Introducción a la programación de Unix

(Versión 1.32)

Miquel Nicolau i Vila

Índice

1 Introducción

En este documento se introducirán los conceptos básicos de la programación con llamadas a sistema de Unix. En ningún momento, sin embargo, quiere ser un catálogo completo de todas las llamadas a sistema y de su uso, sino que quiere presentar las funciones más significativas del entorno Unix relacionadas con la gestión de los procesos, la entrada/salida y la comunicación entre procesos. El objetivo final es ofrecer las herramientas básicas para entender los rasgos más característicos de Unix y poder así empezar a desarrollar aplicaciones sobre este entorno.

El documento se estructura en cuatro capítulos. Este primer capítulo es una introducción que presenta, de forma general, algunos conceptos que ayudarán a entender los cimientos de la programación sobre Unix. Los otros tres capítulos tratan tres ámbitos fundamentales de todo sistema operativo: la gestión de procesos (capítulo 2), la entrada/salida (capítulo 3) y la comunicación y sincronización entre procesos (capítulo 4).

El esquema de los capítulos 2, 3 y 4 es muy similar. En cada uno de ellos se introduce, primero de todo, los conceptos relacionados con la temática tratada, acto seguido se presentan las llamadas a sistema fundamentales y se acaba con un conjunto de ejemplos que ayudarán a clarificar todo lo que se ha tratado.

La presentación de cada llamada a sistema se estructura en cinco apartados:

1. *Sintaxis*: se presenta la sintaxis en lenguaje C de cada llamada a sistema y se describen los ficheros de definiciones (*include*) necesarios.
2. *Descripción*: se introduce con detalle el funcionamiento de la llamada a sistema y las acciones que se llevan a cabo.
3. *Parámetros*: se explican uno a uno los parámetros utilizados y los posibles valores.
4. *Valor devuelto*: se describen los posibles valores devueltos y su significado.
5. *Errores*: se presentan los errores más significativos que puede producir cada llamada a sistema.

1.1 El manual de Unix

Unix ofrece un manual de usuario estructurado en 8 secciones donde se describen todas sus características. Este manual está disponible en el propio sistema operativo (*man pages*) y se puede acceder a través de la orden **`man`** del intérprete de órdenes. Las secciones tratan los siguientes temas:

Sección 1: órdenes de usuario disponibles desde el intérprete (*shell*)
Sección 2: llamadas a sistema
Sección 3: funciones de biblioteca (*library*) del lenguaje C
Sección 4: dispositivos
Sección 5: formatos de los archivos
Sección 6: juegos
Sección 7: entornos, tablas y macros
Sección 8: mantenimiento del sistema

La descripción de cada concepto del manual se acompaña del identificador de la sección a la cual pertenece. Por ejemplo, la orden *ls* que permite listar el contenido de

un directorio aparecerá como *ls (1)* ya que es una orden del intérprete y, por lo tanto, se describirá en la sección 1. Todas las llamadas a sistema estarán explicadas en la sección 2 y las referencias lo indicarán así: *fork (2)*.

El manual en línea de Unix es una herramienta muy importante de apoyo en el uso del sistema operativo y, en concreto, para el buen conocimiento de las llamadas a sistema.

Algunos ejemplos de órdenes:

- `$ man man` (obtendremos ayuda del propio manual)
- `$ man ls` (obtendremos ayuda sobre la orden *ls*)
- `$ man 2 write` (obtendremos ayuda de la llamada a sistema *write*, en ella se ha especificado la sección 2 -con el parámetro 2- para distinguirla de la orden del intérprete *write (1)*.

1.2 El control de errores

La mayoría de llamadas a sistema de Unix devuelven un valor entero al acabar su ejecución. Un valor 0 o positivo indica un fin correcto de la llamada. Un valor negativo (-1) indica un error en su ejecución.

Cada llamada a sistema puede producir errores diferentes y variados. Para poder saber el error que se ha producido, cada proceso dispone de una variable global llamada *errno* que describe el error producido después de cada llamada a sistema. Para poder controlar correctamente el comportamiento de un proceso, que utiliza llamadas a sistema, es del todo necesario hacer un seguimiento detallado de la ejecución de cada llamada. Por esta razón, es del todo aconsejable verificar la correcta finalización de las llamadas y, en caso contrario, detectar el error que se ha producido.

