

5. Desarrollo de programas

Un programa consta de una serie de instrucciones que indican al sistema de computación qué operaciones aritméticas, lógicas, etc. realizar. Estas instrucciones pueden escribirse en lenguajes de alto nivel (y por tanto independientes generalmente del hardware) o no. Si bien, finalmente todas ellas serán transformadas en instrucciones en lenguaje máquina y dependientes del hardware.

Estas instrucciones serán ejecutadas por la UCP e indican la realización de alguna función básica, constando normalmente de un código de operación y uno o más operandos. El código de operación especifica el procesamiento a realizar mientras que los operandos indican la dirección de memoria o los datos a manipular.

5.1 Programación: creación de programas ejecutables.

La programación es la escritura de secuencias de instrucciones encaminadas resolver, mediante su ejecución, cierta problemática. Al igual que ocurre con el hardware, los lenguajes de programación han evolucionado a lo largo de varias generaciones, de manera que cada una de ellas supera a la anterior al ofrecer más facilidades y potencia.

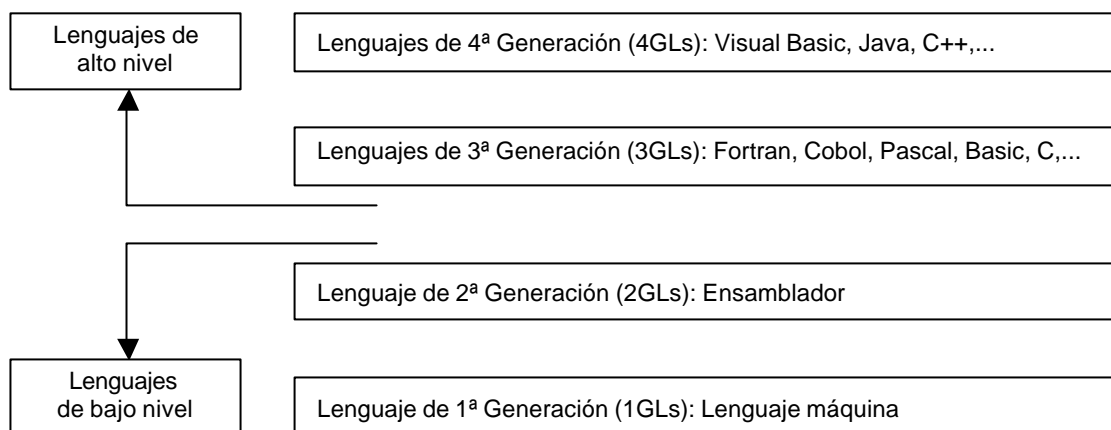


Figura 5: Generaciones de los lenguajes de programación

5.1.1 LENGUAJE MÁQUINA

En este lenguaje, las instrucciones se codifican como series de unos y ceros, por lo que escribir programas en este lenguaje se convierte en una tarea tediosa y no obstante, los programas escritos en este lenguaje son los únicos programas que los

computadores entienden y por tanto, pueden ejecutar. Cualquier programa escrito en otros lenguajes deberá traducirse a este lenguaje, propio de cada máquina.

5.1.2 LENGUAJE ENSAMBLADOR

Se obtiene del reemplazo de los unos y ceros correspondientes a las instrucciones en lenguaje máquina: a los códigos de operación y/u operandos, por símbolos, mnemónicos o mnemotécnicos¹⁷. En consecuencia, el lenguaje ensamblador es diferente para cada hardware puesto que tanto el conjunto de instrucciones como su representación puede variar de una plataforma a otra. No obstante, para que un programa escrito en este lenguaje pueda ser ejecutado, debe traducirse a su correspondiente en unos y ceros. Para ello se utiliza un compilador denominado *ensamblador*.

5.1.3 LENGUAJES DE ALTO NIVEL

Sus instrucciones constituyen conjuntos lingüísticos que acortan las diferencias entre el lenguaje humano y el lenguaje de comunicación con los computadores. Independientemente del computador cualquier programa escrito en un lenguaje de alto nivel debe ser traducido a lenguaje máquina antes de poder ser ejecutado. Esta traducción se realiza mediante programas software denominados *compiladores* e *intérpretes*.

