SISTEMAS OPERATIVOS

Desarrollo de aplicaciones multiproceso en entornos POSIX

V4.2, 29/septiembre/2021

Índice

1 Introducción

En este documento se estudian algunos aspectos de programación que son de interés para desarrollar aplicaciones multiproceso en un sistema operativo de tipo POSIX. Se encuentra estructurado en varios bloques con el fin de facilitar su estudio, los cuales se comentan brevemente a continuación.

El primer bloque proporciona diversa información de carácter general sobre el uso de funciones y llamadas al sistema en los entornos POSIX.

El segundo bloque aborda el manejo de ficheros. Los ficheros se utilizan en todo tipo de aplicaciones, no solo las aplicaciones multiproceso.

El tercer bloque hace referencia al concepto de proceso, indicando algunos de sus atributos asociados. También estudia, desde el punto de vista de la programación, el modo de crear nuevos procesos, suspender temporalmente su ejecución, o hacerlos finalizar.

Por último, se estudian algunos de los métodos de comunicación entre procesos, centrándonos solo en aquellos métodos concretos que utilizaremos en el laboratorio.

En este documento se proporcionan explicaciones resumidas de las diferentes funciones. Si es necesario ampliar información, la primera opción sería consultar la correspondiente página de manual.

2 Generalidades

2.1 Concepto de llamada al sistema

El *núcleo* de un sistema operativo proporciona un conjunto limitado de puntos de acceso directo, a través de los cuales un proceso puede utilizar servicios que le proporciona dicho *núcleo*. A estos puntos se los denomina *llamadas al sistema*.

La forma de implementar las llamadas al sistema puede variar de un sistema a otro. Los sistemas de tipo POSIX disponen de una biblioteca C estándar que proporciona a los programadores de C una interfaz estable, en la cual cada llamada al sistema aparece como una función C normal.

2.2 Gestión de errores

Por lo general, en caso de error las llamadas al sistema y muchas funciones de biblioteca devuelven el valor `-1`, `NULL` o algún otro valor especial indicando tal circunstancia. Además, para que el programador pueda conocer qué clase de error ha ocurrido, el sistema deposita un código numérico en una variable global denominada `errno`. Las páginas de manual de las llamadas al sistema tienen unos apartados llamados “RETURN VALUES” y

"ERRORS" en los que se indican los posibles códigos de error que puede devolver la llamada al sistema. En la página de manual `intro(2)` hay un listado completo de los posibles valores que puede tomar la variable `errno`.

Para usar esta variable debe cargarse el fichero de cabecera errno.h.

```
#include <errno.h>
```

En el fichero de cabecera errno.h se definen las constantes correspondientes a todos los posibles códigos (tales como EPERM, EAGAIN, ENOENT…) Obviamente, siempre han de usarse estas constantes y no los valores numéricos.

No siempre que una llamada devuelve -1 se ha producido realmente un error. Por ejemplo, algunas llamadas al sistema (como las que están relacionadas con la E/S a través de la red o con espera mediante temporizadores) pueden devolver -1 y el valor EINTR en la variable `errno` para indicar que la espera o la operación de E/S ha sido terminada prematuramente al recibir un señal.

Dado que `errno` es una variable entera, es poco útil decirle al usuario que se ha producido el "error 33". Por ello, existe la función de biblioteca `perror()`, la cual escribe por stderr un mensaje explicativo sobre el valor actual de la variable `errno`. A ese mensaje se le puede anteponer un texto proporcionado por el programa y que suele usarse para contextualizar el error. Esta contextualización es importante, porque tampoco sería muy útil que el programa terminara diciendo "error al abrir un fichero" si no se sabe a qué fichero se refiere el error. La función `perror()` tiene el siguiente formato:

```
#include <stdio.h>

void perror(const char *prefijo);
```

El mensaje que genera y muestra en el canal de error estándar (stderr) tiene el siguiente formato: en primer lugar `perror()` imprime el `prefijo` que recibe, a continuación imprime un signo ":", y después imprime el mensaje[1] de error correspondiente al valor actual de `errno`, seguido de un retorno de carro.

Por último, señalar que solo se debe consultar el valor `errno` después de verificar que ha ocurrido un error en una llamada al sistema (y por supuesto, antes de llamar a cualquier otra), pues de lo contrario su valor resulta indefinido.

Puede ver ejemplos de uso de `perror()` en la mayoría de los ejemplos de este documento y en algún caso también se comprueba el valor de `errno`.

[1] En los sistemas utilizados en el laboratorio, este mensaje aparece en idioma inglés.

3 Manejo de ficheros

Los ficheros se pueden manejar o bien mediante las funciones de la biblioteca estándar de C (`fopen`, `fread`, `fwrite`, `fgets`, `fprintf`, `fscanf`, `fclose`, `fseek`, `fflush`, `remove`, etc.) o bien mediante funciones del estándar POSIX (`open`, `read`, `write`, `close`, `lseek`, `fstat`, `unlink`, `remove`, etc.), que en la mayoría de los sistemas de tipo UNIX son directamente llamadas al sistema.

Lo más recomendable es utilizar las funciones de la biblioteca estándar de C siempre que sea posible. Como regla genérica, observe que el nombre de casi todas comienza por "f", aunque también hay alguna función de POSIX cuyo nombre comienza por "f". La lista completa de funciones estándar para el manejo de ficheros está en la página de manual `stdio(3)`.

En los siguientes apartados se exploran ambas opciones para el manejo de ficheros. Primero se presentan las funciones más importantes para las operaciones habituales y al final hay un apartado con varios ejemplos completos.

3.1 Conceptos de manejador y descriptor de fichero

Tanto la biblioteca estándar de C como la de POSIX tienen un concepto muy similar: el *manejador* (en nomenclatura de la biblioteca estándar de C) y el *descriptor* (en nomencluatura de POSIX). Estos son unas estructuras de datos que se obtienen al abrir un canal de E/S (frecuentemente un fichero, pero puede ser una tubería, un *socket*, u otros dispositivos de E/S) y que se utilizan en posteriores funciones para referirse a dicho canal.

En el caso de la biblioteca estándar de C, el manejador es del tipo `FILE*` y en el caso de POSIX el descriptor es un simple entero.

Como norma general, todo proceso tiene abiertos de inicio tres manejadores/descriptores estándar: el stdin/0 corresponde a la entrada estándar (normalmente el teclado, salvo que se haya redirigido), el stdout/1 corresponde a la salida estándar (normalmente la pantalla, salvo que se haya redirigido) y el stderr/2 corresponde a la salida de error estándar (también la pantalla, salvo que se haya redirigido).

Si tiene un manejador de fichero y desea obtener su descriptor, puede usar la función `fileno()`. Si tiene un descriptor de fichero y desea obtener un manejador para él, puede usar la función `fdopen()`. Esto es muy útil cuando se ha abierto un fichero en uno de los entornos pero hay alguna operación para la que se necesita utilizar el otro. Puede ver un ejemplo en el apartado 5.2.2.

3.2 Apertura de ficheros

Para poder usar cualquier canal de E/S, lo primero que hay que hacer es abrirlo. Si el canal no existe previamente, la operación de apertura puede usarse para crearlo.

La función de la biblioteca estándar de C para abrir ficheros es:

```
#include <stdio.h>

FILE* fopen(const char *nombre, const char *modo);
```

siendo:

- `nombre`, el nombre del fichero a abrir (si no se especifica el *path* completo es un nombre relativo al directorio actual).
- `modo`, un conjunto de modificadores que indican el tipo de apertura que se desea realizar. Son letras que se pueden combinar y cuyo uso de ilustra en una tabla posterior.

La función POSIX es:

```
#include <fcntl.h>

int open(const char *nombre, int flags, ...);
```

siendo:

- `nombre`, el nombre del fichero a abrir (si no se especifica el *path* completo es un nombre relativo al directorio actual).
- `flags`, un conjunto de modificadores que indican el tipo de apertura que se desea realizar. Se pueden combinar mediante el operador OR binario "`|`". Su uso se ilustra en una tabla posterior.
- Los puntos suspensivos indican que hay parámetros opcionales.

La siguiente tabla presenta las operaciones de apertura más habituales e indica qué modificadores habría que usar tanto para `fopen()` como para `open()`. En algún caso, la operación no se puede hacer, lo cual se indica mediante el acrónimo "N.D." (no disponible).

Operación	**fopen**	**open**[2]
Abrir un fichero para solo lectura. Si no existe, se devuelve un error.	r	O_RDONLY
Abrir un fichero para lectura y escritura, manteniendo su contenido actual. Si no existe, se devuelve un error.	r+	O_RDWR
Abrir un fichero para lectura y escritura, descartando su contenido actual.	N.D.	O_RDWR | O_TRUNC

[2] Obsérvese que si hay varios modificadores hay que combinarlos con el operador |

Abrir un fichero para escritura. Si no existe, crearlo. Si existe, descartar su contenido. Es decir, en cualquier caso se comienza a escribir en un fichero vacío.	W	O_WRONLY | O_TRUNC | O_CREAT Ver comentario sobre permisos.
Abrir un fichero para escritura a partir del final. Si no existe, crearlo. Es decir, el contenido actual se mantiene y los nuevos datos se añaden al final del fichero.	A	O_WRONLY | O_APPEND | O_CREAT Ver comentario sobre permisos.
Crear un fichero para escribir, devolviendo error si ya existe.	Wx	O_WRONLY | O_CREAT | O_EXCL Ver comentarios sobre permisos y exclusión mutua.

Cuando se usa el modificador `O_CREAT` en la función `open()`, entonces hay que especificar un tercer parámetro de la función, que son los permisos con los que se desea crear el fichero. Esos permisos se indican con notación octal y hay que tener en cuenta que el sistema los ajustará en función del valor actual de `umask()`. En la biblioteca estándar de C no hay posibilidad de especificar los permisos y es como si se hubiera especificado 0666 en la función POSIX.

3.3 Uso de ficheros para sincronización entre procesos

El último de los ejemplos de apertura de ficheros de la tabla anterior muestra un caso de uso del sistema de ficheros como un posible mecanismo de exclusión mutua entre procesos. La forma de utilizarlo sería esta: antes de entrar en la sección crítica, cada proceso intenta crear un fichero con un nombre predeterminado (el mismo para todos los procesos) y el modificador de exclusión mutua. Si el sistema le devuelve error, es porque algún otro proceso lo ha creado antes que él y este proceso tiene que esperar y volver a intentarlo más tarde. Obviamente, cuando un proceso termina su sección crítica, tiene que borrar el fichero para que otros procesos puedan entrar.

El fichero de exclusión mutua puede estar vacío o, si es necesario, puede contener alguna información relevante, como el PID del proceso que lo ha creado. En este último caso, habría que tener en cuenta que podría haber situaciones de carrera entre un proceso que va a guardar información en el fichero y otros procesos que la intentan consultar.

La espera para volver a intentar la entrada a la sección crítica puede hacerse de múltiples formas. Una forma sencilla, aunque no muy elegante, es esperar un tiempo fijo (por ejemplo, 1 segundo) y volver a intentarlo.

Obsérvese que si el proceso que creó el fichero termina su ejecución sin borrarlo, el resto de procesos se quedarán esperando eternamente. Por ejemplo, si cabe la posibilidad de que al proceso le envíen la señal SIGTERM para matarlo, el programa debería capturar dicha señal y tomar las medidas necesarias para borrar el fichero de exclusión mutua antes de suicidarse.

Por otro lado, con este mecanismo no se garantiza ningún orden particular en el acceso a la sección crítica. Es decir, puede suceder que un proceso que acaba de ponerse a la espera acabe entrando a la sección crítica antes que otros que llevan esperando mucho tiempo.

3.4 Uso de ficheros temporales

Cuando varios procesos que se ejecutan concurrentemente necesitan crear un fichero temporal propio en alguna fase del programa, tienen que asegurarse de que el fichero de cada proceso es exclusivamente suyo (es decir, dos procesos no han creado el mismo fichero temporal y se están machacando mutuamente). Por "fichero temporal" se entiende un fichero en el que el proceso guarda datos durante su ejecución, pero esos datos no requieren persistencia y han de eliminarse al terminar la ejecución.

Una primera forma de crear estos ficheros temporales es que cada proceso se invente un nombre aleatorio para su fichero. Aunque esta forma puede llegar a funcionar, siempre se correrá el riesgo de que dos procesos casualmente se inventen el mismo nombre o se inventen un nombre que coincida con un fichero ya existente.

Otra forma mejor de crear estos ficheros temporales es que cada proceso se invente un nombre aleatorio para el fichero e intente crearlo con el modificador de exclusión mutua. Si la creación falla, es que ese nombre ya lo está usando algún otro proceso y hay que volver a intentarlo, inventándose otro nombre. Precisamente ya existen funciones, tanto de la biblioteca estándar de C como de POSIX, que realizan este procedimiento, de manera sencilla para el programador. En concreto son:

La función `tmpfile()` de la biblioteca estándar de C devuelve un manejador a un fichero nuevo en el directorio `/tmp`, que se garantiza no existía antes, abierto en modo de lectura y escritura y con permisos 0600. Además, el fichero temporal se eliminará automáticamente cuando el programa lo cierre o termine.

La función `mkstemp()` de POSIX es similar a la anterior, solo que permite especificar un patrón para el nombre del fichero temporal y el fichero temporal no se elimina automáticamente. Para más detalles, consulte la página de manual.

Incluso aunque sea un único proceso el que va a crear el fichero temporal y no se prevea la posibilidad de colisión con otros procesos concurrentes, se aconseja el uso de estas funciones para crear ficheros temporales, especialmente si el programa maneja datos confidenciales o se ejecuta con privilegios. Ello es para evitar que un proceso malicioso de la misma máquina adivine qué nombre de fichero temporal va a usar este proceso y se adelante creándolo, lo cual le permitiría controlar su contenido.