La estructura general de programación de cualquier llamada a sistema sería la siguiente:

```
#include <errno.h>

int p[2];

...

if (pipe(p) < 0) {

      /* Escribe el error por el canal estándar de errores (2) */

      write(2, "Error pipe\n", strlen("Error pipe\n"));
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}
```

En el código anterior se incluye el fichero *errno.h*, donde se describe la variable global *errno* necesaria para identificar el error producido. El proceso ejecuta la llamada a sistema *pipe* que devolverá el valor -1 en caso de error. Si se produce un error, el código escribirá (*write*) el tipo de error (variable *errno*) y acabará la ejecución del proceso (llama a sistema *exit*).

2 La gestión de procesos

En este capítulo se describirán las características fundamentales de los procesos, sus atributos, la creación y destrucción, junto con las llamadas básicas para su gestión.

2.1 Los procesos

La gestión de procesos es la herramienta fundamental que permite la creación y destrucción de nuevos procesos dentro del sistema operativo. En Unix existe una jerarquía de procesos encabezada por un proceso inicial a partir del cual se genera el resto de procesos del sistema. Este proceso (el proceso *init*) tiene el identificador 1 y tendrá un papel muy importante a lo largo de la vida del sistema tal como se verá más adelante.

Los procesos de Unix tienen un conjunto de atributos, que hay que conocer para poder gestionarlos correctamente, de los cuales hay que destacar los siguientes:

- Identificador del proceso (*PID*)
- Identificador del proceso padre (*parent process* ID)
- Identificador del grupo de procesos *(process gro*up ID)
- Identificador de usuario (*UID*)
- Identificador del grupo de usuarios (*GID*)
- Estado del proceso

2.1.1 Identificador del proceso (*PID*)

El identificador del proceso es un entero positivo único que se asocia a cada proceso en el momento de su creación y que se mantiene hasta su desaparición. Este identificador permitirá gestionar el proceso a lo largo de su vida y hacer el seguimiento de su ejecución.

2.1.2 Identificador del proceso padre (*parent process ID*)

Los procesos de Unix mantienen la relación jerárquica del sistema mediante el identificador de su proceso padre (*parent* process ID). Este identificador permite saber quién ha creado al proceso.

2.1.3 Identificador del grupo de procesos (*process group ID*)

El identificador de grupo indica el grupo de procesos al cual pertenece el proceso. Un grupo de procesos es una agrupación de procesos que facilita la gestión conjunta de algunas funciones como el envío de señales (llamada a sistema *signal*). El proceso líder del grupo es el que define el valor del identificador del grupo, que será el mismo que su identificador de proceso (*PID*). Los grupos de procesos existen mientras exista algún proceso del grupo. Un nuevo proceso mantiene el grupo del proceso padre mientras no se cambie de grupo mediante la llamada a sistema *setpgid*.

2.1.4 Identificador de usuario (*UID*)

Todo proceso pertenece al usuario que lo ha creado y todo usuario de Unix posee un identificador único que lo representa. El identificador de usuario se asignará al proceso en el momento de su creación y le ofrecerá un conjunto de derechos sobre los recursos del sistema.

Todo proceso dispondrá de dos identificadores de usuario:

- el identificador real de usuario (*real user ID*)
- el identificador efectivo de usuario (*effective u*ser ID)

El identificador real de usuario (*real user ID*) no se cambia nunca en toda la vida del proceso y corresponde al identificador del usuario que ha creado el proceso.

El identificador efectivo de usuario (*effective u*ser ID) es el que utiliza el sistema para verificar los derechos del proceso sobre los diferentes recursos del sistema. En el momento de la creación de un proceso, su identificador efectivo de usuario coincide con el identificador real de usuario, pero el identificador efectivo de usuario sí que se puede cambiar de forma controlada a lo largo de la ejecución del proceso. La modificación del identificador efectivo de usuario ofrece una herramienta importante para el acceso restringido de los procesos de usuario a recursos protegidos del sistema.