5.1.4 PASOS PARA LA CREACIÓN DE UN PROGRAMA

Sea cual sea el sistema operativo instalado y el lenguaje de programación utilizado, la generación de un programa ejecutable conlleva los siguientes pasos:

1. Creación del programa o código fuente y su almacenamiento en un archivo.
2. Creación del programa objeto (código y módulos objeto).
3. Creación del código ejecutable o módulo de carga.

Código fuente.

Mediante un editor se procede a la escritura de un programa y almacenarlo en un archivo. Este *código fuente*, si se ha escrito en un lenguaje de programación distinto

¹⁷ Vocablos que facilitan su uso y memorización.

del lenguaje máquina, será incomprensible para el ordenador y por tanto deberá traducirse.

La traducción de este código a un lenguaje comprensible por la máquina es la labor que desempeñan compiladores e intérpretes, y el resultado de su ejecución es el *código objeto*.

Código objeto.

El código objeto está en lenguaje máquina, y por tanto no puede ser enviado a la impresora o visualizado mediante un editor.

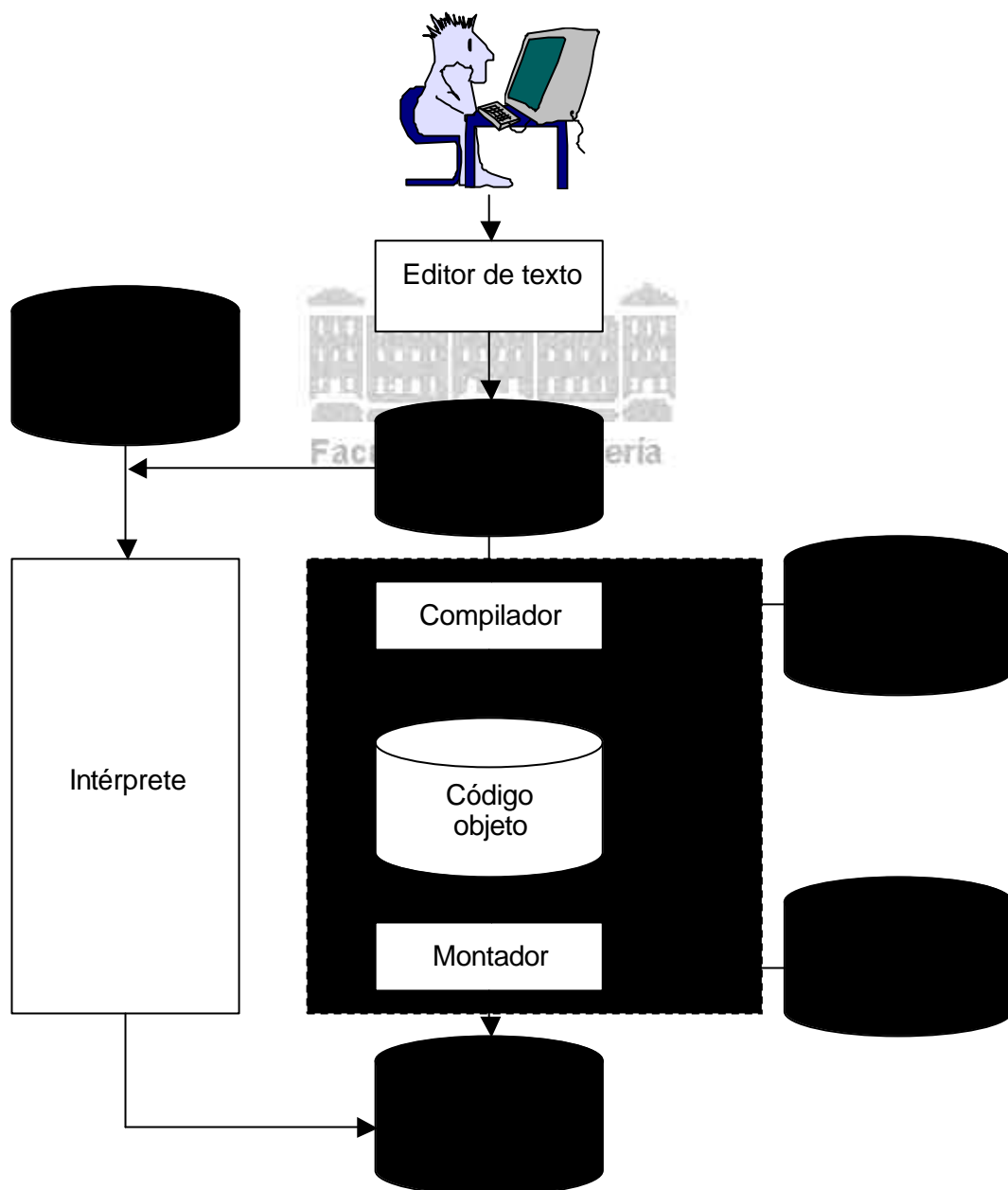


Figura 6: Pasos para la creación de un programa ejecutable

Código ejecutable.

El código objeto puede hacer referencias a otros programas – otros códigos objeto – o bien contener referencias o llamadas a servicios del sistema operativo. Por tanto, aún no es ejecutable. Le faltan algunas partes, trozos de código que proporcionan el interfaz entre el programa y el sistema operativo, y que suelen estar en *archivos de biblioteca*. Todas estas referencias deben ser resueltas para obtener código ejecutable. Esta resolución es la labor del *montador* o *editor de enlaces* que crea el *código ejecutable* o el *módulo de carga*, que es un programa ejecutable. Al igual que en el caso anterior no es posible enviar este código a impresora o visualizarlo mediante editores de texto.

5.2 Compiladores e intérpretes

La función principal de los compiladores y de los intérpretes es la traducción de código fuente (conjunto de instrucciones escritas en un lenguaje de programación de alto nivel) a código máquina.