3.5 Lectura de datos

Se pueden leer datos de un descriptor utilizando la llamada `read()` de POSIX:

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbyte);
```

Esta función lee un número `nbyte` de *bytes* del fichero identificado por el descriptor `fd` y los deposita en el *buffer* cuya dirección indica `buf`.

El número de bytes a leer, `nbyte`, debe ser siempre un número positivo. El descriptor de fichero `fd` debe haber sido abierto con anterioridad.

La función `read()` devuelve el número de *bytes* que realmente ha leído y que ha colocado en el *buffer*. Este valor puede ser menor que la cantidad solicitada si no había datos suficientes (ver comentarios en los siguientes párrafos). Obviamente, también puede devolver -1 en caso de anomalía y señalizar la causa en `errno`.

Si el descriptor corresponde a un fichero, `read()` retornará rápidamente (las lecturas de disco suelen atenderse en unas decenas de milisegundos a lo sumo, o incluso de inmediato si los datos ya estaban en un *buffer*) y el valor devuelto será igual al número de *bytes* solicitado. Si quedaban menos *bytes* por leer que los solicitados, entonces `read()` leerá y devolverá solo los *bytes* realmente existentes. Si ya estábamos al final del fichero, devolverá cero.

Si el descriptor corresponde a un elemento de comunicación entre procesos, entonces hay varios casos:

- Si hay datos pendientes de leer, `read()` retornará inmediatamente con la cantidad solicitada o quizá con una cantidad menor, si la cantidad de datos disponibles era menor que la solicitada.

- Si no hay datos pendientes de leer, `read()` se quedará bloqueada hasta que llegue algo. Obsérvese que esta espera puede tener una duración ilimitada.

- Si el otro extremo del canal de comunicación está cerrado, `read()` devuelve cero inmediatamente. Esto significa que el proceso que gestionaba el otro extremo del canal ha terminado o ha dado por concluida la comunicación.

- En algunas circunstancias, `read()` puede devolver -1 y poner un valor en `errno` indicando que la operación no se ha podido realizar, aunque no siempre debe considerarse un error. Por ejemplo, un valor de EINTR o EAGAIN en `errno` no es indicativo de error.

Tenga en cuenta que `read()` es una función de bajo nivel y que únicamente entiende de leer un bloque de bytes. Es adecuada para leer datos binarios, pero si lo que realmente se

desea leer es cadenas de texto, es mucho más sencillo hacerlo con alguna función de la biblioteca estándar de C.

Así mismo, tenga en cuenta que si en un canal de comunicaciones el proceso que lee y el proceso que escribe manejan tamaños diferentes en las operaciones, se leerán mensajes a medias y eso es más difícil de procesar.

Para leer datos con la biblioteca estándar de C hay dos opciones:

- Para leer texto, se pueden usar las funciones `fgets()` o `fscanf()`. La primera está pensada para leer líneas de texto y la segunda está pensada para leer fundamentalmente números.

  La función `fgets()` no elimina la marca de fin de línea durante la lectura, por lo que la cadena de caracteres obtenida contendrá dicha marca. Dependiendo del sistema operativo en el que se haya creado el fichero de texto, la marca difiere. Lo habitual, sin ser ley absoluta, es un carácter '\n' en POSIX y Mac OS, un carácter '\r' en los Mac OS antiguos y '\r\n' (dos caracteres) en los Windows, aunque puede ocurrir que ni siquiera venga una marca, como cuando la última línea de un fichero de texto no la lleva.

  En la mayoría de los casos el programa no está interesado en esas marcas y ha de eliminarlas antes de procesar cada línea. Una forma típica es buscar la marca al final de la cadena y machacarla con un '\0', lo cual tiene un poco de complejidad si el programa pretende adaptarse a todos los posibles escenarios.

  `fgets()` devuelve NULL si se ha alcanzado el final del fichero y también en caso de algún error de E/S. Para más detalles, consulte la página del manual.

  La función `fscanf()` permite leer números en diversos formatos, aunque nuevamente pueden surgir dificultades según cómo vengan representados los números. Por ejemplo, un problema típico es que el separador de decimales puede ser un punto o una coma, dependiendo de la configuración regional con que se hayan generado los datos.

  El resultado de `fscanf()` permite saber si se pudo leer lo esperado o si hubo algún error, se alcanzó el final del fichero, etc. Para más detalles, consulte la página del manual.

  Otras formas de leer números a partir de cadenas de texto es aplicar funciones como `atoi()`, `strtol()` o `sscanf()`.

- Para leer datos binarios, se usa la función `fread()`. Sus argumentos son similares a los de `read()`. Para más detalles, consulte la página de manual.

3.6 Escritura de datos

Se pueden escribir datos en un descriptor utilizando la llamada `write()` de POSIX:

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t nbyte);
```

Esta función intenta escribir `nbyte` *bytes* desde el *buffer* apuntado por `buf` al fichero especificado por el descriptor `fd`.

El número de *bytes* `nbyte` debe ser siempre un número positivo. El descriptor de fichero `fd` debe haber sido abierto con anterioridad.

La función `write()` devuelve el número de *bytes* que realmente ha escrito. En descriptores que no correspondan a ficheros, este valor puede ser menor que la cantidad solicitada (ver comentarios en los siguientes párrafos). Obviamente, también puede devolver -1 en caso de anomalía y señalizar la causa en `errno`.

Si el descriptor corresponde a un fichero, `write()` retornará rápidamente (habitualmente retornará de inmediato, pues el sistema operativo suele hacer *buffering* de los datos escritos) y el valor devuelto será igual al número de *bytes* solicitado. Si devuelve un valor inferior al número de *bytes* solicitados, es casi seguro que se debe a un problema (disco lleno, cuota de disco agotada…).

Al usarse *buffering* para las escrituras en disco, lo habitual es que cuando `write()` retorna los datos continúen en memoria y cabe la posibilidad de que se pierdan si hubiera una caída abrupta del sistema (como un apagón eléctrico). Si la aplicación necesita estar segura de que los datos están en almacenamiento persistente, entonces debe usar `fsync()` a continuación, pero esto causa que el rendimiento de las escrituras se reduzca notablemente.

Si el descriptor corresponde a un elemento de comunicación entre procesos, entonces hay varios casos:

- Si el canal no admite más datos en este momento, por ejemplo porque el *buffer* de salida está lleno o porque ha entrado en acción algún mecanismo de control de congestión, `write()` se queda bloqueada hasta que sea posible escribir. Obsérvese que esta espera puede tener una duración ilimitada.

- Si el otro extremo del canal de comunicación está cerrado, el proceso recibe una señal SIGPIPE, la cual por defecto hace que termine abruptamente. Si la señal SIGPIPE se ignoró al principio del programa (ver apartado 0), entonces `write()` devolverá -1 y `errno` valdrá EPIPE.

- En algunas ocasiones puede que `write()` no escriba todos los datos, por ejemplo porque solo una parte cabían en un *buffer* de salida. En ese caso, devuelve la cantidad realmente escrita y el programa tendrá que intentar escribir el resto en posteriores llamadas a `write()`.

Tenga en cuenta que `write()` es una función de bajo nivel y que únicamente entiende de escritir un bloque de bytes. Es adecuada para escribir datos binarios, pero si lo que realmente se desea escribir es cadenas de texto, es mucho más sencillo hacerlo con alguna función de la biblioteca estándar de C.

Para escribir datos con la biblioteca estándar de C hay dos opciones:

- Para escribir texto, se pueden usar las funciones `fprintf()` o `fputs()`.
- Para escribir datos binarios, se usa la función `fwrite()`. Sus argumentos son similares a los de `write()`. Para más detalles, consulte la página de manual.

3.6.1 Escritura de mensajes por la salida estándar.

Al escribir mensajes por la salida estándar (la pantalla), se recomienda terminar todas las líneas con el carácter de fin de línea (`\n`) o bien usar la función `fflush()` después de realizar la escritura.

El motivo es que las funciones de la biblioteca estándar de C hacen *buffering* de los caracteres y solo vacían el *buffer*, y por tanto lo envían a la pantalla, si este se llena, si se invoca a una función de entrada o bien se recibe un carácter de fin de línea.

3.7 Acceso directo a posiciones de un fichero

Cuando se trabaja con ficheros, existe el concepto de posición actual en el fichero (también se usan las nomenclaturas “desplazamiento actual en el fichero” o “puntero del fichero”). Esta es la posición a partir de la cual se realizará la próxima operación de lectura o de escritura en el fichero y la unidad de medida es el *byte*. En los canales de comunicación entre procesos este concepto no tiene sentido, así que este apartado solo se aplica a ficheros.

Cuando se abre un fichero, la posición actual es cero (inicio del fichero), salvo que se haya abierto con el modo de escribir al final, en cuyo caso la posición actual sería justo después del último *byte* almacenado en el fichero. Cuando se leen o escriben datos, la posición actual avanza tantos *bytes* como se hayan leído o escrito.

También es posible modificar el puntero del fichero explícitamente para acceder de manera directa a una cierta posición. Si el fichero contiene registros de datos de tamaño fijo, entonces es trivial calcular el desplazamiento del enésimo registro de un fichero (basta con multiplicar el número del registro por el tamaño de cada registro). En otros casos quizá no sea factible calcular el desplazamiento de un registro dado. Por ejemplo, en un fichero de texto donde las líneas pueden tener longitudes diferentes, no se puede calcular cuál es el desplazamiento de la enésima línea.

En la biblioteca estándar de C existen dos funciones para consultar y cambiar la posición actual del fichero:

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

```
long ftell(FILE* f);
int fseek(FILE* f, long desplazamiento, int base);
```

La función `ftell()` devuelve la posición actual en el fichero.

La función `fseek()` se utiliza para cambiar la posición actual del fichero. La nueva posición puede especificarse de tres maneras, en función del valor del parámetro `base`:

- Si `base` es `SEEK_SET`, entonces el `desplazamiento` es relativo al inicio del fichero. En otras palabras, es un desplazamiento absoluto.
- Si `base` es `SEEK_CUR`, entonces el `desplazamiento` es relativo a la posición actual del fichero. Esto es útil para avanzar o retroceder, pero sin tener que calcular la posición absoluta.
- Si `base` es `SEEK_END`, entonces el `desplazamiento` es relativo al final del fichero. Esto suele utilizarse para ir directamente al final del fichero, si especificamos el valor cero en el parámetro `desplazamiento`.

Para averiguar el tamaño de un fichero, primero se usa `fseek()` para posicionarse al final y a continuación se usa `ftell()` para obtener la posición actual, que nos dará el tamaño del fichero en octetos.

En POSIX tanto la consulta como el cambio de posición se hacen con esta función:

```
#include <unistd.h>

off_t lseek(int fd, off_t desplazamiento, int base);
```

El significado de los parámetros `base` y `desplazamiento` es análogo al del caso de `fseek()`. La única salvedad es que el valor del desplazamiento es de tipo `off_t`, que en muchos sistemas es un tipo de datos de mayor tamaño que un `int` y que un `long`, por lo que no puede mezclarse alegremente con variables de esos tipos (habrá que declarar las variables de tipo `off_t` o bien usar moldeados correctamente).

Para la consulta de la posición actual, hay que tener en cuenta que `lseek()` siempre devuelve la posición resultante de cualquier operación. Por tanto, puede averiguarse la posición actual si `base` es `SEEK_CUR` y el desplazamiento es cero.

3.8 Cierre de ficheros

Los ficheros se cierran con la función `fclose()` de la biblioteca estándar de C:

```
#include <stdio.h>

int fclose(FILE* f);
```

Si el fichero se estaba manejando con las funciones de POSIX, entonces se cierran con `close()`:

```
#include <unistd.h>

int close(int fd);
```

Habitualmente estas funciones no fallan, pero en ficheros abiertos para escritura podrían quedar datos pendientes de volcar y justo esa última escritura podría dar error.

3.9 Redirección de un descriptor

Como es sabido, cuando se abre un fichero o una tubería se obtiene un descriptor que permite operar con él. Además, todo proceso tiene los descriptores estándar stdin, stdout y stderr, que normalmente están vinculados a los dispositivos del teclado y pantalla, y que utilizan las funciones típicas de E/S, como fgets, printf, etc.

En algunos programas es útil que un descriptor que hasta ese momento estaba conectado a un cierto fichero o dispositivo, pase a apuntar a otro fichero, tubería o dispositivo. De esta forma, el programa puede obtener o enviar datos que vienen o van a sitios alternativos, sin cambiar el código que los obtiene o los genera. Esta operación se denomina redirección de un descriptor y, por ejemplo, es muy habitual cuando se quiere guardar en un fichero la salida estándar que un programa normalmente sacaría por pantalla. Si se trabaja con el intérprete de mandatos, esto se consigue con los operadores < y >, pero en este apartado se va a explicar cómo redirigir descriptores desde un programa en C.

Es interesante tener en cuenta que la función POSIX `open()` devuelve siempre el descriptor de valor más bajo que esté libre. Por tanto, una posible forma de redirigir la salida estándar del proceso al fichero `salida.txt` sería la siguiente:

```
int fd;
close (fileno(stdout));
fd = open ("salida.txt",O_WRONLY | O_CREAT | O_TRUNC,DEFFILEMODE);
```

Si después de ejecutar estas sentencias `fd` es distinto de `fileno(stdout)`, no se ha conseguido el efecto deseado. Sin embargo, esta situación sólo puede darse en el caso de que el descriptor 0 estuviera libre o que el programa en ejecución sea multihilo y otro hilo haya reabierto el descriptor 1 entre la ejecución de la línea correspondiente a `close` y la ejecución de la línea correspondiente a `open`. La redirección de la entrada estándar se haría de forma análoga, pero teniendo en cuenta el modo en el que debe abrirse el fichero (lectura en lugar de escritura).

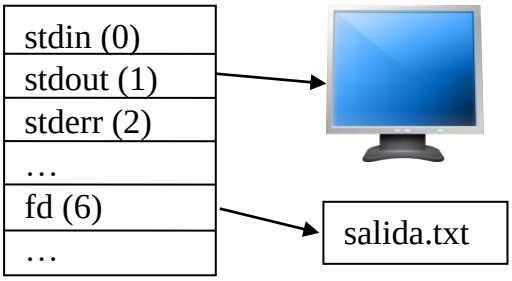
Otra forma de redirigir descriptores es usar las llamadas al sistema `dup()` y `dup2()`.

Este ejemplo muestra cómo crear el fichero “salida.txt” y redirigir a él la salida estándar usando `dup2()`. A partir de entonces, todo lo que el proceso genere con `printf()` o cualquier otra función que escribe a `stdout` aparecerá en el fichero “salida.txt”.

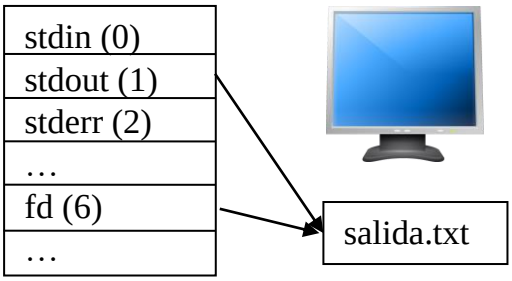
```
int fd;
fd = open ("salida.txt",O_WRONLY | O_CREAT | O_TRUNC,DEFFILEMODE);
if ( fd != -1 )
{  dup2 (fd, fileno(stdout));
   close (fd);
}
else
   /* tratar error */
```

A continuación se explica con mayor detalle el ejemplo anterior:

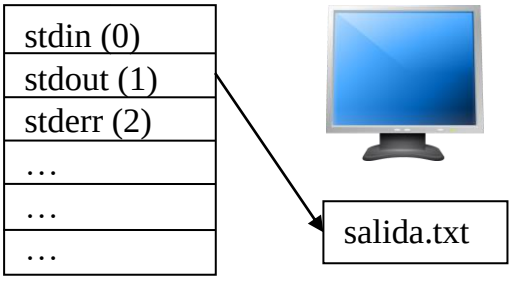
1. Abrir el fichero al que se quiere redirigir el descriptor. En este ejemplo el fichero se crea o, si ya existe, se descarta su contenido anterior, pero podría abrirse para añadir datos al final, con otros permisos, etc. En cualquier caso, se obtendrá un descriptor nuevo en `fd`, por ejemplo el número 6. Por su parte, `stdout` (descriptor 1), continúa apuntando a la pantalla, como es habitual.



2. Duplicar con `dup2()` el descriptor deseado en el descriptor nuevo. Esto implícitamente cierra la vinculación que tuviera antes el descriptor duplicado.



3. Cerrar el descriptor nuevo, puesto que ya ha sido duplicado y no se necesita tener dos descriptores para el mismo fichero.



También es posible redirigir tanto la salida estándar como el error estándar **al mismo fichero**. Para ello, basta con usar `dup2()` dos veces.

