      int tz_dsttime;
      /* Corrección de la hora según estaciones */
};
```

Cuando la variable de entorno *tz* se halla definida, no es necesaria la corrección horaria representada en *tzp*. Existe una función en <time.h> que permite obtener la hora de forma que su lectura resulte más cómoda:

```
#include <time.h>
struct tm *localtime( const time_t *timer);
```

donde *timer* es del tipo *time_t* y es la hora en segundos respecto a la hora de referencia. Devuelve una estructura del tipo *tm* que contiene la hora, el día, el mes, el año, etc.

La estructura *tm* se define como sigue:

```
#include <time.h>
struct tm
{
      int tm_sec; /* Segundos de 0 a 59 */
      int tm_min; /* Minutos de 0 a 59 */
```

```
      int tm_hour; /* Hora de 0 a 23 */
      int tm_mday; /* Día del mes, del 1 al 31 */
      int tm_mon; /* Mes del año, del 0 al 11 */
      int tm_year; /* Año, en decenas desde 1900 */
      int tm_wday; /* Día de la semana:
                        Domingo = 0
                        Lunes = 1
                        ......
                        Sábado = 6 */
      int tm_yday; /* Día del año, del 0 al 365 */
      int m_isdst; /* Para la corrección de la fecha */
};
```

B. Tiempo de proceso

A la hora de contar el tiempo real que un proceso consume en la ejecución de un conjunto de sentencias, se considera únicamente el tiempo de ejecución de ese proceso y el de sus hijos (y los hijos de sus hijos, etc.), para los que el proceso padre está esperando su terminación mediante una llamada a *wait*.

Este tiempo se mide en pulsos de reloj: para obtener la medida del tiempo en segundos, se debe conocer el número de pulsos por segundo que se encuentran en *CLK_TCK*. A partir de este valor, los segundos que el proceso tarda en ejecutar un conjunto de líneas de código se puede calcular como:

tiempo = número de pulsos / *CLK_TCK*

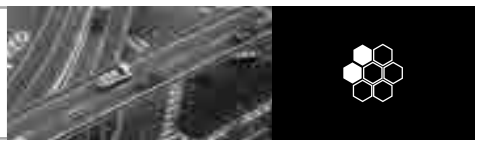
El tiempo contabilizado se divide en cuatro tipos distintos:

- Tiempo empleado por la CPU en ejecutar el proceso en modo usuario.
- Tiempo empleado por la CPU en ejecutar el proceso en modo *kernel*.
- Tiempo empleado por la CPU en ejecutar los procesos hijo (y los hijos de los hijos, etc.) en modo usuario.
- Tiempo empleado por la CPU en ejecutar los procesos hijo (y los hijos de los hijos, etc.) en modo *kernel*.

Existe una estructura en C que refleja estos cuatro tipos de medidas:

```
struct tms

{
      clock_t tms_utime; /* CPU modo usuario */
      clock_t tms_stime; /* CPU modo kernel */
      clock_t tms_cutime; /* CPU hijos modo usuario */
      clock_t tms_cstime; /* CPU hijos modo kernel */
};
```

donde *clock_t* se define para contar los pulsos de reloj.



La función que permite rellenar esta estructura es:

```
#include <sys/times.h>
clock_t times (struct tms *buffer);
```

donde *buffer* es la estructura que rellena la función *times*.

Si todo va bien, devuelve el número de pulsos transcurridos desde un instante anterior elegido de modo arbitrario; si, por el contrario, algo va mal, devuelve –1 y el código de error correspondiente en *errno*. El punto arbitrario al que hace referencia la función *times* puede ser el momento de arranque del sistema o cualquier otro; lo que importa es que ese momento no varía de una llamada a otra, por lo que se puede calcular el tiempo transcurrido como la diferencia entre los dos tiempos devueltos por dos llamadas sucesivas a *times*.

C. Funciones *getitimer* y *setitimer*

Como se ha explicado en la descripción de la señal *SIGALARM* de la tabla 2.1, la función *alarm* permite controlar intervalos de tiempo medidos en segundos. Cuando sea necesario un control más preciso, se emplearán las funciones *getitimer* y *setitimer*. Con todas estas funciones, el sistema provee a cada proceso de tres temporizadores distintos, cada uno de los cuales va paulatinamente reduciéndose en un dominio de tiempo diferente: tiempo real, tiempo de ejecución en modo usuario o tiempo de ejecución en modo *kernel*.

Cuando alguno de los temporizadores llega a 0, se envía una señal al proceso distinta para cada temporizador. El temporizador se restablece con un tiempo prefijado al definirlo.

El procedimiento para utilizar estos temporizadores es:

- **Definición del temporizador.** Incluye, por un lado, la declaración del tiempo en segundos y microsegundos que el temporizador debe consumir en un principio. Por otro, se define el valor con el que deberá inicializarse cada vez que termine. Si este tiempo de reinicialización se define con el valor 0, no volverá a generar más alarmas que la primera.

- **Captura de la señal.** Cada vez que el temporizador llega a 0 manda una señal al proceso, de manera que el programador utiliza la rutina de captura para ejecutar de manera síncrona las acciones deseadas con el tipo de tiempo definido con anterioridad.

- **Reinicialización automática del temporizador.** Una vez que el temporizador genera la señal, vuelve a comenzar el proceso de manera automática, sin que sea necesario que el programador deba indicárselo.

La función *getitimer* permite obtener el tiempo del temporizador, así como su valor de reinicialización. Su definición es:

```
include </sys/time.h>
int getitimer (int cual, struct itimerval *valor);
```

donde *cual* define el tipo de temporizador y *valor* es una variable de tipo *itimerval*, en la que se recoge el valor del tiempo inicial dado al temporizador y el tiempo de reinicialización. Devuelve 0, si la ejecución se lleva a cabo de forma correcta, y –1, en caso contrario; el tipo de error queda recogido en *errno*.

La función *setitimer* permite definir el tiempo del temporizador y su valor de reinicialización, al tiempo que devuelve los dos valores de la configuración anterior. Su definición es:

```
include </sys/time.h>
int setitimer (int cual, const struct itimerval *valor,
struct       itimerval *ovalor);
```

donde *cual* y *valor* tienen el mismo significado que en *getitimer*, y *ovalor* es una variable de tipo *itimerval*, en la que la función devuelve el valor del tiempo inicial y el tiempo de reinicialización de la anterior configuración del temporizador, si *ovalue* es distinto de 0. Devuelve 0, si la ejecución se lleva a cabo de forma correcta, y –1, en caso contrario; el tipo de error queda recogido en *errno*.

El atributo *cual* define uno de los tres tipos de temporizador y puede tomar uno de los siguientes valores:

- *ITIMER_REAL*: disminuye en tiempo real. Cuando llega a 0, genera la señal *SIGALARM*.
- *ITIMER_VIRTUAL*: disminuye únicamente cuando el proceso se está ejecutando. Cuando llega a 0, genera la señal *SIGVTALRM*.
- *ITIMER_PROF*: disminuye tanto si el proceso se está ejecutando como si el sistema está actuando en nombre del proceso. Si se considera junto a *ITIMER_VIRTUAL*, permite estimar el tiempo empleado por la aplicación en espacio de usuario y *kernel*. La señal que se genera cuando llega a 0 es *SIGPROF*.

La estructura *itimerval* que define los parámetros del temporizador se define como sigue:

```
struct itimerval
{
     struct timeval it_interval; /* siguiente valor */
     timeval it_value; /* valor actual */
};
```

El elemento *it_value* se fija con la cantidad de tiempo restante del temporizador, o con 0, si el temporizador no está activado. De manera análoga, en *it_interval* se establece el valor de reinicialización.

Los temporizadores disminuyen el tiempo desde *it_value* hasta 0, generan una señal (*SIGALRM, SIGVTALRM* o *SIGPROF*, según el tipo de temporizador) y vuelven al valor de reinicialización contenido en *it_interval*. Un temporizador que se pone a 0 se para. Puede llegar a 0, porque *it_value* sea 0, debido a que ha expirado su tiempo o a que *it_interval* es igual a 0.

D. Aplicación de los temporizadores

Existe una función en C, denominada *sleep*, que detiene la ejecución de un proceso hasta que se recibe alguna señal. Con lo visto hasta aquí, somos capaces de desarrollar una función que cumpla estas características. La función se va a denominar *mi_sleep* y recibirá como parámetro un número que indique el tiempo en segundos que se desee detener la ejecución de un proceso.

Ejemplo práctico

2 En principio, la función sería algo tan sencillo como definir un gestor para la señal *SIGALRM* y un temporizador con el tiempo definido en su parámetro. Vamos a denominar a esta primera versión de la función *mi_sleep1*. En este caso, su código sería:

```
#include <stdio.h>
#include <signal.h>

void gestor_SIGALRM( int senal )
{
   printf("Ha llegado la señal del temporizador\n");
}

/* La función devuelve el número de segundos que quedan para terminar el tiempo
fijado por el programador; éste será 0, a menos que haya ocurrido algún error */
int mi_sleep1( int segundos )
{
   int codigo_error;

/* Se instala el gestor con signal, porque únicamente se va a recibir una señal,
la que vamos a generar fijando el temporizador */

codigo_error = signal( SIGALRM, gestor_SIGALRM );

if( codigo_error == SIG_ERR )
   {
         return segundos;
   }

/* Definimos el temporizador */
   alarm( segundos );

/* Esperamos la llegada de la señal SIGALRM enviada por el temporizador */
   pause();

/* Retornamos lo que queda, que será 0, si no ha habido otra señal SIGALRM */
   return ( alarm(0) );
}
```

2.10 Sincronización de procesos

Una de las principales utilidades de las señales es la sincronización entre dos procesos. Por **sincronización** se entiende que un proceso realizará un conjunto de instrucciones cuando otro proceso se lo indique, o bien paralizará una actividad hasta que se cumpla una condición determinada.

Hasta que no conozcamos los medios de comunicación que permiten intercambiar datos, la única posibilidad que tenemos para que dos procesos conozcan el *pid* de otro proceso es la ejecución de dos procesos, uno padre y otro hijo. En este sentido, vamos a desarrollar un ejemplo en el que un proceso padre y un proceso hijo se ejecutan de forma síncrona; el código de lo que realiza cada uno de los procesos no es relevante para el ejemplo.

La ejecución del proceso padre crea, en primer lugar, el proceso hijo; éste comenzará entonces a ejecutar un código cualquiera, de forma independiente de lo que debe ejecutar el proceso padre. Cuando llegue el momento en que el hijo necesite que el padre realice un conjunto de acciones, quedará a la espera de que el padre termine las operaciones oportunas y envíe una señal para que comiencen a ejecutarse las acciones del hijo; esto es lo que llamamos ejecutar dos procesos de forma sincronizada. El proceso padre enviará una señal *SIGUSR1* al proceso hijo cuando desee que comience a ejecutarse. Una vez que el hijo haya terminado las acciones que tenía asignadas, enviará una señal *SIGUSR1* al proceso padre.

Evidentemente, la ejecución del hijo depende de la ejecución del padre. Si en el proceso padre ocurre algún error en la ejecución de acciones previas a la llamada al hijo, o bien se determina que no es necesario que el proceso hijo continúe ejecutándose, el proceso padre envía una señal *SIGTERM* para que finalice el proceso hijo sin realizar ninguna acción. Cuando el proceso padre no puede llevar a cabo el trabajo que tiene asignado, no tiene sentido que el hijo realice ninguna operación.

Esta secuencia de acciones ejecutadas en paralelo se puede repetir infinitamente; es decir, el bucle de acciones conjuntas sería:

1. El proceso padre crea el proceso hijo.

2. El proceso padre ejecuta un conjunto de acciones a partir de las cuales se desea que el proceso hijo continúe.

3. Si no hay error y se desea que el hijo ejecute sus acciones:
   3.1. El proceso padre envía una señal *SIGUSR1* al hijo para que comience.
   3.2. El proceso hijo realiza un conjunto de acciones.
   3.3. El proceso hijo envía la señal *SIGUSR1* al padre.
   3.4. Volvemos a 2.

4. En caso contrario:
   4.1. El proceso padre envía una señal *SIGTERM* al hijo para que termine.
   4.2. El proceso hijo termina.
   4.3. El proceso padre termina.

El código completo de las acciones que se realizan en los procesos padre e hijo se encuentra en las rutinas de tratamiento de la señal *SIGUSR1*, que es distinta para el padre y el hijo. Evidentemente, para que todo el proceso comience, se debe llamar por primera vez a la rutina de gestión de la señal *SIGUSR1* en el padre, sin esperar a la recepción de la señal; es decir, directamente desde el código.

Ejemplo práctico

3 Como el código al completo se basa en la gestión de las señales, lo único que hace el código del programa principal es dejar los procesos en *pause*; es decir, esperando la recepción de señales.