Sin embargo, un compilador difiere respecto de un intérprete, de manera que suele ser posible categorizar los lenguajes de programación en lenguajes compilados y lenguajes interpretados según hagan uso de uno u otro.

5.2.1 COMPILADOR

Es un programa del sistema que traduce las instrucciones de un programa escrito en un lenguaje de programación de alto nivel – como C, Pascal, etc. – a lenguaje máquina. Procesa, traduce o *compila* el programa completo, y genera, si todo es correcto, un código objeto que puede ser almacenado, y si no, los mensajes relativos a posibles errores y/o advertencias. Cada lenguaje de programación requiere un compilador, que varía con las distintas versiones del lenguaje de programación.

Los compiladores suelen generar código objeto generalmente más óptimo y eficiente que el de los intérpretes, logrando que la ejecución del programa compilado sea más rápida y precise de menos espacio. Además, una vez compilado, es posible hacer uso cuantas veces se desee al código objeto generado.

Algunos ejemplos de lenguajes generalmente compilados podrían ser C, COBOL, Pascal, Visual Basic, etc.

5.2.2 INTÉRPRETE

Al igual que el compilador, traduce un programa escrito en lenguaje de alto nivel a lenguaje máquina. Sin embargo, en vez de traducir el programa fuente completo “de una pasada” va traduciendo instrucción a instrucción, generando una realimentación inmediata. Así, si el código fuente contiene algún error, el intérprete lo muestra al procesar la línea que lo contiene. De esta manera, los intérpretes son idóneos para la detección y corrección de errores, minimizando el tiempo de producción de código.

Su principal inconveniente se deriva de la no generación de un archivo de código objeto, que implica la necesidad de realizar el proceso de traducción cada vez que se ejecuta el programa en cuestión. Además, el código ejecutable que generan suele ser menos eficiente que el producido por un compilador, pero es necesario disponer de menos espacio en disco puesto que no hay archivos objeto ni archivos ejecutables.