3.10 Consulta de los atributos de un fichero

Esta función POSIX se utiliza para acceder a la información del i-node que representa a un fichero como puede ser: propietario, permisos, tamaño, tipo de fichero (si es fichero o directorio), etc.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *nombre, struct stat *buf);
```

En la estructura `buf` se almacena información del fichero `nombre`. Mediante una serie de macros es posible consultar el valor de algunos campos de la estructura de tipo `stat`. Por ejemplo, uno de los campos de `stat` es `st_mode` y si queremos saber si un fichero es de tipo directorio se consultaría así:

```
struct stat buf;
stat ("un_fichero_o_directorio", &buf);
if ((buf.st_mode&S_IFDIR) != 0 )  //¡Es un directorio!
```

En la biblioteca estándar de C no hay ninguna función equivalente.

3.11 Borrado de ficheros

Para borrar un fichero puede usarse la siguiente función de la biblioteca estándar de C:

```
#include <stdio.h>

int remove(const char *nombre);
```

Otra alternativa es usar la llamada al sistema `unlink()` de POSIX.

3.12 Recorrido de directorios

Para recorrer todos los ficheros de un directorio pueden usarse las funciones descritas en las páginas de manual `directory(3)` y `dir(5)`. En el apartado 3.13.4 se muestra un ejemplo de uso.

3.13 Ejemplos de manejo de ficheros

A continuación se presentan varios ejemplos de programas o fragmentos de programas en los que se ilustra el uso de las principales funciones de manejo de ficheros.

3.13.1 Ficheros de texto

El siguiente programa lee un fichero de texto y copia en otro las líneas que tengan más de 30 caracteres.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sysexits.h>

#define MAX_LONGITUD 30

int main(int argc, char** argv)
{
   FILE *fich_entrada, *fich_salida;
   char *nombre_fich_entrada, *nombre_fich_salida;
   char linea[1000];

   // Toma los nombres de los ficheros como argumentos de línea de mandatos.
   if (argc != 3) {
      fprintf(stderr, "Uso: %s fich_entrada fich_salida\n", argv[0]);
      exit(EX_USAGE);
   }
   nombre_fich_entrada = argv[1];
   nombre_fich_salida = argv[2];
   // Abre el fichero de entrada para leer. Si da error, escribe un mensaje y
termina.
   if ((fich_entrada = fopen(nombre_fich_entrada, "r")) == NULL) {
      perror(nombre_fich_entrada);
      exit(EX_NOINPUT);
   }
   // Abre el fichero de salida para escribir. Si da error, escribe un mensaje y
termina.
   if ((fich_salida = fopen(nombre_fich_salida, "w")) == NULL) {
      perror(nombre_fich_salida);
      exit(EX_CANTCREAT);
   }
   // Se lee el fichero de entrada línea a línea y se escriben
   // en el de salida las que superan la longitud establecida. La
   // operación termina cuando fgets señaliza que se ha alcanzado
   // el final del fichero.
   // NOTA: si es importante detectar y procesar posibles errores
   // de E/S, habría que examinar con más detalle el resultado de
   // fgets, fputs y fclose.
   while (fgets(linea, sizeof(linea), fich_entrada) != NULL)
```

```
      if (strlen(linea) > MAX_LONGITUD)
         fputs(linea, fich_salida);
   fclose(fich_entrada);
   fclose(fich_salida);
   exit(EX_OK);
}
```

En este ejemplo ilustra una manera sencilla pero recomendable de aceptar argumentos de línea de mandatos. En primer lugar se comprueba que la cantidad de argumentos recibidos coincide con la esperada. Si no es así, se muestra un mensaje informativo sobre cómo usar el programa y se termina.

Como la operación de apertura de un fichero es el punto en el que es más probable que haya problemas, siempre debe comprobarse que la apertura ha tenido éxito. En caso contrario, se muestra un mensaje informativo haciendo uso de `perror()` y se termina el programa.

El bucle de lectura y escritura de líneas, así como el cierre de los ficheros, presuponen que todas las operaciones se realizarán sin error. Esto puede ser suficiente para un programa de prueba y en la mayoría de los ejemplos solo se comprueban los errores más probables para no hacer farragoso el ejemplo, pero en un programa de producción debe comprobarse el resultado de todas y cada una de las funciones que se usen.

En todos los puntos donde el programa puede terminar con `exit()` hay que especificar un código de retorno. En lugar de inventarse los códigos, en este ejemplo se usan los códigos recomendados en la página de manual `sysexits(3)`.

3.13.2 Ficheros binarios

Aquí se muestran dos programas de ejemplo relacionados entre sí. El primero genera un fichero binario con varios registros del tipo `Canal`.

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

/* Tipo de datos de los registros que se guardarán en el fichero. */
typedef struct {
   char banda;
   int frecuencia;
   int ancho;
   double potencia;
} Canal;

int main(int argc, char** argv)
{
   FILE* fichero;
   Canal canal;

   /* Crea el fichero. */
   if ((fichero = fopen("ej1.dat", "w")) == NULL) {
      perror("ej1.dat");
      exit(EX_CANTCREAT);
```

```
   }
   /* Escribe 10 registros en el fichero. */
   for (i = 0; i < 10; i++) {
      ... Aquí pondría datos en la variable canal ...
      if (fwrite(&canal, sizeof(Canal), 1, fichero) != 1) {
         perror("fwrite");
         exit(EX_IOERR);
      }
   }
   fclose(fichero);
   exit(EX_OK);
}
```

El segundo programa modifica el contenido de uno de los registros del fichero. El registro que hay que modificar, en concreto su posición dentro del fichero, se especifica como argumento de línea de mandatos del programa. A continuación se comentan algunos puntos interesantes sobre este ejemplo:

- El programa recibe un número de registro, pero la función fseek() precisa un desplazamiento en bytes. Por ese motivo, hay que multiplicar o dividir por el tamaño de cada registro para convertir entre ambas numeraciones.

- Al leer el registro, el puntero de la posición actual en el fichero avanza por el número de bytes leídos. Para escribir el registro modificado es necesario volver a posicionar el puntero en el desplazamiento del registro. En el ejemplo de abajo ello se hace repitiendo el mismo `fseek()` que se usó para leer el registro. Otra opción habría sido ejecutar la sentencia `fseek(fichero, -sizeof(Canal), SEEK_CUR);` para retroceder un registro desde la posición actual.

- Se han omitido todas las comprobaciones de errores de `fseek()`, `fread()` y `fwrite()` para no hacer farragoso el código. En un programa más elaborado deberían comprobarse al menos los casos más probables de fallo.

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

/* Estructura de los registros que se guardarán en el fichero. */
typedef struct {
   char banda;
   int frecuencia;
   int ancho;
   double potencia;
} Canal;

int main(int argc, char** argv)
{
   FILE* fichero;
   Canal canal;
   int num_canal, num_registros;

   /* Toma el número del canal que hay que modificar. */
```

```
   num_canal = atoi(argv[1]);
   /* Abre el fichero en lect/escr, preservando su contenido. */
   if ((fichero = fopen("ej1.dat", "r+")) == NULL) {
      perror("ej1.dat");
      exit(EX_NOINPUT);
   }
   /* Calcula el número total de registros que hay en el fichero. */
   fseek(fichero, 0, SEEK_END);
   num_registros = ftell(fichero) / sizeof(Canal);
   printf("Hay %d registros en el fichero.\n", num_registros);
   if (num_canal < num_registros) {
      /* Se posiciona en el número de canal especificado. */
      fseek(fichero, num_canal * sizeof(Canal), SEEK_SET);
      /* Lee el registro. */
      fread(&canal, sizeof(Canal), 1, fichero);
      /* Actualiza los datos (por ejemplo, dobla la potencia). */
      canal.potencia = canal.potencia * 2;
      /* Vuelve a posicionarse en el número de canal especificado. */
      fseek(fichero, num_canal * sizeof(Canal), SEEK_SET);
      /* Modifica el registro. */
      fwrite(&canal, sizeof(Canal), 1, fichero);
      printf("Canal %d modificado.\n", num_canal);
   }
   else
      printf("El número de canal indicado no es válido.");
   fclose(fichero);
   exit(EX_OK);
}
```

Observe que el acceso directo a un registro de un fichero solo es posible si se conoce el número de posición del registro dentro del fichero y además todos los registros tienen el mismo tamaño. En otro caso o bien hay que hacer una búsqueda secuencial, que es un procedimiento muy lento, o bien hay que usar otra técnica de gestión de ficheros más adecuada, lo cual queda fuera del alcance de este documento.

3.13.3 Exclusión mutua mediante un fichero

El siguiente ejemplo muestra cómo podría usarse un fichero para conseguir exclusión mutua entre varios procesos. El fragmento de código que se muestra debería ser ejecutado por cada proceso para entrar y salir de la sección crítica.

```
/* Todos los procesos deben usar el mismo fichero. Si no, no se */
/* sincronizarán entre sí. */
#define FICH_EXCLUSION "llave"

   FILE* fichero;

   /* Entrada a la sección crítica. Solo puede pasar cuando consiga */
   /* crear el fichero de exclusión mutua. */
   while ((fichero = fopen(FICH_EXCLUSION, "wx")) == NULL) {
      ... esperar, por ejemplo con sleep() ...
   }

   ... Aquí vendría la sección crítica ...

   /* Salida de la sección crítica. Cierra y elimina el fichero */
```

```
   /* para que otros procesos puedan entrar. */
   fclose(fichero);
   remove(FICH_EXCLUSION);
```

El código anterior supone que si `fopen()` falla es porque el fichero ya existe, pero en realidad podría fallar por otros motivos, como por ejemplo que el proceso no tenga permiso de escritura en el directorio donde pretende crear el fichero. Para ser más correctos, debería comprobarse el valor de `errno` cuando `fopen()` falla y esperar solo si es `EEXIST`. Si `fopen()` ha fallado por algún otro motivo, habrá que tratar adecuadamente la causa del error.

3.13.4 Recorrido de un directorio

Este ejemplo muestra cómo recorrer un directorio y calcular el tamaño total de todos los ficheros que hay en él.

Obsérvese que la función `readdir(3)` devuelve, entre otras cosas, el nombre de cada fichero encontrado, pero ese nombre es relativo al directorio que se está recorriendo, que puede ser diferente al directorio donde se ejecuta el programa. Si se quiere acceder a dichos ficheros es necesario componer un nombre válido desde el directorio donde se ejecuta el programa.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/param.h>
#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>
#include <unistd.h>
#include <dirent.h>
#include <string.h>

int main(int argc, char** argv)
{
   char* dir_entrada;
   DIR* dirp;
   struct dirent* fp;
   struct stat st;
   unsigned long tam_total;
   char nombre_fichero[MAXPATHLEN+1];

   /* Toma el nombre del directorio que hay que recorrer. */
   dir_entrada = argv[1];
   /* Abre el directorio. */
   if ((dirp = opendir(dir_entrada)) == NULL) {
      perror(dir_entrada);
      exit(EX_NOINPUT);
   }
   tam_total = 0;
   /* Recorre el directorio. */
   while ((fp = readdir(dirp)) != NULL) {
      /* Solo procesa los ficheros normales (regulares). */
      if (fp->d_type == DT_REG) {
         /* Para consultar los atributos del fichero tiene que componer
```

```
            su nombre con el directorio, pero se asegura de que no
            sobrepasa el tamaño de la variable usada para generarlo. */
         if (strlen(dir_entrada) + 1 + fp->d_namlen < sizeof(nombre_fichero)) {
            strcpy(nombre_fichero, dir_entrada);
            strcat(nombre_fichero, "/");
            strcat(nombre_fichero, fp->d_name);
            /* Otra alternativa para componer el nombre. */
            // sprintf(nombre_fichero, "%s/%s", dir_entrada, fp->d_name);
            /* Obtiene el tamaño del fichero y lo suma al total. */
            if (stat(nombre_fichero, &st) != -1)
               tam_total += st.st_size;
            else
               perror(nombre_fichero);
         }
      }
   }
   closedir(dirp);
   printf("Tamaño total: %ld octetos.\n", tam_total);
   exit(EX_OK);
}
```

4 Procesos

4.1 Concepto de proceso

Un proceso es básicamente un programa que se encuentra en ejecución. En los sistemas multiproceso varios procesos se encuentran simultáneamente en ejecución repartiéndose entre todos el tiempo de procesador, y utilizando conjunta y sincronizadamente todos los demás recursos disponibles.

4.2 Identificación de un proceso: concepto de PID, PPID, PGID.

Cada proceso tiene asociado un identificador único e inmutable (denominado *process ID* ó *PID*). Es un entero positivo, generalmente situado en el rango `0` a `99999`, que es asignado por el *núcleo* a cada nuevo proceso que se crea. Los procesos del sistema suelen tener PID bajos. Si se alcanza el máximo valor de PID, el sistema vuelve a asignar nuevos PID a partir de 0, saltándose los que todavía estén en uso.

Todo proceso desciende de algún otro proceso y se usa el acrónimo PPID (*parent PID*) para referirse al identificador del proceso padre. Si el proceso padre muere antes que el hijo, entonces el proceso hijo pasa a ser descendiente del proceso `init`, cuyo PID es siempre el 1. Los procesos también pueden desvincularse de su proceso padre (para convertirse en *demonios*) y en este caso también pasarán a tener a `init` como proceso padre.

Los procesos además pertenecen a grupos de procesos, los cuales se identifican mediante un número llamado PGID (*process group ID*) que normalmente coincide con el PID del primer proceso del grupo. Los grupos de procesos sirven para diferenciar entre tareas y así poder hacer envíos de señales en bloque a todos los procesos que forman parte de una tarea multiproceso y también para gestionar el foco del terminal entre varias tareas. Todo proceso nuevo pertenece inicialmente al mismo grupo de procesos que su padre, aunque luego puede

cambiarse de grupo bajo ciertas condiciones. Por ejemplo, el intérprete de mandatos cambia a un grupo de procesos nuevo a todos los procesos que componen cada mandato.

Las siguientes tres llamadas al sistema sirven para que un proceso averigüe su PID, su PPID y su PGID. La cuarta sirve para que un proceso averigüe el PGID de otro proceso:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
pid_t getpgrp(void);
pid_t getpgid(pid_t pid);
```

La siguiente llamada al sistema sirve para cambiar el PGID de un proceso. Para poder cambiarlo han de darse ciertas condiciones que se describen en la página de manual:

```
#include <sys/types.h>
#include <unistd.h>

pid_t setpgid(pid_t pid, pid_t pgrp);
```

4.3 Dueño de un proceso: concepto de UID.

En los sistemas multiusuario pueden existir varios usuarios distintos en el sistema, cada uno de ellos ejecutando un cierto número de procesos, y es necesario proteger a unos usuarios de otros.

Cada usuario del sistema tiene asociado un identificador único (denominado *User ID* ó *UID*). Se trata de un entero siempre positivo (estando reservado el UID `0` para el superusuario) que se asocia a los procesos que el usuario ejecuta.

Un proceso puede conocer el UID del usuario al que pertenece utilizando las llamadas al sistema `getuid()` y `geteuid()`, las cuales tienen el siguiente formato:

```
#include <sys/types.h>
#include <unistd.h>

uid_t getuid(void);
uid_t geteuid(void);
```

Estas llamadas devuelven el UID del usuario que creó el proceso (getuid) y el del usuario bajo el cual se ejecuta ahora el proceso (geteuid). Lo habitual es que ambos UID coincidan, aunque en el caso especial de los programas setuid pueden diferir.

4.4 Acceso a las variables de entorno

El intérprete de mandatos tiene definidas unas variables de entorno que sirven para configurar ciertos programas, como por ejemplo las variables HOME, PATH, TERM, LANG, EDITOR…

Desde el intérprete de mandatos se puede puede consultar su valor con el mandato `echo`. Por ejemplo para conocer el valor de la variable HOME se ejecutaría:

```
[estudiante@localhost ~]$ echo $HOME
/home/estudiante
[estudiante@localhost ~]$
```

Estas variables también pueden ser consultadas (o modificadas) desde procesos lanzados desde el intérprete de mandatos. Las llamadas al sistema que realizan estas operaciones son:

Obtener el valor de una variable: `char * getenv (const char *name)`

Modificar el valor de una variable: `char * setenv (const char *name, const char *value, int overwrite)`

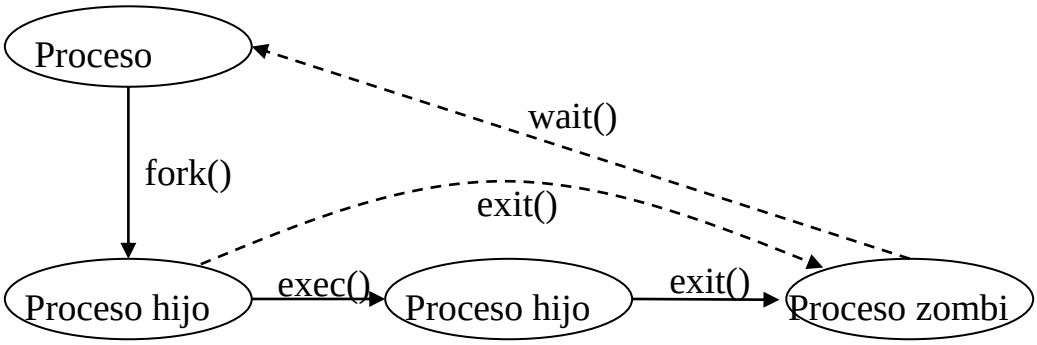
Ejemplo de uso:

```
#include <stdlib.h>
#include <stdio.h>
int main (int argc, char** argv){
   printf ("El valor de HOME es %s\n",getenv("HOME");
   setenv ("LANG","es_ES.ISO8859-15",1);
}
```

4.5 Creación y terminación de procesos

4.5.1 Ciclo de vida de los procesos

El siguiente diagrama ilustra el ciclo de vida genérico de los procesos:



Un proceso (al que llamamos padre) crea un proceso nuevo (al que llamamos hijo) mediante la llamada al sistema `fork()`. El proceso hijo es un clon del padre (ver más detalle en la descripción de `fork()`, más abajo), pero en todo caso es un proceso independiente, que avanzará concurrentemente con el resto de procesos.