```
#include <stdio.h>
#include <signal.h>

                /* pid del proceso padre y del hijo */
int pid_padre, pid_hijo;

                /* Rutina de tratamiento de SIGUSR1 para el padre */

                void gestor_padre( int senal )
{
   int exito;
   printf("ACCIONES DEL PADRE\n");
/*Se llevan a cabo las acciones que corresponden al padre*/
.................................
/* Si estas acciones han tenido éxito y el programador quiere llamar al proceso
hijo pondrá la variable exito a 1 y, si no, a 0 */
/* Llamamos al proceso hijo */
   if( exito )
   {
   /* Reinstalamos el gestor para la próxima señal */
   signal( SIGUSR1, gestor_padre );
   kill( pid_hijo, SIGUSR1 );
   }
   else
   {
   /* Termina el proceso padre y el hijo */
   kill( pid_hijo, SIGTERM );
                printf( "Se termina la ejecución \n");
                exit(0);
                }
                }
/* Rutina de tratamiento de SIGUSR1 para el hijo */
void gestor_hijo( int senal )
{
   printf("ACCIONES DEL HIJO\n");
/*Se llevan a cabo las acciones que corresponden al hijo*/
.................................
```

```
    /* Reinstalamos el gestor para la próxima señal */
    signal( SIGUSR1, gestor_hijo );
    /* Llamamos al proceso padre */
    kill( pid_padre, SIGUSR1 );
}
                /* FUNCIÓN PRINCIPAL */
main( void )
{
    pid_padre = getpid();
if( (pid_hijo = fork() ) == -1 )
    {
          printf( "Error al crear el proceso hijo\n");
          exit( -1 );
    }
    else if( pid_hijo == 0 )
    {
    /* Código del hijo */
    signal( SIGUSR1, gestor_hijo );
    while( 1 )
    {
          pause();
    }
}
else
{
    /* Código del padre */
/* empezamos la ejecución del programa síncrono */
    gestor_padre(0);
    while(1)
    {
          pause();
    }
}
exit(0);
}
```

Cuando el programa principal llama la función *gestor_padre* comienza la ejecución síncrona y los procesos se envían señales mutuamente. La primera señal la envía el padre al hijo cuando termina de ejecutar *gestor_padre*. Si el código de esta rutina es muy corto, puede ocurrir que la señal llegue al hijo antes de que se encuentre operativo. Para evitar este problema, podemos hacer que el proceso padre espere unos segundos:

```
    ..................
    /* Código del padre */
    sleep( 3 );
    /* Empezamos la ejecución del programa síncrono */
    gestor_padre(0);
    while(1)
    {
          pause();
    }
    ..................
```

Ejercicios

1. **Desarrolla un programa que calcule la pendiente de una recta y, en caso de error de coma flotante (división por 0), capture la señal de error y genere una pendiente aleatoria. El programa deberá seguir ejecutándose, aunque los resultados no sean los esperados.**

2. **Desarrolla un programa que, mediante *setjmp* y *longjmp*, sea capaz de gestionar un mensaje *SIGINT* generado por el usuario, de manera que, en vez de terminar el proceso, comience su ejecución desde el principio; es decir, que convierta la señal *SIGINT* en una reinicialización del programa.**

3. **Soluciona el problema que puede aparecer en la implementación del ejemplo de la sección 2.10. (d) cuando el programador fija un tiempo en segundos tan pequeño que el temporizador genera la señal *SIGALRM* antes de entrar en *pause*, ya que entonces se llama a la rutina antes de *pause* y, como ya ha finalizado la ejecución del temporizador, no se va a generar ninguna otra señal *SIGALRM*, con lo que nunca saldrá de *pause*.**

4. **El uso de temporizadores no asegura que la ejecución del proceso sea perfectamente síncrona; de hecho, si trata de realizar siempre el mismo número de acciones entre dos señales del temporizador definido con el mismo número de segundos, se observará que no siempre puede realizar las mismas acciones. Para comprobarlo, desarrolla un programa que escriba en un fichero la letra *a* continuamente entre dos señales *SIGALRM*. Esto significa que el programa definirá un temporizador de *n* segundos y comenzará a escribir en fichero; al llegar la señal cerrará ese fichero y abrirá otro distinto (con nombres a1, a2, a3, a4, etc.), definirá de nuevo el temporizador y comenzará a escribir en el nuevo fichero. Se debe verificar si, tras cientos de ejecuciones y, por tanto, cientos de ficheros generados, el número de letras escritas en cada fichero es exactamente el mismo.**

5. **Para comprobar si el número de letras de los ficheros del ejercicio anterior es el mismo, resultará más cómodo ir contando las letras y escribiendo en un fichero, al que llamaríamos *resultado*, el número de letras para cada ejecución. Realiza esta mejora utilizando un proceso hijo que cuente el número de letras de un fichero. El proceso padre avisará al proceso hijo de que ha terminado de escribir, y éste contará las letras del último fichero generado por el padre y escribirá en resultado el nombre del fichero (a1, a2, a3, etc.) y el número de letras.**

Evaluación

1. **Determina qué utilidades principales posee el uso de señales entre procesos respecto a otras formas de comunicación, como semáforos o *pipes*.**
2. **Enumera las posibles acciones que puede realizar el *kernel* al aparecer una señal.**
3. **¿Qué le sucede a un proceso sin gestor de señales cuando recibe una señal?**
4. **Explica las distintas formas de gestionar una señal indicando cuál es el uso de cada una de ellas.**
5. **Indica las diferencias entre la función y el comando *kill* del sistema operativo UNIX.**
6. **¿Qué función se debería emplear para que un proceso se envíe señales a sí mismo?**
7. **¿Cuáles son las cuatro medidas de tiempo que se contabilizan en el sistema operativo?**
8. **Indica las diferencias entre las funciones de control del tiempo *alarm*, *getitimer* y *setitimer*.**

Actividades prácticas

1. **Utiliza las funciones que permiten bloquear las señales en un proceso, realizando un programa que modifique la máscara de señales a medida que emplee la función *raise* para comprobar que esas señales no son en realidad recibidas.**

2. **Partiendo del ejercicio anterior, utiliza la función *sigpending* para observar que las señales se generan, pero no afectan a la ejecución del proceso.**

3. **Desarrolla una aplicación que grabe periódicamente en un fichero los procesos que se están ejecutando, ordenados por su fecha de creación. El periodo se pasará como un argumento en línea de comando.**