Un ejemplo de lenguaje interpretado es Lisp, de frecuente utilización en el área de la Inteligencia Artificial. Otro ejemplo, si bien debe puntualizarse, lo constituye Java, que aunque es fundamentalmente un lenguaje interpretado, hace uso de un compilador para traducir ciertos fragmentos de código, en lo que se conoce como compilación JIT¹⁸ (Just-In-Time).

Pero quizás el lenguaje interpretado por excelencia hoy en día sea HTML (HyperText Markup Language) y que es utilizado para describir el contenido (texto, sonido, imágenes, etc.) de documentos web. De hecho, cada vez que una página es visualizada por un navegador, éste interpreta el código HTML que define su contenido.

Otros lenguajes interpretados se utilizan en las páginas ASP (*Active Server Pages*), todos los *shell scripts* o guiones de *shell*, y en cierta manera, Visual Basic, cuyos ejecutables, a pesar de tener extensión *.exe* no son sino series de comandos que son interpretados por las *dlls* (*dynamic linking library*) de Visual Basic.

5.3 Utilidad *make* en Unix

Se trata de una utilidad necesaria cuando el código fuente de un programa se encuentra repartido en archivos. La utilidad *make* mantiene el control de todos estos archivos fuente, registrando cuándo han sido modificados por última vez.

¹⁸ Método de producción de código consistente en compilar los fragmentos que son interpretados en repetidas ocasiones, evitando así volver a pasar varias veces por el mismo código.

Cuando se ejecuta *make* por primera vez sobre un programa, se compilan todos los archivos fuente que conformen un mismo programa. Cada vez que es invocada de nuevo sólo compila los archivos modificados desde la última vez que se activó; compila los módulos fuente cuya fecha y/u hora sean posteriores a la del objeto.

En cualquier caso, una vez generados los códigos objeto, enlaza todos ellos, con el fin de generar un único ejecutable.

La información que necesita la utilidad *make* se obtiene a partir de un archivo de control, que contiene reglas que especifican, entre otras informaciones, las dependencias de los archivos fuente.

El Entorno Integrado de Desarrollo (IDE) Borland C++ dispone de una utilidad semejante, al permitir que se lleve a cabo la compilación únicamente de aquellos archivos de código fuente que han sido modificados desde la última compilación (opción *build*), frente a la opción de compilar todos los archivos que componen una aplicación hayan sido éstos modificados o no (*build all*).

5.4 Señales

Una señal es una indicación generada por parte de un proceso para indicar a otro proceso o al sistema que se ha cumplido cierta condición. Así por ejemplo, cuando un proceso finaliza envía una señal al núcleo del sistema operativo para indicarle que ha acabado.

Cuando un proceso desea mandar una señal de interrupción a otro proceso la enviará a través del núcleo del sistema operativo, el cual la hará llegar al proceso destino. Es decir, el proceso emisor de la señal se la remite al núcleo, el único elemento que conoce y controla los diversos dispositivos y/o procesos existentes en el sistema, y por tanto, el único capaz de hacerla llegar a su receptor.

Al recibir una señal – el proceso receptor – por defecto, finaliza. Sin embargo, se puede especificar que se “capture” la señal, de forma que si es una señal específica, el proceso, en vez de finalizar realice cierta acción (o simplemente la ignore).

Un aspecto importante de las señales, es que constituyen un mecanismo de comunicación asíncrono, es decir, un proceso puede recibir una señal en cualquier momento de su ejecución, de manera que ésta puede verse abortada en cualquier instante. Por ello, si se desea capturar una señal, deberá hacerse cuanto antes, garantizando así que el proceso no finalizará al recibirla.