Esta regla tiene una excepción lógica: el primer proceso que se crea en el sistema no puede ser generado a partir de otro proceso anterior, porque simplemente no existe otro anterior. Por ello, este primer proceso del sistema, que se denomina `init`, siempre con PID `1`, es creado por el *núcleo* de forma especial al arrancar el sistema. También hay algunos otros

procesos del núcleo que se crean espontáneamente durante el arranque, aunque su cantidad y finalidad depende de la versión concreta del sistema operativo que se utilice. Los procesos del núcleo suelen aparecer entre corchetes en los listados que genera el programa `ps`.

El proceso hijo realizará las acciones para las que fue creado, que puede que estén directamente en su propio código, o puede que se realicen por medio de un programa ejecutable diferente. En este último caso, el proceso hijo necesitará usar alguna llamada al sistema de la familia `exec()`.

En algún momento, el proceso hijo terminará su ejecución. Lo normal será que finalice voluntariamente mediante la función `exit()` o por terminar la función `main()`. No obstante, pueden darse otros casos (no ilustrados en la imagen), como que el hijo termine abruptamente por un error de programación (fallo de segmentación, división por cero…), o que termine porque le envíen una señal que lo mate inmediatamente. Sea como fuere la terminación, el proceso pasa al estado zombi, y en ese estado se mantendrá hasta que el padre reconozca su terminación con `wait()`.

Si el padre termina antes que el hijo o bien el padre termina sin reconocer la muerte de algunos de sus hijos, entonces sus procesos hijos aún existentes son heredados por el proceso del sistema `init` (PID 1), el cual reconocerá sus muertes cuando se produzcan. Con carácter general, en una aplicación multiproceso bien codificada el padre no debería terminar hasta que haya reconocido la muerte de todos sus hijos.

Mientras el hijo se ejecuta, el padre tiene dos alternativas: esperar síncronamente a que el hijo termine, o bien continuar haciendo otras cosas en paralelo con el hijo. Dependiendo de la aplicación, el programador optará por una u otra.

Si el programador opta por esperar síncronamente a que termine el hijo (en otras palabras, no tiene sentido que el padre avance hasta que termine el hijo), el padre invocará a alguna de las llamadas al sistema de la familia `wait()` y se quedará bloqueado hasta que el hijo finalice. Aunque podríamos hablar de una aplicación multiproceso, más bien sería una aplicación biproceso, puesto que como máximo habrá dos procesos en ejecución: el padre y el hijo. Además, habría escasas posibilidades de conflictos de concurrencia entre ambos procesos.

Si el programador opta por que ambos procesos avancen en paralelo, entonces el padre no puede quedarse bloqueado en `wait()`, sino que seguirá ejecutando otras acciones. Entre ellas podría estar la creación de más procesos hijos para lograr una verdadera aplicación multiproceso.

En una aplicación multiproceso, el padre probablemente necesitará manejar una tabla de procesos hijos que le permita gestionar el ciclo de vida de sus hijos. En dicha tabla, para cada proceso hijo se almacenará su PID y cualquier otro dato que el padre pueda requerir posteriormente para gestionarlo.

En una aplicación multiproceso, los procesos hijos pueden terminar mientras el padre está haciendo cualquier cosa. Como es importante que el padre reconozca la muerte de los hijos

lo antes posible, hay dos técnicas básicas: capturar la señal `SIGCHLD` o bien sondear frecuentemente a ver si ha terminado algún hijo. Con carácter general, capturar `SIGCHLD` suele ser más ágil, pero puede ser algo más complejo de implementar correctamente. En ambas técnicas se acaba usando alguna llamada al sistema de la familia `wait()` para reconocer la muerte del hijo.

En los siguientes apartados se explican en mayor detalle las operaciones más habituales con procesos.

4.5.2 Terminación de procesos: función exit()

Como es bien sabido, un programa en C puede finalizar su ejecución en cualquier momento llamando a la función de biblioteca `exit()`. Esta función recibe un argumento entero, que es el código de retorno que se desea comunicar al proceso padre.

La función `exit()` realiza diversas tareas importantes, como escribir todos los datos pendientes en los *buffers* de los ficheros abiertos, y finalmente invoca a la llamada al sistema `_exit()`, que es la que realmente indica al núcleo la finalización del proceso.

Esta función de biblioteca tiene el siguiente formato:

```
#include <stdlib.h>

void exit(int status);
```

El parámetro `status` se conoce como **código de terminación** y es un valor que el proceso que termina puede devolver al proceso padre que lo generó y que sirve como forma sencilla de dar información sobre el resultado del trabajo realizado. Este valor lo recibe el proceso padre al reconocer la muerte de su hijo.

Si no se usa esta llamada para terminar el programa, entonces el proceso termina implícitamente al terminar la función `main()` y el código de terminación devuelto al padre es el resultado de dicha función.

El significado del código de terminación es indiferente para el sistema operativo y queda enteramente a criterio del programador de la aplicación. Ahora bien, un criterio habitual es que los procesos que han podido realizar su trabajo sin problemas devuelvan al padre el valor de *status* `0`. En caso de producirse alguna circunstancia anormal o bien si quieren diferenciar entre distintas formas de terminar puede devolverse un valor de *status* en el rango `1` a `255`. Existen una serie de códigos genéricos recomendados para las situaciones más frecuentes. Para más detalles, véase la página de manual `sysexits(3)`.

Conviene recordar aquí que un proceso también puede terminar su ejecución involuntariamente por diversas causas. Uno de los casos más comunes es que el proceso hijo tenga un error de programación que le haya hecho acceder a una dirección de memoria inválida, en cuyo caso el núcleo genera una señal que aborta la ejecución del proceso inmediatamente. Otro caso común es que otro proceso le haya enviado una señal que

provoque su terminación inmediata, como SIGKILL o SIGTERM (si no está capturada o ignorada).

Para que la aplicación no falle, el padre tendrá que poder diferenciar entre la terminación voluntaria del hijo y los casos de terminación involuntaria (ver apartado 4.5.4). Por ejemplo, si el hijo terminó involuntariamente, no habrá usado `exit()` y el código de terminación que pueda obtener el padre será basura. Además, el proceso hijo podrá haber dejado cosas incompletas, como ficheros a medio componer, etc.

4.5.3 Creación de procesos: llamada al sistema fork()

Esta llamada al sistema es la única forma de generar un nuevo proceso en el sistema. Para ello, `fork()` *duplica* el proceso llamante. Es decir, crea un nuevo proceso que es una copia idéntica del proceso que ejecuta el `fork()`.

La copia realmente no es del todo exacta. Presenta una diferencia evidente[3]: el PID de la copia no es el mismo que el PID del original (ya que, como se ha estudiado, este identificador debe ser distinto para todos los procesos del sistema).

Siguiendo una nomenclatura ampliamente utilizada, en adelante nos referiremos al proceso que ejecuta el `fork()` como *proceso **padre*** y al nuevo proceso creado copia del anterior como *proceso **hijo***.

Esta llamada tiene el siguiente formato:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Se trata de una función sin parámetros que es llamada **una vez** (en el proceso que será el padre), y sin embargo, debido a que *duplica* el proceso, retorna **dos veces** (una en cada proceso: una en el padre, y otra en el hijo).

El valor que devuelve (al retornar) es distinto en el padre y en el hijo. Este valor distinto permite al *padre* saber que él es el *padre*, y al *hijo* saber que él es el *hijo*. Concretamente, en el proceso *padre* devuelve un número entero correspondiente al PID del *hijo* (por tanto, un valor siempre mayor que `0`), mientras que en el *hijo* devuelve siempre el valor `(pid_t)0`.

En caso de error, es decir, si no ha podido crear el nuevo proceso hijo, entonces devuelve el valor `(pid_t)-1` al proceso llamante.

Utilizando el valor que devuelve el `fork()` en cada proceso es posible lograr que ambos ejecuten secciones de código distintas del programa.

[3] Existen otras diferencias entre ambos procesos, pero son demasiado específicas como para abordar aquí su estudio pues se aprecian en condiciones que nunca se nos presentarán en nuestro entorno de laboratorio.

A continuación se muestra un ejemplo de ello, en el cual un proceso genera un proceso hijo utilizando la llamada al sistema `fork()` y, a continuación, padre e hijo imprimen cada uno un mensaje distinto en la salida estándar (por defecto la pantalla):

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

main()
{
   pid_t pid_hijo;

   switch(pid_hijo=fork()){           /* crea un hijo */

   case (pid_t)-1:
      perror("error en fork");
      break;

   case (pid_t)0:
      /* proceso hijo */
      printf("soy el hijo. Mi PID es: %ld \n", (long)getpid());
      exit(0); /* terminar el proceso hijo */

   default:
      /* proceso padre */
      printf("soy el padre. PID de mi hijo: %ld \n", (long)pid_hijo);
   }/*switch*/
}/*main*/
```

Si se estudia detenidamente y paso a paso el ejemplo anterior, se observa que inicialmente únicamente existe el proceso padre. Para crear un proceso hijo, primero se asegura de que se vuelcan todos los posibles datos pendientes en los *buffers* de E/S y después ejecuta la llamada `fork()`:

```
...
fflush(NULL);  /* Vuelca todos los buffers. */
switch(pid_hijo=fork()){           /* crea un hijo */
...
```

Si se produce cualquier error y no es posible duplicar el proceso, entonces `fork()` devuelve `(pid_t)-1`, y por tanto no se ha creado el proceso hijo. En este caso el proceso que deseaba ser padre ejecuta la sección del `switch` que se muestra:

```
...
case (pid_t)-1:
   perror("error en fork");
   break;
...
```

Si no se produce ningún error, cuando `fork()` retorna **ya existen 2 procesos**, padre e hijo, copia el uno del otro, ejecutándose simultáneamente en el sistema. Veámos a continuación que hace cada uno de ellos a partir de este momento.

- En el proceso hijo `fork()` devuelve el valor `(pid_t)0`. Este valor se almacena en la variable `pid_hijo` del hijo (copia de la del padre). Por ello, en el proceso hijo se continua ejecutando la siguiente sección del `switch`:

```
...
case (pid_t)0:
   printf("soy el hijo. Mi PID es: %ld \n", (long)getpid());
   exit (0);
...
```

- Es importante observar que la última instrucción del case que contiene el código del hijo debe terminar con la función `exit ()` y no con la instrucción `break`. Esto es debido a que si pusiéramos un `break` el hijo saldría del `case` y continuaría ejecutando la siguiente instrucción tras el `switch`, es decir, tendríamos al proceso hijo ejecutando el mismo código que ejecutaría el padre a partir del `switch`.

- En el proceso padre `fork()` devuelve el PID del proceso hijo creado, cuyo valor se almacena en la variable `pid_hijo` del padre. Debido a que el PID de cualquier proceso, una vez que el sistema está inicializado y funcionando, es siempre un número positivo distinto de cero, en el proceso padre se ejecuta la sección siguiente del `switch`:

```
...
default:
   printf("soy el padre. PID de hijo: %ld \n",(long)pid_hijo);

}/*switch*/
...
```

Utilizando este valor devuelto por `fork()` los procesos padre e hijo pueden ejecutar partes distintas del mismo programa. Es interesante comentar algunas conclusiones y subrayar ciertos aspectos:

- Para que no todos los procesos del sistema ejecuten partes distintas de un mismo programa, sino que ejecuten programas totalmente distintos, se dispone de otras llamadas al sistema que permiten, una vez creado el proceso hijo con `fork()`, desvincularse del programa que se estaba ejecutando y cargar uno nuevo en ese proceso.

- Insistir además en que, al ser todos los datos del padre duplicados en el momento del `fork()`, el hijo dispone de una copia de todos ellos. A partir del `fork()` padre e hijo son procesos distintos por lo que las modificaciones que cualquiera de ellos realice sobre su copia de los datos no afecta ya a los del otro.

  En el ejemplo anterior, la variable `pid_hijo` en el padre tenía algún valor inicial que fue copiado en la variable `pid_hijo` del proceso hijo; tras el `fork()` padre e hijo asignan valores diferentes a estas variables, ya que son variables distintas pertenecientes a procesos distintos.

- Es importante señalar también que junto con el resto de los datos, también se duplica la tabla de descriptores[4] de ficheros del padre. Ello implica que los ficheros que el proceso padre tuviera abiertos en el momento del `fork()` se encuentran también abiertos y accesibles para el proceso hijo tras el `fork()`.

  En el ejemplo anterior, la función `printf()` del padre imprime un mensaje en el fichero de salida que tenga establecido el padre (por defecto la pantalla). La función `printf()` que ejecuta el hijo imprime un mensaje en el mismo fichero de salida que tuviera el padre, ya que lo ha heredado. Por ello, si el fichero de salida es la pantalla, en ésta aparecerán los mensajes siguientes tras la ejecución del programa:

```
soy el hijo. Mi PID es: 1784
soy el padre.PID de hijo: 1784
```

  o bien

```
soy el padre.PID de hijo: 1784
soy el hijo. Mi PID es: 1784
```

  El orden dependerá de cómo el núcleo planifique la ejecución de estos dos procesos.

- Finalmente, obsérvese que el padre conoce el PID del hijo que se ha creado, pero el hijo no conoce el PID del padre. Si desea conocerlo solo tiene que utilizar la llamada al sistema `getppid()`.

4.5.4 Reconocimiento de la terminación de un hijo: llamadas al sistema de la familia wait()

Las llamadas al sistema `wait()`, `waitpid()`, `wait3()`, `wait4()` y `wait6()` se utilizan para obtener información sobre la actividad en los procesos hijos. Esta *actividad* puede ser una de las siguientes :

- Algún proceso hijo ha finalizado debido a una llamada a `exit()`.
- Algún proceso hijo ha finalizado como consecuencia de la recepción de una señal.
- Otros casos que no se tratarán en este manual, como que algún proceso hijo ha sido suspendido, reanudado, está bajo depuración, etc.

Cuando un proceso termina su ejecución, el sistema operativo libera todos los recursos asociados al mismo (memoria, ficheros abiertos…), pero no lo elimina completamente de la tabla de procesos del sistema, sino que lo pasa a un estado denominado “zombi”. Además, puede enviar la señal `SIGCHLD` al proceso padre para notificarle que hay novedades en la actividad de alguno de sus hijos (las señales se explican en el apartado 5.1). El proceso

[4] un descriptor de fichero lo forman los datos gestionados por el sistema que permiten a un proceso manejar un fichero; no confundirlo con el fichero en sí mismo.

zombi queda en ese estado hasta que el padre *reconoce* su terminación y obtiene ciertos datos sobre la finalización del hijo. Una vez que el proceso hijo es reconocido por el padre, sí se elimina definitivamente del sistema. Para evitar situaciones extrañas de horfandad, cuando un proceso padre termina antes que sus hijos, éstos son adoptados por el proceso `init` (PID 1), el cual actuará como proceso padre en adelante.

Para que un proceso padre reconozca la muerte de sus hijos debe usar alguna de las llamadas al sistema de la familia `wait()`. Los prototipos de las llamadas al sistema que se van a comentar abajo son los siguientes:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);

pid_t waitpid(pid_t wpid, int *status, int options);

#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *status, int options, struct rusage *rusage);

pid_t wait4(pid_t wpid, int *status, int options, struct rusage *rusage);
```

Habitualmente estas llamadas son bloqueantes, es decir, dejan bloqueado al proceso que la invoca (en este caso, el padre) hasta que hay algún resultado que comunicar. Más concretamente, al invocar a cualquiera de estas llamadas al sistema puede suceder lo siguiente:

- Si no existe ningún proceso hijo, la llamada al sistema devuelve -1 inmediatamente y errno tiene el valor ECHILD..
- Si existen procesos hijos, pero ninguno está zombi, la llamada al sistema se bloquea hasta que alguno termine. Las llamadas al sistema `wait3()` y `wait4()` permiten especificar la opción `WNOHANG` en el parámetro `options` para que la llamada al sistema no se bloquee y así poder hacer un sondeo.
- Si hay algún proceso hijo en estado zombi (o la llamada al sistema estaba bloqueada hasta que algún hijo pasara al estado zombi), se devuelve su PID y, opcionalmente, se devuelve su código de estado en la variable apuntada por `status` e información sobre su uso de recursos en la variable apuntada por `rusage`. Si alguno de esos punteros es nulo, no se devuelve el correspondiente valor.