Existen diversos tipos de señales. Cada una de ellos tiene asociado un número de señal, de manera que cada número de señal especifica cierto suceso concreto. En el fichero de cabecera “signal.h” se asigna a cada una de estas señales un nombre simbólico que facilita su manejo en tiempo de programación. Las señales más utilizadas en Linux son las siguientes:

Nº de señal	Nombre	Significado
1	SIGHUP	Se ha perdido la conexión con el terminal.
2	SIGINT	Pulsación de las teclas de interrupción (Ctrl-c).
3	SIGQUIT	Pulsación de las teclas de abandono (Ctrl-\\).
8	SIGFPE	Excepción de coma flotante (Ej: División por 0).
9	SIGKILL	Matar al proceso receptor.
14	SIGALRM	Señal de alarma, producida al expirar un contador.
15	SIGTERM	Señal correspondiente por defecto al comando kill.

Tabla 7: Señales más utilizadas en Linux.

La señal de desconexión – SIGHUP – se utiliza para indicar a un proceso que el sistema ha perdido la conexión con el terminal. El núcleo la genera al detectar que el cable que va del terminal al computador ha fallado o bien cuando la conexión telefónica (vía módem) ha fallado. En algunas versiones de Unix se genera también esta señal para indicar que el usuario se ha desconectado del terminal.

La señal de interrupción – SIGINT – se genera cuando se presionan teclas de interrupción, como <Ctrl-C> o <Supr>. Estas teclas o combinaciones de teclas varían de sistema a sistema pero en todos ellos siempre está disponible al menos una de ellas.

En Linux existe también la señal de abandono – SIGQUIT – , que se genera desde el teclado al presionar la combinación de teclas <Ctrl-\\> provocando que el proceso aborta la ejecución antes de finalizar.

Las señales que implican matar al proceso receptor – SIGKILL y SIGTERM – se generan haciendo uso de la llamada al sistema *kill*. La diferencia entre ellas radica en que SIGTERM se puede capturar, mientras que SIGKILL no puede ser capturada, y siempre provoca la terminación del proceso receptor.

La señal SIGALRM se dispara cuando un temporizador, puesto en marcha mediante la llamada al sistema *alarm*, llega a 0. Generalmente, se utiliza para dotar al proceso de

cierto nivel de sincronismo, pues es posible disparar esta señal transcurrido cierto tiempo, el indicado como parámetro en la llamada al sistema *alarm*.

5.4.1 LAS FUNCIONES *pause()* Y *alarm()*

La función *pause()*

Existe en Linux otra función relacionada con las señales, la función *pause()*, que permite a un programa hacer uso de la llamada al sistema *pause*. Esta llamada, bloquea al proceso que la ejecuta hasta que éste recibe una señal, siempre y cuando la señal **no esté ignorada**.

El propósito fundamental de la existencia de esta función es lograr la cooperación entre varios procesos, deteniendo uno de ellos hasta que el otro lo “reactive” mediante el envío de una señal (que deberá, lógicamente, estar capturada).

El formato de esta señal es:

```
# include <unistd.h>

int pause (void);
```



La función *alarm()*

En Linux cada proceso tiene un temporizador virtual asociado, de manera que es posible establecer restricciones temporales durante su ejecución. Estas condiciones temporales se establecen mediante la función *alarm()*, que hace uso de la llamada *alarm*.

Su prototipo es:

```
# include <unistd.h>

long alarm (long secs);
```

La ejecución de esta función provoca que un proceso se mande a sí mismo una señal SIGALRM tras un determinado número de segundos (pasado como parámetro a la función *alarm*).

Puesto que cada proceso en Linux tiene asociado un único contador, cada proceso en Linux puede tener como máximo una única alarma pendiente. Además, la invocación a esta función con el parámetro cero (0) provoca la anulación de una potencial temporización que el proceso que la ejecuta pueda tener en marcha.

La invocación a la función *alarm* anula cualquier temporización anterior establecida, y de hecho, el valor que devuelve la ejecución de esta señal será el valor que tenía el

temporizador en el momento de su invocación. El valor por defecto de este temporizador es cero.

5.4.2 CAPTURA DE LAS SEÑALES: LA ORDEN *trap* Y LA LLAMADA *signal*.

Para lograr que la acción por defecto de un proceso cuando recibe una señal (terminar inmediatamente) no se lleve a cabo se puede utilizar la orden *trap* dentro de un guión del *shell* y la llamada al sistema *signal* en tiempo de programación. En cualquiera de los dos casos se puede especificar otra acción diferente, como por ejemplo ignorar ciertas señales o ejecutar cierta orden.

La orden *trap*

El formato de la orden *trap* es el siguiente:

```
trap '[comandos opcionales separados por ;]' <números de señal>
```

La parte de comandos opcionales puede no estar presente, y cuando lo está, las órdenes que incluye se ejecutan siempre que el proceso recibe una de las señales especificadas. De cualquier manera, deben ponerse las comillas simples o dobles incluso cuando no se especifique ningún comando; sin ellas, la sentencia *trap* reinicializa las señales especificadas.

Para capturar más de una señal, basta con indicar los números correspondientes a las mismas separados simplemente mediante espacios en blanco.

Ejemplo 1:

```
trap 'echo matado por una señal; exit' 15
```

En este caso, si el proceso recibe la orden *term* (la número 15) se ejecuta el comando *echo*, mostrando por pantalla el mensaje “matado por una señal”. A continuación se ejecuta el comando *exit*, que hace que el guión finalice.

Ejemplo 2:

```
trap '' 2 15
```

Si el proceso recibe la señal de interrupción (número 2) o la de terminación (número 15), simplemente las ignora y continúa el guión, ya que no se ha especificado ninguna orden o comando en la parte opcional.

REINICIALIZACIÓN O ANULACIÓN DE LAS CAPTURAS

La aparición de una sentencia *trap* dentro de un guión varía las consecuencias de las señales que recibe un determinado proceso. Si se utiliza la sentencia *trap* sin la parte de órdenes opcional o sin las comillas pertinentes, se reinician las señales afectadas haciendo que vuelvan a aplicarse las acciones por defecto.

De esta forma, es posible mantener la captura de alguna(s) señal(es) en determinados fragmentos de un guión, permitiendo que esta captura no tenga efecto para otros.

Ejemplo 1:

```
trap " " 2 3 15
```

```
<Fragmento a>
```

```
trap 2 3 15
```

```
<Fragmento b>
```

En el momento de ejecución de la primera sentencia, quedarán ignoradas las señales de interrupción, abandono y terminación de manera que incluso aunque se presionen las teclas correspondientes el guión continuará su ejecución. Esta captura estará vigente durante todo el "Fragmento a". No obstante, cuando se ejecute la segunda sentencia *trap* las señales indicadas quedarán reiniciadas y por tanto se les aplicará el tratamiento por defecto durante el "Fragmento b".

La llamada signal

El formato de la llamada al sistema *signal* es el siguiente:

```
void (*signal (int signum, void (*handler)(int)))(int);
```

siendo *signum* el número de la señal a capturar, y *handler* un puntero a una función en la que se especifica la acción a realizar cuando se reciba la señal. Esta función debe recibir siempre un entero (que se corresponde o identifica a la señal sobre la que se aplica) y no devolver nada (void). La propia función *signal* devuelve un puntero a la función apuntada por *handler*.

Una vez que se ha capturado una señal de cierto tipo, este captura es válida únicamente para la primera señal de ese tipo que sea recibida por el proceso. Por ello, habitualmente se suele incluir de nuevo la llamada *signal* dentro de la función apuntada por *handler*.

Ejemplo 1:

```
#include <stdio.h>

#include <signal.h>

#include <unistd.h>

int contador_ctrl_c = 0;

void (* old_handler)(int);

void ctrl_c(int);

main()
{
    int c;

    old_handler = signal (SIGINT, ctrl_c);
    while ((c = getchar()) != '\n');
    printf("Contador Ctrl-c = %n", contador_ctrl_c);

    (void) signal(SIGINT, old_handler);

    for (;;)

}

void ctrl_c (int sigsum)
{
    (void) signal (SIGINT, old_handler);

    ++ contador_ctrl_c;

}
```

Este programa simplemente lee caracteres de teclado hasta que se teclee <ctrl-\>, es decir, se salte de línea. Después se mete en un bucle infinito del cual no se puede salir mediante la combinación de teclas <Ctrl-c> al estar capturada la señal que ésta produce.