El código de estado que se devuelve en `status` tiene una cierta codificación que depende del motivo exacto por el que se ha notificado actividad en el proceso hijo. Para procesar ese código de estado deben usarse algunas macros definidas en `<sys/wait.h>`. Todas tienen como parámetro el código de estado que devuelven las llamadas al sistema de la familia `wait()`, y comprueban si se verifican o no ciertas condiciones. De entre ellas, se muestran las siguientes:

- `WIFEXITED(stat)` Es distinto de cero (cierto) si el proceso hijo terminó normalmente. En este caso, se puede utilizar la macro `WEXITSTATUS(stat)` la cual reporta el código de retorno que devolvió el hijo al llamar a `exit()`.

- `WIFSIGNALED(stat)` Es distinto de cero (cierto) si el proceso hijo terminó debido a la recepción de una señal. En este caso, se puede averiguar qué señal concreta ocasionó la muerte del hijo con la macro `WTERMSIG(stat)`.

El siguiente ejemplo muestra un caso sencillo en el que un proceso crea un único proceso hijo y después espera síncronamente a que el hijo termine su ejecución. La primera versión del ejemplo es muy simple y luego se refina.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

   int estado;
   pid_t pid_hijo;

   fflush(NULL);
   switch(pid_hijo=fork()){

   case (pid_t)-1:
      perror("error en fork");
      exit(1);

   case (pid_t)0:
      /* proceso hijo */
      ... Hace algo ...
      exit(N);    /* fin del hijo. Devuelve código de retorno N. */

   default:
      /* proceso padre */
         /* espera a que termine el hijo */
         /* Ojo: este ejemplo es demasiado simple */
         wait(&estado);
      // Imprime el código de retorno del hijo, que es N
      // (el valor pasado a exit por el hijo).
      printf("El hijo terminó con código de retorno %d\n",
               WEXITSTATUS (estado));

   }/*switch*/

   ... El padre continúa ...
```

El ejemplo anterior no tiene en cuenta que `wait()` podría retornar por otras causas aparte de la terminación normal del hijo. Algunas de estas causas podrían ser:

- El proceso padre podría haber recibido una señal que ha capturado para ejecutar un manejador. En este caso, `wait()` retorna, aunque el proceso hijo no ha sufrido ningún cambio de estado ni ha terminado.

- El proceso hijo podría haber recibido una señal que lo mató inmediatamente, en cuyo caso no existe código de retorno.

- El proceso hijo podría haber abortado con un fallo de segmentación u otro error grave, en cuyo caso tampoco existe un código de retorno. Observe que este caso realmente es un subconjunto del caso anterior, porque el proceso también termina con una señal (SEGV, FPE…)

La primera causa provocará que `wait()` devuelva -1 y la variable `errno` tenga el valor `EINTR`. Para discernir los demás casos habrá que usar las macros que se han indicado más arriba.

Con todo ello, una versión más refinada del ejemplo sería esta:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

   int estado;
   int error;
   pid_t pid_hijo;

   fflush(NULL);
   switch(pid_hijo=fork()){

   case (pid_t)-1:
      perror("error en fork");
      exit(1);

   case (pid_t)0:
      /* proceso hijo */
      ... Hace algo ...
      exit(N);    /* fin del hijo. Devuelve código de retorno N. */

   default:
      /* proceso padre */
      do{
         /* espera a que termine el hijo */
         error=wait(&estado);
         /* si devuelve error y errno toma el valor EINTR, entonces se
         ha interrumpido a causa de una señal y se relanza otra vez */
      }while((error==-1) && (errno==EINTR));
      // Si el hijo terminó con normalidad, imprime el código de
      // retorno, que es el valor N pasado a exit por el hijo.
      if (error != -1) {
         if (WIFEXITED(estado))
            printf("El hijo terminó con código de retorno %d\n",
               WEXITSTATUS (estado));
         else
            printf("El hijo ha terminado con una señal.\n");
      }
      else
         perror("wait");
```

```
}/*switch*/

... El padre continúa ...
```

Se recuerda que en este apartado no se contemplan aplicaciones en las que se necesite controlar la actividad de suspensión, reanudación o depuración de procesos hijos.

4.5.5 Ejecución de programas: llamadas al sistema de la familia exec()

En los apartados anteriores se ha visto cómo crear procesos nuevos, pero hasta ahora todos los procesos nuevos continuaban ejecutando el mismo programa que el proceso padre, aunque desviándose por un camino de ejecución diferente.

Pero hay casos en que los procesos se crean para ejecutar un programa diferente al que ejecutaba el padre y eso se puede conseguir con las funciones POSIX de la familia `exec()`.

Estas llamadas al sistema sirven para sustituir al programa que ejecuta el proceso que invoca a `exec()` por un programa nuevo que empieza desde cero. La única e importante salvedad es que el nuevo programa comenzará con los descriptores de fichero abiertos que haya dejado el programa anterior al ejecutar `exec()`. Esto es fundamentalmente útil cuando el proceso ha redirigido la entrada o salida estándar antes de ejecutar el nuevo programa y en otros casos similares.

La familia de funciones `exec()` tiene diversas variantes que difieren en cómo se especifican los argumentos del programa, dónde se busca el fichero ejecutable y si se pueden definir variables de entorno. En POSIX el nombre de todas estas funciones empieza por *exec* y posteriormente se añaden algunas letras que indican cómo se comportan:

- Si por ejemplo se le añade una “l” (la función se llamaría *execl*) significa que es necesario pasar todos los argumentos individualmente separados por comas y terminar la lista de parámetros en NULL.
- Si se añade una “e” también hay que pasar las variables de entorno.
- Con una “p” se indica que utilice la variable de entorno PATH para encontrar el programa que se pasa como primer argumento. Esta variable de entorno contiene la lista de posibles directorios del sistema donde están los ejecutables instalados y así el programador no tiene que buscar manualmente dónde está el programa[5].
- Finalmente con una “v” se indica que los argumentos se pasan como un array de cadenas. Existen más matices y posibilidades que se pueden consultar en la página de manual `exec(3)`.

[5] Puede ejecutar `echo $PATH` para ver el valor actual de la variable. Habitualmente el directorio `~/bin` está incluido en la lista y es cómodo instalar en él aquellos programas propios que se usen con frecuencia.

Observe que, si la invocación a `exec()` tiene éxito (es decir, es posible ejecutar el programa indicado), el programa desde el que se llamó a `exec()` ya no proseguirá su ejecución.

A continuación se muestra un ejemplo de un programa que recibe datos por medio de la función `obtener_dato()`, no especificada. Para cada dato que recibe, crea un proceso hijo y hace que ejecute el programa `/usr/local/bin/programa` con la opción `-x DATO`, donde `DATO` es el dato que ha recibido. Observe los siguientes detalles:

- El primer parámetro de cualquier función de la familia `exec()` es el nombre del fichero ejecutable. Si se ha usado una función sin "p", ese nombre deberá apuntar directamente al fichero ejecutable deseado.
- El resto de parámetros de cualquier función de la familia `exec()` son los argumentos para el programa que se va a ejecutar, y el primero de ellos (segundo parámetro de `exec()`) ha de ser el nombre del propio programa, generalmente el mismo que el primer parámetro de `exec()`.
- Todos los argumentos de `exec()` son cadenas (o tablas de cadenas en algunas variantes). Si es necesario pasar algún argumento numérico o de otro tipo, hay que convertirlo a cadena, por ejemplo con `sprintf()`.
- Si `exec()` retorna, es porque no pudo ejecutar el programa indicado. Eso denota un fallo (quizá el programa no existe, o no se tienen permisos de ejecución, etc.) e implica que el proceso hijo ha de suicidarse de inmediato. Ese proceso se creó para ejecutar un programa y, si no pudo hacerlo, su existencia ya no tiene sentido. Si el proceso hijo continuara su ejecución, lo que sucedería es que seguiría ejecutando el código tras el `switch` del `fork()` y tendríamos a dos procesos actuando como "padres".

```
#include <stdio.h>
#include <sysexits.h>
#include <stdlib.h>
#include <unistd.h>

int obtener_dato(void);   // No codificada.

int main(int argc, char** argv)
{
   int x;
   char equis[20];
   pid_t pid_hijo;

   do {
      /* Obtiene un dato. */
      x = obtener_dato();
      /* Crea un proceso hijo. */
      pid_hijo = fork();
      switch(pid_hijo) {
         case (pid_t) -1: // Error
            perror("fork");
            exit(EX_OSERR);
         case (pid_t) 0: // Hijo
```

```
            sprintf(equis, "%d", x);
            execl("/usr/local/bin/programa", "programa", "-x", equis, NULL);
            /* Si llega aquí, es que execl falló y se suicida. */
            perror("execl");
            exit(EX_UNAVAILABLE);
         default: // Padre
            /* Hace algo. Por ejemplo, anotar información sobre el */
   /* hijo en una tabla para más adelante detectar los */
            /* procesos que terminan. */
            ...
      }
   } while (1);
}
```

Si en el ejemplo anterior se desea usar execv(), la ejecución del programa se haría así:

```
   char *argumentos[4];

      ...
         case (pid_t) 0: // Hijo
            sprintf(equis, "%d", x);
            argumentos[0] = "programa";
            argumentos[1] = "-x";
            argumentos[2] = equis;
            argumentos[3] = NULL;
            execv("/usr/local/bin/programa", argumentos);
            /* Si llega aquí, es que execv falló y se suicida. */
            perror("execv");
            exit(EX_UNAVAILABLE);
      ...
```

4.5.6 Pausas en la ejecución: función de biblioteca sleep()

La función de biblioteca `sleep()` provoca que el proceso que la ejecuta pase al estado bloqueado durante un tiempo. Su formato es:

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

El parámetro `seconds` indica cuántos segundos se desea suspender la ejecución del proceso[6].

Si, mientras está bloqueado, el proceso recibe alguna señal que tiene definido un manejador, la función retorna prematuramente y devuelve el número de segundos que le quedaban por esperar. Si la espera se completa sin haber recibido señales, la función devuelve cero. Además, la espera podría alargarse más de lo solicitado si el sistema está cargado y el proceso tiene que esperar un tiempo en el estado preparado antes de retornar a la CPU.

[6] Para hacer esperas con mayor resolución pueden usarse las funciones `usleep()` y `nanosleep()`.

4.5.7 Temporizadores: llamada al sistema setitimer()

El sistema operativo proporciona tres temporizadores a cada proceso. Cada temporizador es un reloj de cuenta atrás que el proceso puede activar con un tiempo inicial y que provocará que el proceso reciba una señal cuando el temporizador llegue a cero. La diferencia entre estos tres temporizadores es cómo se decrementa su valor y qué señal generan al expirar. También puede consultarse el tiempo restante en cualquier momento o anular el temporizador antes de que expire.

La gestión de los temporizadores se realiza por medio de estas llamadas al sistema:

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *value);

int setitimer(int which, const struct itimerval *value,
          struct itimerval *ovalue);
```

El parámetro `which` sirve para seleccionar el temporizador con el que se quiere trabajar. Sus posibles valores y las características específicas de cada uno son:

- `ITIMER_REAL`: este temporizador se decrementa en tiempo real, es decir, al mismo ritmo que un reloj de pared. Cuando expira, el proceso recibe la señal `SIGALRM`.

- `ITIMER_VIRTUAL`: este temporizador se decrementa en tiempo virtual del proceso, es decir, cuando la CPU está ejecutando código de usuario del proceso. Cuando expira, el proceso recibe la señal `SIGVTALRM`.

- `ITIMER_PROF`: este temporizador se decrementa en tiempo virtual del proceso y también en tiempo de sistema relacionado con el proceso, es decir, cuando la CPU está ejecutando código de usuario del proceso o bien código del núcleo para atender alguna llamada al sistema invocada por el proceso o algún evento relacionado con el proceso. Cuando expira, el proceso recibe la señal `SIGPROF`.

El parámetro `value` se utiliza para indicar el valor que se quiere poner en el temporizador (en `setitimer`) o para recibir el valor actual del temporizador (en `getitimer`). En la página de manual de estas llamadas al sistema se describe su estructura, el valor máximo, la resolución mínima, etc. El valor puede ser para una cuenta atrás única, o para una cuenta atrás que se reinicia automáticamente cada vez que expira.

El parámetro `ovalue` se utiliza, opcionalmente, para averiguar el valor anterior del temporizador. Esto es útil cuando el temporizador se quiere usar para varias cuentas atrás simultáneamente, ya que permite reajustar el temporizador cuando expira con el tiempo restante para la siguiente cuenta que estaba programada.

El siguiente ejemplo muestra una forma sencilla de usar el temporizador de tiempo virtual del proceso para medir la velocidad de cómputo de la CPU para hacer un cierto cálculo (por ejemplo, cifrar datos). Para ello cuenta cuántas veces es capaz de repetir el cálculo en un

intervalo de 10 segundos. En este ejemplo se ha usado el temporizador de tiempo virtual del proceso para que el cálculo de la velocidad no se vea afectado por el consumo de CPU de otros procesos. También se supone que el contador `num_iteraciones` no llega a dar la vuelta durante el intervalo de prueba.

```
#include <sys/time.h>
#include <stdio.h>
#include <signal.h>

#define SEG_PRUEBA 10

// Se pondrá a 1 cuando expire el temporizador.
int expirado = 0;

void manejador_temporizador(int s)
{
   expirado = 1;
}

int main(int argc, char** argv)
{
   struct itimerval cuenta;
   int num_iteraciones;

   // Captura la señal que se recibirá cuando expire el temporizador.
   signal(SIGVTALRM, manejador_temporizador);

   // it_value indica el valor inicial de la cuenta atrás.
   cuenta.it_value.tv_sec = SEG_PRUEBA;
   cuenta.it_value.tv_usec = 0;
   // it_interval=0 para cuenta atrás única.
   cuenta.it_interval.tv_sec = 0;
   cuenta.it_interval.tv_usec = 0;

   // Lanza el temporizador.
   setitimer(ITIMER_VIRTUAL, &cuenta, NULL);

   // Hace todas las iteraciones que pueda hasta que expire la alarma.
   num_iteraciones = 0;
   while (! expirado) {
      //... cifrar 1 KB
      num_iteraciones++;
   }
   printf("Velocidad: %.3f KB/seg\n", 1.0*num_iteraciones/SEG_PRUEBA);
}
```

A veces se quiere que un proceso espere un tiempo máximo en una llamada al sistema bloqueante. Algunas de estas llamadas al sistema (por ejemplo, `select()` o `poll()`) tienen un parámetro que permite especificar un tiempo máximo de espera. Otras no tienen ese parámetro, pero se desbloquean si el proceso recibe alguna señal (por ejemplo, `wait()`). Para este segundo caso, el temporizador de tiempo real puede usarse para imponer un tiempo máximo de bloqueo. La idea es activar el temporizador de tiempo real antes de invocar a la llamada al sistema bloqueante. Cuando la llamada al sistema retorna, se puede examinar su valor de retorno y la variable `errno` para saber si retornó porque expiró

el temporizador (el valor de retorno será -1 y `errno` valdrá `EINTR`), o bien porque se produjo el evento por el que estaba esperando el proceso (en este caso habrá que cancelar el temporizador para que no interrumpa más adelante). Si el proceso puede recibir más señales, aparte de las del temporizador, entonces será necesario usar alguna variable auxiliar para determinar qué señal interrumpió la llamada al sistema bloqueante.

5 Comunicación entre procesos

En este apartado se van a mostrar algunos de los posibles mecanismos que permiten la comunicación entre procesos. Algunos de los mecanismos únicamente son aplicables a procesos que mantienen una relación entre ellos (padre-hijo, o entre hijos que proceden de un mismo padre), mientras que otros además son también aplicables a procesos independientes, los cuales pueden incluso estar situados en máquinas distintas.

De entre todos los posibles en un entorno POSIX, únicamente estudiaremos aquellos que vamos a utilizar en las prácticas de este laboratorio.

5.1 Señales

Una forma básica de comunicación entre procesos, y entre el núcleo y los procesos, es el mecanismo que proporcionan las señales. Una señal es una notificación a un proceso de que ha ocurrido un evento. Normalmente ocurren de forma asíncrona, por lo que el proceso receptor no conoce el momento en el que recibirá la señal. Aunque el mecanismo de las señales es completamente software, el modelo está inspirado en el mecanismo de las interrupciones hardware.

Existen varias señales diferentes, cada una de las cuales tiene un nombre asociado, definido en `<signal.h>`. En la siguiente tabla se muestran algunas de las señales más comunes, junto a su descripción y la acción que realiza por defecto el sistema con el proceso que recibe la señal:

Tipo de señal	*Número*	*Descripción de la señal*	*Por defecto*
`SIGHUP`	1	Desconexión del terminal	Finalizar
`SIGINT`	2	Carácter de interrupción (`Control-C`)	Finalizar
`SIGKILL`	9	Eliminar el proceso	Finalizar
`SIGPIPE`	13	Tubería rota	Finalizar
`SIGTERM`	15	Señal de finalización *software*	Finalizar
`SIGUSR1`	16	Señal definible por el usuario 1	Finalizar
`SIGUSR2`	17	Señal definible por el usuario 2	Finalizar
`SIGCHLD`	18	Cambio en el estado de un proceso hijo	Ignorar
`SIGSTOP`	23	Señal de parada	Suspender el proceso

SIGTSTP	24	Señal de parada generada desde el teclado (Control-Z)	Suspender el proceso
SIGCONT	25	Continuar un proceso suspendido	Continuar el proceso

5.1.1 Envío de señales a un proceso

Las señales pueden ser generadas de varias formas distintas. Entre ellas:

- Un proceso puede enviar una señal a otro haciendo uso de la llamada al sistema `kill()`, que se explica más adelante.

  Sin embargo, un proceso no puede enviar señales a cualquier otro. Únicamente a aquellos procesos que tengan el mismo UID efectivo (es decir, que *pertenezcan* al mismo usuario). Como excepción, los procesos del superusuario pueden enviar señales a cualquier proceso del sistema.

- Generadas con el mandato `kill` en el *shell* (intérprete de comandos). Este mandato básicamente se encarga de generar una llamada al sistema `kill()` descrita en el punto anterior. Su formato puede ser dependiente del *shell* utilizado y se puede consultar en su documentación.

- Ciertas combinaciones de caracteres en la entrada estándar (habitualmente el teclado) provocan el envío de señales al proceso que se está ejecutando en primer plano: `Control-C` genera una señal `SIGINT` para terminar con dicho proceso, o bien `Control-Z` para generar una señal `SIGTSTP` para suspender temporalmente la ejecución de dicho proceso.

  Estas combinaciones de teclas son detectadas por el núcleo del sistema, el cual se encarga de enviar la señal adecuada al proceso en cuestión.

- Ciertos eventos provocan que el núcleo genere una señal automáticamente para algún proceso. Por ejemplo, cuando un proceso termina, su padre recibe la señal SIGCHLD. Si un proceso intenta ejecutar alguna instrucción incorrecta, el propio proceso recibe una señal dependiente del caso concreto. Por ejemplo, SIGFPE para ciertas operaciones matemáticas incorrectas, SIGBUS o SIGSEGV para un acceso a una dirección de memoria incorrecta (como un puntero inválido o un puntero nulo), etc.

5.1.1.1 Llamada al sistema kill()

Esta llamada, que permite enviar una señal a un proceso, tiene el siguiente formato:

```
#include <sys/types.h>
#include <signal.h>

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

donde el parámetro `pid` indica el PID del proceso al que enviar la señal, y `sig` especifica el tipo de señal a enviar (uno de los nombres de la tabla anterior, predefinidos en `<signal.h>`). En caso de error retorna el valor entero `-1`.

Si para el parámetro `pid` se utiliza el valor especial cero, entonces la señal se envía a todos los procesos del grupo de procesos al que pertenece el proceso que envía la señal. En general, esto significa que se envía a todos los descendientes del proceso, y también a sí mismo.

5.1.2 Recepción de señales en un proceso

Hasta este momento hemos estudiado los tipos de señales más relevantes, y algunos posibles eventos que pueden generar una señal. A continuación vamos a mostrar las señales desde el punto de vista del proceso receptor. De este modo, el proceso receptor puede actuar de varias formas al recibir una señal:

- No especificar nada al respecto de una señal determinada. En este caso, se realizan las acciones que cada sistema tiene previstas por defecto para cada señal (esta acción por defecto puede ser en unos casos simplemente ignorarla, y en otros realizar alguna acción, tal como finalizar la ejecución del proceso).

  En la tabla anterior se especifica la acción por defecto establecida para cada una de las señales mostradas.

- El proceso puede proporcionar una función para que ésta sea ejecutada asíncronamente cuando se reciba la señal. A esta función se la denomina comúnmente *manejador* (*handler*), y a este mecanismo se lo denomina *capturar la señal*. Para establecer esta captura se utiliza la función `signal()`.

- Un proceso también puede ignorar la señal. Esto significa que, cuando se reciba ese tipo de señal, no se realiza ninguna acción. Para establecer que se ignore una determinada señal, se puede hacer también uso de la función `signal()`[7].

5.1.2.1 Función de biblioteca signal()

Esta función de biblioteca permite elegir la forma en que se trata una señal. Esto es, permite capturarla, ignorarla, o bien reestablecer la acción por defecto que el sistema tiene prevista para ella.

Para cada tipo de señal existente es posible establecer un tipo de tratamiento diferente. Las señales `SIGKILL` y `SIGSTOP` son excepciones, y no puede ser modificado el tratamiento por defecto que el sistema tiene establecido para ellas.

[7] Existen otras funciones de biblioteca auxiliares que permiten hacer tareas específicas sobre señales, simplificando el interfaz que ofrece signal().

El formato de esta función es el siguiente:

```
#include <signal.h>

void (*signal(int sig, void (*manejador)(int))) (int);
```

Es decir, `signal()` es una función que devuelve un puntero a otra función. Esta declaración que parece a primera vista complicada, en realidad no lo es tanto:

- El parámetro `sig` indica a qué señal nos estamos refiriendo. No puede tomar nunca los valores `SIGKILL` ni `SIGSTOP` (lo cual implica que ningún proceso puede capturar o ignorar estas señales).
- El parámetro `manejador` sirve para indicar el tipo de tratamiento deseado para la señal.
  - Si deseamos capturar la señal, se pasa a través de este parámetro la función manejador que deseamos que se ejecute cuando esta señal se reciba. Esta función debe tener el siguiente prototipo:

    `void manejador(int sig);`

    Así, cuando se recibe la señal se interrumpe el flujo normal del programa para pasar a ejecutar esta función. El sistema proporciona además un parámetro entero `sig` para indicar al manejador el código de la señal que se ha recibido. Tras ejecutar este *manejador*, se reanuda el flujo normal del programa en la instrucción siguiente a aquella tras cuya finalización fue interrumpido.
  - Si se desea ignorar la señal se debe pasar a través de este parámetro `manejador` el valor especial predefinido `SIG_IGN`.
  - Si se desea reestablecer el modo por defecto que tenga establecido el sistema para esa señal se debe pasar el valor especial predefinido `SIG_DFL`.
- La función `signal()` devuelve la dirección de la función que era ejecutada anteriormente cada vez que se recibía la señal. Este valor sería útil si en un programa se quisiera cambiar un manejador de señal y más adelante restaurar el que había anteriormente.

Por ejemplo, si decidimos capturar la señal `SIGINT` (recuerde que esta señal se genera al pulsar Control-C) y deseamos que cuando esta señal se reciba se ejecute asíncronamente la función `manejador_SIGINT`, la cual imprime un mensaje en la salida estándar del proceso, entonces:

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

```
void manejador_SIGINT(int sig)
{
   printf("He recibido una señal SIGINT\n");
}

main()
{
   /* Captura de la señal SIGINT */
   signal(SIGINT, manejador_SIGINT);

   /* Bucle para hacer pruebas. */
   while (1) {
      printf(".");
      fflush(stdout);
      sleep(1);
   }
}
```

Si, por ejemplo, deseamos ignorar la señal `SIGINT` para que el proceso no pueda ser terminado pulsando Control-C, ejecutamos lo siguiente:

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

main()
{
...
   signal(SIGINT, SIG_IGN);
...
}
```

Si, por ejemplo, a continuación decidimos volver a establecer la forma de gestionar esta señal que el sistema establece por defecto (la cual es la finalización del proceso), entonces:

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

main()
{
...
   signal(SIGINT, SIG_DFL);
...
}
```

Por último, algunos detalles a tener en cuenta en relación con la gestión de señales son:

- Al instalar un manejador para una determinada señal, el manejador permanece instalado indefinidamente. Es decir, tras atender la primera señal que se reciba con el manejador, la siguiente y sucesivas vuelve a tratarse con el mismo manejador.

- Las señales se generan y se atienden de un modo muy similar al de las interrupciones hardware. Por cada proceso, el sistema operativo mantiene un array de bits (un bit por señal), de forma que, cuando un proceso *A* envía una señal *s* a otro proceso *B*, se activa el bit correspondiente a la señal *s* en el array de bits del proceso *B* (podría ser que ese bit ya estuviera activado, con lo que el envío de esta señal no tendría ningún efecto). El sistema operativo decidirá en qué momento debe ejecutarse el manejador correspondiente a esa señal. Justo antes de ejecutar el manejador de una señal, el sistema operativo desactiva el bit que indica que se ha recibido esa señal. El manejador de una señal tiene que tener en cuenta que cuando es invocado es porque el proceso ha recibido la señal una o varias veces, ya que desde que se recibe la señal la primera vez, hasta que se ejecuta el manejador, ha podido pasar un tiempo en la cola de preparados esperando ser elegido para tomar la CPU, y durante ese tiempo, otros procesos pueden haberle enviado la misma señal.

- Se puede bloquear y desbloquear explícitamente la recepción de señales mediante la llamada al sistema `sigprocmask()`. Esto es útil cuando un programa entra en una región crítica dentro de la cual no es seguro procesar el manejador de alguna señal. Si se envió la señal una o varias veces al proceso mientras este la tenía bloqueada, entonces se ejecutará el manejador una única vez al desbloquearla.

5.1.2.2 Ejemplo de bloqueo de señales

Casi cualquier aplicación que maneje señales de una forma no trivial tendrá algunas secciones críticas en las que el programa principal no debe ser interrumpido por un manejador de señal, so pena de sufrir errores de concurrencia. Por ejemplo, si una aplicación multiproceso tiene una tabla de procesos hijos y el programa principal la modifica para añadir procesos cuando los crea, mientras que el manejador de SIGCHLD la modifica para eliminarlos cuando terminan, la parte del programa principal que añade procesos a la tabla es una sección crítica y durante ella ha de evitarse que se active el manejador de `SIGCHLD`[8].

La forma de garantizar la atomicidad de las secciones críticas frente a señales POSIX es análoga a la que se usa en las interrupciones hardware: al inicio de la sección crítica se bloquea la recepción de ciertas señales y al final de la sección crítica se vuelve a permitir dicha recepción. Si llegaron señales durante la sección crítica, se habrán quedado encoladas y se procesarán una vez terminada la sección crítica.

Para bloquear una determinada señal es necesario manejar máscaras. Las máscaras son arrays de bits que corresponden cada uno a una señal. Estos bits indican si en caso de recibir

[8] La parte del manejador que elimina los procesos de la tabla también sería una sección crítica, pero su atomicidad está garantizada porque el programa principal no puede interrumpir al manejador.

una determinada señal, se debe atender o no. Por diseño, algunas señales, como por ejemplo SIGKILL, no se pueden enmascarar. A continuación se muestra un ejemplo de funciones para enmascarar y desenmascarar una señal concreta que se recibe como parámetro. Las funciones de ejemplo devuelven además la máscara antigua, por si el llamante quiere hacer algo con ella, como por ejemplo restaurarla posteriormente.

```
#include <signal.h>

/* bloquea la señal indicada en el parámetro sig y devuelve la antigua
   máscara de señales bloqueadas antes de realizar esta operación.
*/
sigset_t enmascarar ( int sig )
{  sigset_t mascara;
   sigset_t antigua;

   sigemptyset ( &mascara ); /* pone todos los bits a 0) */
   sigaddset ( &mascara, sig ); /* pone a 1 el bit correspondiente */
   sigprocmask ( SIG_BLOCK, &mascara, &antigua );
   return antigua;
}

/* bloquea la señal indicada en el parámetro sig y devuelve la antigua
   máscara de señales bloqueadas antes de realizar esta operación.
*/
sigset_t desenmascarar ( int sig )
{  sigset_t mascara;
   sigset_t antigua;

   sigemptyset ( &mascara ); /* pone todos los bits a 0) */
   sigaddset ( &mascara, sig ); /* pone a 1 el bit correspondiente */
   sigprocmask ( SIG_UNBLOCK, &mascara, &antigua );
   return antigua;
}
```

Para restaurar una máscara anterior, basta con cambiar la máscara actual por la que se respaldó anteriormente. Para ello se usa la función `sigsetmask()`.

5.1.2.3 Ejemplo de esperar hasta que se reciba una señal

En ocasiones, un proceso necesita esperar hasta que se active una señal. La llamada al sistema `sigsuspend()` sirve para este fin, pero hay que usarla correctamente para evitar problemas de concurrencia, particularmente carreras. Esta llamada al sistema desbloquea las señales que se le indican y espera hasta que llegue una señal cualquiera que esté desbloqueada, todo ello de forma atómica.

Supongamos que un programa quiere esperar hasta que llegue la señal `SIGUSR1`. Inicialmente la variable global `aviso` vale 0, y el programa sabe que ha llegado la señal porque el manejador de `SIGUSR1` la pone a 1.

Una versión simple, aunque incorrecta, del fragmento de código que espera a que llegue la señal podría ser esta:

```
#include <signal.h>

...
// Este ejemplo es incorrecto: puede sufrir carreras
if (! aviso)
   sigsuspend(NULL);
...
```

La idea es que solo hay que esperar si la señal no había llegado antes del `if`. Ahora bien, si el proceso tiene la mala suerte de que la señal `SIGUSR1` llega justo después de ver que `aviso` está a cero, pero antes de invocar a `sigsuspend()`, entonces se quedará bloqueado indefinidamente[9]. Además, este código tiene el defecto de que `sigsuspend()` sale por la recepción de cualquier señal, no solo `SIGUSR1`, aunque esto se corrige fácilmente cambiando el `if` por un `while`.