Ejemplo 2:

```

void trapper (int);

main()
{
    int i;
    for (i = 0; i<32; ++i)
        signal (i, trapper);

    for (;;)
    }

void trapper(int sig)
{
    signal (sig, trapper);
    printf ("Recibida la señal : %d\n", sig);
}

```

Esta función captura todas las señales susceptibles de ser capturadas (desde la número 0 hasta la número 31), mostrando un mensaje por pantalla cada vez que se reciba alguna de ellas.

La función *signal* también admite como segundo parámetro la constante SIG_IGN que implica ignorar la señal especificada como primer parámetro. Ignorar una señal conlleva que no se aplique al proceso el tratamiento por defecto, es decir, el proceso no es eliminado, pero no implica el desbloqueo del mismo si se encuentra retenido en espera de la recepción de una señal (tras haber ejecutado una llamada *pause*).

Otro valor posible para el segundo parámetro de esta función es la constante SIG_DFL que conlleva la aplicación del tratamiento por defecto al recibir una señal, y que suele utilizarse para anular una captura llevada a cabo mediante una invocación anterior a la función *signal*.

Las señales y el editor joe.

El editor *joe*, de uso bastante común en las instalaciones Linux puede ser origen de la generación de señales de tipo SIGHUP. De hecho, si éste editor quedara abierto en una sesión durante cierto tiempo sin que se realice sobre él ningún tipo de interacción (por ejemplo porque una persona que estaba trabajando con él abandona la máquina sin cerrar el editor), el sistema generará una señal SIGHUP dirigida hacia el propio *joe*.

No obstante, este editor tiene capturada esta señal, de manera que su recepción no provoque la “muerte” inmediata del proceso al recibirla. Así, el editor, al recibir la señal, salvará el contenido del fichero abierto en el momento de la recepción del señal en un fichero de nombre *DEAD_JOE*, y solamente después se cerrará.

5.4.3 ENVÍO DE SEÑALES ENTRE PROCESOS: LA LLAMADA KILL.

La forma más natural de generación de señales en un sistema suele ser el cambio de estado de algún proceso, un error detectado por hardware – como una excepción de coma flotante – o la pulsación de determinada combinación de teclas por parte del usuario, y la expiración de un temporizador activado mediante la llamada *alarm*.

Además de estos casos, es posible que un proceso envíe deliberadamente una señal a otro proceso, siempre y cuando cuente con los permisos oportunos (es decir, siempre que se trata de procesos de igual *uid*). Este envío de señales se logra mediante la llamada *kill*:

```
# include <signal.h>

# include <sys/types.h>

int kill (pid_t pid, int sig);
```

El primer parámetro representa al proceso que recibirá la señal, mientras que el segundo especificará la señal a enviar. Nótese que a pesar del nombre de la función, su efecto es exclusivamente el **envío** de una señal de un proceso a otro. Si como resultado de este envío muriera algún proceso, esta “muerte” solamente sería atribuible al tratamiento por defecto de la señal recibida.

Si se especifica como primer parámetro el valor cero (0), se enviará la señal especificada como segundo parámetro a todos los procesos cuyo identificador de grupo coincida con el identificador de grupo del proceso que invoca a la función.

Linux puede además hacer uso de la función *kill* con un significado especial, indicando como primer parámetro el valor -1 . Esto provocará que se envíe la señal especificada a todos los procesos existentes en el sistema excepto el proceso *init* y el proceso que invoca a la función.

También es posible comprobar el correcto funcionamiento de esta función comprobando el valor que devuelve, pues si éste es -1 , la función *kill* no habrá podido enviar la señal a alguno de los procesos especificados como segundo parámetro.