Para implementar correctamente la espera hay que establecer una sección crítica con respecto a `SIGUSR1` en la comprobación de la variable `aviso`, pero dicha sección crítica no debe abarcar a `sigsuspend()`, porque entonces nunca saldría con la recepción de `SIGUSR1`. Esto se consigue así:

```
#include <signal.h>

...
// Bloquea SIGUSR1
sigset_t mascara, mascara_anterior;
sigemptyset ( &mascara );
sigaddset ( &mascara, SIGUSR1 );
sigprocmask ( SIG_BLOCK, &mascara, &mascara_anterior );
// Comprueba la variable, sabiendo que el manejador no puede cambiarla
while (! aviso)
   sigsuspend(&mascara_anterior); // Espera, desbloqueando SIGUSR1
// Finalmente, desbloquea SIGUSR1
sigprocmask ( SIG_UNBLOCK, &mascara, NULL );
...
```

5.1.3 Ejemplos y conceptos finales

Para afianzar estos conceptos, a continuación se muestra un programa en el que el hijo ejecuta un cierto bucle hasta que el padre le ordena terminar enviándole la señal SIGTERM. Obsérvese que, por defecto, la señal SIGTERM causa la finalización inmediata del proceso que la recibe, por lo que parece innecesario capturarla para terminar. Ahora bien, capturarla es útil cuando se quiere que la terminación del hijo sea ordenada y no abrupta.

[9] Cuando una aplicación puede fallar dependiendo de si ciertos eventos concurrentes se dan en un orden o en otro, se dice que la aplicación tiene un problema de carreras.

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int terminar_hijo = 0;


void manejador_SIGTERM(int sig)
{
  terminar_hijo = 1;
}


main()
{
   pid_t pid_hijo;


   /* Captura de la señal SIGTERM */
   signal(SIGTERM, manejador_SIGTERM);

   /* Creamos un hijo */
   fflush(NULL);
   switch(pid_hijo=fork()){

   case (pid_t)-1:
      perror("error en fork\n");
      exit(1);
      break;

   case (pid_t)0:
      /* proceso hijo */
      while (! terminar_hijo) {
        ... Hace algo ...
      }
      ... Termina ordenadamente ...
      exit(0);
      break;

   default:
      /* proceso padre */
      ... Hace algo ...
      if(kill(pid_hijo, SIGTERM)==-1){  /* envío de señal */
         perror("error en kill");
         exit(1);
      }
      exit(0); /* fin del padre */
   } /* switch */
} /* main */
```

A continuación se muestra otro ejemplo en el que un proceso crea varios hijos y, mientras sigue haciendo otras cosas, va contando cuántos hijos vivos le quedan. Cuando todos los

hijos han terminado, el padre también termina. Para estar al tanto de la terminación de los hijos, el proceso padre captura la señal SIGCHLD.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int num_hijos = 0;


void manejador_SIGCHLD(int sig)
{
  pid_t pid_hijo;
  int e;

  do {
     pid_hijo = wait3(&e, WNOHANG, NULL);
     if (pid_hijo > (pid_t) 0 &&(WIFEXITED(e) ||  WIFSIGNALED(e)))
       num_hijos--;
  } while (pid_hijo > (pid_t) 0);
}


main()
{
   pid_t pid_hijo;
   int i;

   /* Captura de la señal SIGCHLD */
   signal(SIGCHLD, manejador_SIGCHLD);

   /* Creamos varios hijos */
   fflush(NULL);
   for (i = 0; i < N; i++) {
      num_hijos++;
      switch(pid_hijo=fork()){
      case (pid_t)-1:
         perror("error en fork\n");
         exit(1);
         break;
      case (pid_t)0:
         /* proceso hijo */
         ... Hace algo ...
         exit(0);
         break;
      }
   }

  /* El proceso padre continúa por aquí tras crear todos los hijos. */
   while (num_hijos > 0) {
      ... Hace algo ...
   }
}
```

Obsérvese que en el manejador de la señal SIGCHLD se ha implementado un bucle para reconocer la muerte de todos los posibles procesos zombis que haya en ese momento. Esto es necesario por si varios procesos hijos han terminado mientras el proceso padre estaba preparado para acceder a la CPU.

Finalmente, destacar además los siguientes conceptos relativos a las señales:

- Obsérvese que, para que el manejador de la señal pueda influir en la ejecución del programa, se usan variables globales. Es uno de los pocos casos en que el uso de variables globales está justificado, pero no debe abusarse de ellas.

- Aunque un manejador de señal puede ejecutar cualquier acción que se desee, con carácter general se recomienda que no sea una función demasiado compleja, sino que más bien se limite a actualizar alguna estructura de datos global y deje que el programa principal lleve a cabo otras acciones más complejas al detectar los cambios realizados por el manejador de la señal. En cualquier caso, nunca deberían ser funciones que puedan dejar bloqueado al proceso y siempre han de contemplarse posibles conflictos de concurrencia con el programa principal. La página de manual `sigaction(2)` contiene una lista de funciones cuyo uso es seguro, desde el punto de vista de la concurrencia, dentro de un manejador de señal.

- Obsérvese que el hecho de que el sistema operativo pase como parámetro a la función *manejador* (en los ejemplos anteriores: `manejador_SIGINT`, `manejador_SIGTERM`) qué señal que se ha producido (a través del parámetro `sig`) hace posible, si se desea, establecer un único manejador para todas las señales. Así, el manejador puede discriminar a través de este parámetro qué señal se ha recibido y actuar en consecuencia, en lugar de establecer varios manejadores diferentes, uno para cada señal capturada.

  Por defecto, si un proceso recibe una señal en un momento en el que está ejecutando una llamada al sistema, ésta se completará antes de ejecutar el manejador para la señal. Sin embargo, cuando se recibe alguna señal mientras se ejecuta una llamada al sistema de las consideradas *lentas* (como `read()`, `select()`, `sleep()`, `wait()`,..., es decir, llamadas que se pueden encontrar bloqueadas algún tiempo en espera de algún evento), entonces se da por finalizada la llamada al sistema que se estaba ejecutando, y se pasa a ejecutar el *manejador* establecido para la señal. Si el manejador no termina la ejecución del proceso, al terminar de ejecutarse, la llamada al sistema (o función de biblioteca) que fue interrumpida por la señal retorna con un código de error (y actualiza la variable `errno` con el valor `EINTR` para indicar que la finalización de la llamada ha sido debido a una señal). En este caso, el programador puede, si lo desea, relanzar (volver a ejecutar) la llamada al sistema *lenta* para realizar la tarea que fue interrumpida.

- Finalmente, obsérvese cómo las señales permiten que un proceso reciba la notificación de un evento, pero el proceso no puede extraer de la señal ninguna información adicional (salvo aquello que pueda deducir en función del tipo de señal que haya recibido).

Si un proceso desea, por tanto, enviar datos o información de cualquier tipo a otro proceso, es necesario utilizar algún otro mecanismo de comunicación entre procesos distinto de las señales, o bien que las complemente.

5.2 Tuberías (*pipes*)

Una tubería (*pipe*) proporciona un canal de comunicación unidireccional[10], que permite enviar datos de un extremo del canal al otro y viceversa, simultáneamente.

Nos va a permitir que dos o más procesos emparentados (padres con hijos o hijos con hijos) se comuniquen enviandose cualquier tipo de información entre ellos.

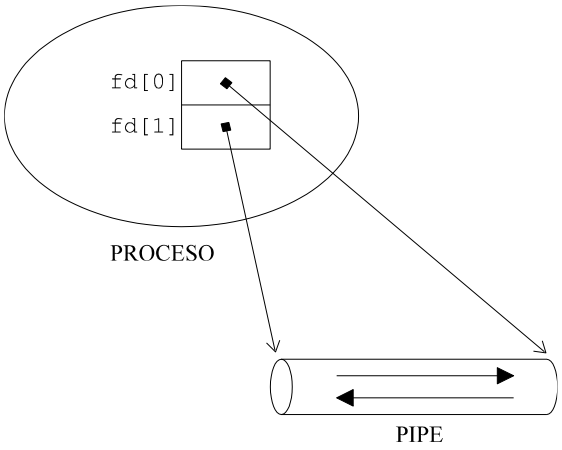
5.2.1 Llamada al sistema pipe()

La llamada al sistema para crear una tubería tiene el siguiente formato:

```
#include <unistd.h>

int pipe(int fd[2]);
```

A través del parámetro `fd` la función devuelve dos descriptores de fichero abiertos para lectura/escritura: `fd[0]` y `fd[1]` que permiten leer y escribir en cada extremo del canal. La función devuelve un valor `-1` en caso de que ocurra algún error y no sea posible crear la tubería.



A continuación se muestra un ejemplo de utilización de tubos. En el ejemplo, introducimos una estructura de datos por un extremo del canal, y por el otro extremo la leemos.

[10] El estándar POSIX define las tuberías como canales unidireccionales, aunque algunos sistemas operativos compatibles con POSIX, como FreeBSD, permiten comunicación bidireccional. Si se desea utilizar tuberías unidireccionales para comunicación bidireccional, es necesario crear dos tuberías distintas, para utilizar uno de ellas en uno de los sentidos de la comunicación (padre→hijo), y la otra en el sentido opuesto (hijo→padre).

Obviamente, lo que hace este ejemplo es poco útil y simplemente sirve para ver cómo se pueden escribir y leer datos en una tubería.

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

typedef struct {
   int campo1;
   long campo2;
   char campo3[10];
} Ejemplo;

main()
{
   int tubo[2];
   ssize_t leidos;
   Ejemplo cosa1, cosa2;

   /* Creamos la tubería */
   if(pipe(tubo)==-1){
      perror("error en pipe");
      exit(1);
   }

   ... Rellenamos cosa1 con datos cualesquiera

   /* Escribimos cosa1 en un extremo de la tubería */
   if(write(tubo[1], &cosa1, sizeof(cosa1))!=sizeof(cosa1)){
      perror("error en write");
      exit(1);
   }

   /* Leemos del otro extremo de la tubería */
   if((leidos=read(tubo[0], &cosa2, sizeof(cosa2)))==(ssize_t)-1){
      perror("error en read");
      exit(1);
   }

   ... En cosa2 habrá lo mismo que había en cosa1


}/*main*/
```

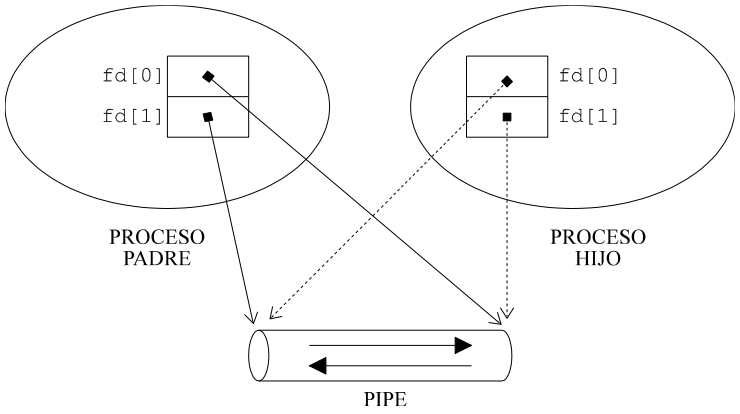
Los datos se escriben en un extremo de la tubería y son entregados en el otro extremo en el mismo orden en que fueron escritos. Cuando se solicita la lectura de un cierto número de *bytes* de la tubería se leen tantos caracteres como se encuentren disponibles, hasta el número solicitado. Estos *bytes* pueden haber sido escritos a partir de una única llamada a `write()` o de varias llamadas sucesivas.

En caso de que no exista ningún carácter en el canal y se solicite una lectura, ésta se quedará bloqueada esperando a que exista algún carácter que leer (si se ha cerrado el extremo

opuesto de escritura de la tubería, la llamada `read()` finaliza indicando que el número de caracteres leídos ha sido `0`).

5.2.2 Ejemplo de comunicación padre-hijo utilizando tubos

Como se puede observar, dentro de un mismo proceso este mecanismo no tiene una utilidad especial. Su utilidad aparece en el momento en que un proceso crea una tubería, y a continuación ejecuta un `fork()`. Entonces, el hijo hereda todos los descriptores que el padre tuviera abiertos, y entre ellos los de la tubería, por lo que tanto padre como hijo tienen acceso a ambos extremos de la tubería.



Al tener acceso padre e hijo a ambos extremos de la tubería pueden utilizarse dos combinaciones diferentes: que el padre lea/escriba de `fd[0]` y el hijo lea/escriba de `fd[1]`, o viceversa. En ambos casos, el extremo de la tubería que no se utilice en el padre es conveniente cerrarlo haciendo uso de `close()`, al igual que el extremo contrario que no se utiliza en el hijo.

A continuación se muestra un ejemplo de comunicación bidireccional entre padre e hijo. Ambos se envían mutuamente un mensaje a través de la tubería y cada uno de ellos imprime lo recibido a través de la tubería en la salida estándar.

Como los mensajes intercambiados en este ejemplo son cadenas de texto, resulta más conveniente escribirlas y leerlas de la tubería con las funciones de la biblioteca estándar de C. Por ello se utiliza fdopen() para obtener un manejador de fichero de la biblioteca estándar de C a partir de los descriptores de fichero que crea el núcleo para los extremos de la tubería.

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sysexits.h>

main()
```

```
{
   int tubo[2];
   char buf[100];    /*  Buffer de 100 caracteres */
   pid_t pid_hijo;
   FILE* canal;

   /* Creamos la tubería */
   if(pipe(tubo)==-1){
      perror("error en pipe");
      exit(EX_OSERR);
   }

   /* Creamos un hijo */
   fflush(NULL);
   switch(pid_hijo=fork()){

   case (pid_t)-1:
      /* Error */
      perror("error en fork");
      exit(EX_OSERR);
      break;

   case (pid_t)0:
      /* proceso hijo */
      /* hijo elige tubo[0] para lectura/escritura. Cierra tubo[1] */
      close(tubo[1]);
      canal = fdopen(tubo[0], "r+");

      /* Envía un mensaje al padre */
      fprintf(canal, "Hola papi. Soy el hijo %d\n", getpid());

      /* Espera leer un mensaje del padre */
      fgets(buf, sizeof(buf), canal);

      /* Muestra el mensaje leído de la tubería y termina */
      printf("%s", buf);
      exit(EX_OK);   /* fin normal del hijo */
      break;



   default:
      /* proceso padre */
      /* padre elige tubo[1] para lect/escr ya que hijo usa tubo[0] */
      close(tubo[0]);
      canal = fdopen(tubo[1], "r+");

      /* Envía un mensaje al hijo */
      fprintf(canal, "Hola. Soy tu padre.\n");

      /* Espera recibir un mensaje del hijo */
      fgets(buf, sizeof(buf), canal);

      /* Muestra el mensaje leído de la tubería y finaliza */
      printf("%s", buf);
      exit(EX_OK); /* fin normal del padre */

   }/*switch*/

}/*main*/
```

Utilizando este mecanismo, es posible por tanto que el proceso padre se comunique con el proceso hijo (o también que dos procesos hijos de un mismo proceso padre se comuniquen) a través de la tubería.

5.2.3 Ejemplo de comunicación de proceso padre con múltiples hijos utilizando tubos

A continuación se presenta otro ejemplo más de comunicación entre un proceso padre y sus hijos mediante tuberías. El padre crea varios hijos y usa una tubería diferente para comunicarse con cada hijo. Después de crear cada hijo, el padre envía un número al hijo a través de la tubería. Por su parte, cada hijo lee el número de la tubería y responde al padre con la raíz cuadrada de dicho número. Después de crear todos los hijos, el padre entra en un bucle en el que espera a que vayan terminando los hijos. Para cada hijo que termina, el padre lee su respuesta y la escribe por pantalla.

Hay algunos elementos de interés que se pueden comentar en este ejemplo:

- El padre necesita guardar información sobre sus hijos. En concreto, necesita guardar el descriptor de la tubería que le comunica con cada hijo. La estructura de datos `Hijo` agrupa toda la información necesaria sobre un hijo, la cual incluye el PID del hijo para poder localizarlo. La información sobre todos los hijos se guarda en una tabla llamada `hijos`. Como el tamaño de la tabla se decide en tiempo de ejecución, se ha usado `calloc()` para obtener memoria para la tabla dinámicamente.

- El programa usa números pseudoaleatorios para que las pruebas sean variadas. Estos números se generan con la función `random()` y para que la secuencia sea diferente en cada ejecución se inicializa el generador de números pseudoaleatorios con `srandom()`. Nota: observe qué sucede si elimina la llamada a `srandom()`.

- En las terminaciones del programa con `exit()` se usan los códigos de estado recomendados por la página de manual `sysexits(3)`.

- El tratamiento de errores del programa es muy básico. En primer lugar, no se comprueba si ha habido error en todos los casos, sino solo en los que se consideran más delicados. En segundo lugar, el comportamiento en caso de error es escribir un mensaje por pantalla y suicidarse sin más.

- El bucle que usa el padre para esperar a que terminen los hijos también es un tanto básico. En primer lugar, mientras el padre espera a que termine un hijo (en wait()), no puede hacer nada más. Para este programa de ejemplo esto es suficiente, pero si el padre necesitara poder hacer otras cosas mientras tanto debería usar un mecanismo basado en capturar SIGCHLD, tal como se describe en el apartado 4.5.4. En segundo lugar, cuando wait() retorna el padre supone que es porque ha terminado un hijo, pero como se comentó en el apartado 4.5.4, esta llamada al sistema podría haber retornado porque el hijo se ha suspendido o porque el padre ha recibido una señal

mientras esperaba. En este ejemplo se ha optado por no considerar esos casos, pero en una aplicación más elaborada podría ser necesario tenerlos en cuenta.

- El código que ejecuta el hijo se ha llevado a la función `proceso_hijo()` con el fin de no hacer farragoso el programa principal y de delimitar más claramente la funcionalidad del hijo.

A continuación se muestra el código del ejemplo:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sysexits.h>
#include <time.h>
#include <unistd.h>

typedef struct {
  pid_t pid;
  int tubo;
} Hijo;

void proceso_hijo(int tubo);

main()
{
  int tubo[2], i, x, num_hijos, num_terminados;
  Hijo* hijos;
  pid_t pid_hijo;
  double res;

  /* Inicializa números pseudoaleatorios. */
  srandom(time(NULL));

  /* Se inventa un número de hijos 4-10 */
  num_hijos = 4 + random() % 7;

  /* Crea dinámicamente la tabla de hijos. */
  if ((hijos = calloc(num_hijos, sizeof(Hijo))) == NULL) {
    perror("error al pedir memoria");
    exit(EX_OSERR);
  }

  /* Creamos los hijos */
  fflush(NULL);
  for (i = 0; i < num_hijos; i++) {
    /* Creamos la tubería que se usará para este hijo. */
    if(pipe(tubo)==-1){
      perror("error en pipe");
      exit(EX_OSERR);
    }
    /* Creamos el hijo, rellenando su estructura Hijo. */
    hijos[i].pid = fork();
    switch(hijos[i].pid){
      case (pid_t)-1:
        /* Error */
        perror("error en fork");
```

```
        exit(EX_OSERR);
        break;

      case (pid_t)0:
        /* proceso hijo */
        /* hijo elige tubo[0] para su extremo. Cierra tubo[1] */
        close(tubo[1]);
        proceso_hijo(tubo[0]);
        exit(EX_OK);

      default:
        /* proceso padre */
        /* elige tubo[1] para su extremo. Cierra tubo[0] */
        close(tubo[0]);
        /* guarda el descriptor de este tubo en la estructura Hijo */
        hijos[i].tubo = tubo[1];
        /* se inventa un número y se lo envía al hijo */
        x = random() % 1000;
        if (write(tubo[1], &x, sizeof(int))!=sizeof(int)) {
          perror("envio x");
          exit(EX_IOERR);
        }
        printf("Envío %d al hijo %d\n", x, hijos[i].pid);
    }
  }

  /* Aquí solo llega el padre. */
  /* Espera a que terminen los hijos y va mostrando */
  /* el resultado que cada uno le envíe. */

  num_terminados = 0;
  while (num_terminados < num_hijos) {
    /* Espera a que termine un hijo. */
    pid_hijo = wait(NULL);
    /* Busca el hijo en la tabla de hijos. */
    for (i = 0; i < num_hijos; i++)
      if (hijos[i].pid == pid_hijo)
        break;
    if (read(hijos[i].tubo, &res, sizeof(double))!=sizeof(double)) {
      perror("lectura resultado");
      exit(EX_IOERR);
    }
    printf("El hijo %d ha devuelto %f\n", pid_hijo, res);
    /* cierra la tubería, porque no la necesita más */
    close(hijos[i].tubo);
    num_terminados++;
  }
  exit(EX_OK);

}

void proceso_hijo(int tubo) {
  int x;
  double raiz;

  if (read(tubo, &x, sizeof(int))!=sizeof(int)) {
    perror("lectura x");
    exit(EX_IOERR);
  }
  raiz = sqrt((double) x);
  if (write(tubo, &raiz, sizeof(double))!=sizeof(double)) {
```

```
    perror("escritura raiz");
    exit(EX_IOERR);
  }
}
```

5.2.4 Función de biblioteca select()

Básicamente, esta función permite que un proceso de usuario solicite al *núcleo* la espera de algunos eventos determinados, reanudándose la ejecución de dicho proceso únicamente cuando se produzca alguno de ellos. Su formato es:

```
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>

int select(int nfds,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *timeout);

void FD_SET(int fd, fd_set &fdset);
void FD_CLR(int fd, fd_set &fdset);
int  FD_ISSET(int fd, fd_set &fdset);
void FD_ZERO(fd_set &fdset);
```

Esta función se puede utilizar para esperar el momento en el que existan datos que leer en un *socket*. Los *socket* se utilizan para comunicar procesos que pueden estar en diferentes máquinas y quedan fuera de estudio en este documento.

`select()` no es únicamente aplicable a los *socket*, sino también a cualquier mecanismo que proporcione el sistema y que sea manejable por un proceso a través de un descriptor de fichero. Es decir, puede ser aplicable a *ficheros* (entre ellos el teclado y la pantalla), *sockets*, *tuberías*, etc. Además, es posible mezclar varios de estos tipos simultáneamente en la misma llamada, si se desea.

A continuación se muestra un ejemplo de utilización de la función `select()` para conseguir que en una comunicación entre un proceso padre y su proceso hijo, vía tubería, el padre pueda conocer en qué momento existen datos que leer de la tubería, y simultáneamente pueda recibir datos del teclado (entrada estándar):

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h >

#define HIJO "hola papi\n"
```

```
main()
{
   int tubo[2];
   ssize_t leidos;
   char buf[100], buf2[100]; /* Buffers de 100 caracteres */
   pid_t pid_hijo;
   int maxfds;
   fd_set lectura;      /* Máscara de lectura */

   /* Creamos la tubería */
   if(pipe(tubo)==-1){
      perror("error en pipe");
      exit(1);
   }

   /* Creamos un hijo */
   fflush(NULL);
   switch(pid_hijo=fork()){

   case (pid_t)-1:
      /* Error */
      perror("error en fork");
      exit(1);
      break;

   case (pid_t)0):
      /* proceso hijo */
      /* hijo elige tubo[0] para lectura/escritura. Cierra tubo[1] */
      close(tubo[1]);

      /* Envía un mensaje al padre */
      if(write(tubo[0], HIJO, strlen(HIJO))!=(ssize_t)strlen(HIJO)){
         perror("hijo:error en write pipe");
         exit(1);
      }
      exit(0);    /* fin normal del hijo */
      break;

   default:
      /* proceso padre */
      /* padre elige tubo[1] para lect/escr ya que hijo usa tubo[0] */
      close(tubo[0]);

      /* Calcula el mayor descriptor de fichero a comprobar */
      maxfds=(tubo[1] > fileno(stdin))? tubo[1] : fileno(stdin);

      buf2[0]='\0';     /* inicializa condición de finalización  */
      do{
         /* Preparación de la máscara de lectura */
         FD_ZERO(&lectura);
         FD_SET(tubo[1], &lectura);
         FD_SET(fileno(stdin), &lectura);

         /* Espera datos que leer en tubo[1] o en el teclado */
         if(select(maxfds+1, &lectura, (fd_set *)NULL,
               (fd_set *)NULL, (struct timeval *)NULL)==-1){
            if(errno==EINTR)        /* si señal, relanzar select() */
               continue;
            perror("padre: error en select");
            exit(1);
         }
```

```
         /* Si hay datos que leer en tubo[1] se leen en buf */
         if(FD_ISSET(tubo[1], &lectura))
            if((leidos=read(tubo[1], buf, sizeof(buf)))==(ssize_t)-1){
               perror("padre:error en read pipe");
               exit(1);
            }

         /* Si hay datos que leer en el teclado se leen en buf2 */
         if(FD_ISSET(fileno(stdin), &lectura))
            if((leidos=read(fileno(stdin), buf2, sizeof(buf2)))
                  ==(ssize_t)-1){
               perror("padre:error en read stdin");
               exit(1);
            }

      }while(buf2[0]!='F');   /* condición de finalización */
      exit(0);                /* fin normal del padre */

   }/*switch*/
}/*main*/
```

El ejemplo anterior, crea un proceso hijo y se prepara a recibir tanto desde la tubería que le comunica con el proceso hijo, como desde el teclado. Para ello prepara una máscara de lectura y ejecuta la función `select()`.

Obsérvese como en el caso de que se produzca alguna señal esta función devuelve el valor `-1`, y asigna el valor `EINTR` a la variable global `errno`. Este caso no es un funcionamiento anormal, por lo que se vuelve a llamar a la función `select()` otra vez para continuar la espera.

En cualquier otro caso, la función retorna cuando haya algo disponible para leer en la tubería o en la entrada estándar. En este caso se ejecutan los `read()` correspondientes. Todo el proceso termina cuando el primer carácter leído desde la entrada estándar es la letra `F`.

5.3 Compartir memoria entre procesos

Existe un mecanismo para que los procesos puedan compartir memoria, basándose en la llamada al sistema `mmap()` cuya función principal es la de proyectar el contenido de un fichero en memoria para poderlo manipular mediate instrucciones de manejo de memoria (se accede con variables) pero también permite definir una zona de memoria que puede ser compartida por varios procesos.

En este apartado se verán los dos ejemplos de `mmap()`, como proyección de ficheros en memoria y como mecanismo de compartición de memoria entre procesos.

Su prototipo es:

```
void *mmap(void *direccion, size_t longitud,
     int proteccion, int indicador, int desccriptor,
     off_t desplazamiento);
```

Parámetros

`direccion`: dirección donde proyectar. Si se pasa el valor NULL el sistema operativo elige una la dirección.

`longitud`: especifica el número de bytes a proyectar.

`protección`: el tipo de acceso (lectura, escritura o ejecución) que debe coincidir con el tipo de acceso usado en la apertura del fichero. Los valores son: PROT_READ, PROT_WRITE o PROT_EXEC.

`indicador`: información sobre la proyección. Las más relevantes son:

- MAP_SHARED: región compartida. Un proceso hijo compartirá esta región con el padre.
- MAP_PRIVATE: región privada. Un proceso hijo no compartirá esta región con el padre sino que obtendrá un duplicado de la misma.
- MAP_FIXED. El fichero debe proyectarse justo en la dirección especificada en el primer parámetro, siempre que éste sea distinto de NULL.
- MAP_ANON: Proyección sin soporte de ficheros.

`descriptor`: representa el descriptor de fichero a proyectar (debe estar previamente abierto). En el caso de no usar soporte de fichero, el valor será -1.

`desplazamiento`: desplazamiento dentro del fichero a partir del cual se realiza la proyección. La proyección se realiza desde *desplazamiento* hasta (*desplazamiento* + *longitud*). En el caso de no usar soporte de fichero, el valor será 0.

Devuelve:

– La dirección de memoria donde se ha proyectado el fichero.

– MAP_FAILED: en caso de error.

Para eliminar una proyección previa o parte de la misma, se usa la llamada al sistema **`munmap`**, con el siguiente prototipo:

```
void munmap(void *direc, size_t lon);
```

Desproyecta parte del espacio de direcciones de un proceso desde la dirección *direc* hasta (*direc* + *lon*).

5.3.1 Ejemplo de proyección de un fichero en memoria

El siguiente programa cuenta el número de apariciones de un carácter en un fichero:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main ( int argc, char **argv ){
   int i, fd, contador = 0;
   char caracter;
   char *org, *p;
   struct stat bstat;

   if ( argc != 3) {
      fprintf ( stderr, "Uso: %s caracter fichero\n", argv[0] );
      return (1);
   }

/* El caracter a contar es el primero del primer argumento */
caracter = argv[1][0];

/* Abre el fichero para lectura */
if ( ( fd = open (argv[2], O_RDONLY) ) < 0 ) {
      perror ( "No puede abrirse el fichero" );
      return (1);
}

/* Averigua la longitud del fichero */
if (fstat ( fd, &bstat) < 0 ) {
      perror ( "Error al procesar la longitud del fichero" );
      close (fd);
      return (1);
}

/* Se proyecta el fichero */
if ( ( org = mmap ( (caddr_t) 0, bstat.st_size, PROT_READ, MAP_SHARED, fd,
(off_t)0) ) == MAP_FAILED) {
      perror ("Error al proyectar el fichero");
      close (fd);
      return (1);
}

/* Se cierra el fichero */
close (fd);

/* Bucle de acceso */
p = org;
for ( i = 0; i < bstat.st_size; i++)
      if ( *p++ == caracter ) contador ++;

/* Se elimina la proyección */
munmap ( org, bstat.st_size );
printf ( "%d\n", contador );
return (0);
}
```

5.3.2 Ejemplo de compartir memoria entre dos procesos

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <stdlib.h>

#define TAM 5  // Nº de elementos de estructuras de tipo Datos

void *memCompartida; // Dirección de memoria compartida

typedef struct {  /* Tipo de datos que se almacenará en la memoria
                  compartida */
   int campo1;
   int campo2;
} Datos;

Datos *Tabla;  /* Variable global que apuntará a la memoria compartida,
                para poder usar la estructura cómodamente */
int main ( int argc, char **argv ){
   int i;
   ……

   //Crear zona de memoria compartida
   if ( (memCompartida = mmap ( NULL, sizeof(Datos) * TAM, PROT_READ |
         PROT_WRITE, MAP_ANON | MAP_SHARED, -1,(off_t)0) )==MAP_FAILED) {
      perror ("Error al crear la Tabla");
      exit(-1);
   }

   Tabla=memCompartida; /* Usaremos la variable Tabla para acceder
                         cómodamente a los elementos de tipo Datos */

   for (i=0;i< TAM;i++){
      Tabla[i].campo1 = 1;
      Tabla[i].campo2 = 2;
   }
   ……
   /* A continuación, si este proceso hace un fork() el proceso hijo tendría
acceso a la variable Tabla, que si bien es una copia de la variable del padre,
estaría apuntando a la misma dirección, con lo que cualquier modificació que
haga el hijo se haria sobre la misma dirección de memoria del padre con lo cual
estaría compartiendo memoria */
```

Hay que destacar que cuando se usa memoria compartida entre dos procesos se deberían utilizar mecanismos de sincronización para el acceso a la misma.