El cambio de identificador efectivo de usuario lo puede provocar la ejecución (llamada a sistema *exec*) de un fichero ejecutable que tenga activo el bit *setUID* (*set-user-ID*). Si el bit *set-user-ID* está activo, el identificador efectivo de usuario del proceso que ha ejecutado (*exec*) el fichero, tomará como valor al identificador del propietario del fichero ejecutable. El identificador real de usuario no es modificará.

Un ejemplo claro de la utilización del *setUID* se puede encontrar en la orden *passwd*. Esta orden permite que cualquier usuario pueda modificar su clave de acceso al sistema. Esta clave está almacenada en un fichero (*/etc/passwd*) que es propiedad del usuario *root* o *super-user* y que sólo este usuario puede modificar. El fichero ejecutable con la orden *passwd* pertenece también al usuario *root* y tiene el bit *setUID* activo. Por lo tanto, cualquier proceso que ejecute esta orden cambiará su identificador efectivo de usuario que tomará por valor al identificador de usuario *root* y, por lo tanto, podrá acceder al fichero de claves (*/etc/passwd*) y modificarlo.

2.1.5 Identificador del grupo de usuarios (*GID*)

Todo proceso pertenece al grupo de usuarios del usuario que lo ha creado. El identificador del grupo de usuarios se asignará al proceso en el momento de su creación y le ofrecerá un conjunto de derechos sobre los recursos del sistema.

Todo proceso dispondrá de dos identificadores del grupo de usuarios:

- el identificador real del grupo de usuarios (*real group* ID)
- el identificador efectivo del grupo de usuarios *(effective gr*oup ID)

El identificador real del grupo de usuarios (*real group ID*) no se cambia nunca en toda la vida del proceso y corresponde al identificador del grupo de usuarios del usuario que ha creado el proceso.

El identificador efectivo del grupo de usuarios *(effective group ID*) es el que utiliza el sistema para verificar los derechos del proceso sobre los diferentes recursos del sistema. En el momento de la creación de un proceso, su identificador efectivo del grupo de usuarios coincide con el identificador real del grupo de usuarios, pero el identificador efectivo del grupo de usuarios sí que se puede cambiar de forma controlada a lo largo de la ejecución del proceso. La modificación del identificador efectivo del grupo de usuarios ofrece una herramienta importante para el acceso restringido de los procesos de usuario a recursos protegidos del sistema.

El cambio de identificador efectivo del grupo de usuarios lo puede provocar la ejecución (llamada a sistema *exec*) de un fichero ejecutable que tenga activo el bit *setGID* (*set-group-ID*). Si el bit *set-group-ID* está activo, el identificador efectivo del grupo de usuarios del proceso que ha ejecutado (*exec*) el fichero, tomará como valor el identificador del grupo de usuarios del propietario del fichero ejecutable. El identificador real del grupo de usuarios no se modificará.

2.1.6 Estado del proceso

Como en todo sistema operativo, los procesos pueden estar en diferentes estados: en ejecución, bloqueados, preparados, *zombie,* ...

En Unix hay que destacar un estado particular que tienen algunos procesos después de su destrucción: el estado *zombie*. Este estado corresponde a un proceso que ya no puede volver a ejecutarse porque ya ha finalizado, pero que todavía está presente en el sistema porque no ha podido liberar todos sus recursos. La razón del estado *zombie* tiene que ver con la jerarquía de procesos de Unix que se origina en la creación de procesos y se mantiene hasta su desaparición.

Cada proceso es hijo de su proceso padre que es el responsable de liberar los recursos de sus procesos hijo en el momento de su finalización. Para liberar los procesos y eliminarlos del todo del sistema, es necesario que el proceso padre se sincronice (llamada *wait*) con la finalización (*exit*) de sus procesos hijo. En este proceso de sincronización, el proceso hijo informa a su padre de la causa de su muerte al mismo tiempo que libera todos sus recursos y desaparece del sistema. Si un proceso finaliza sin esta sincronización padre-hijo, el proceso pasa al estado *zombie* y se quedará en este estado hasta que pueda liberar los recursos pendientes. Si el proceso padre desapareciera sin haber realizado la sincronización (*wait*) con sus procesos hijo, los procesos hijo pasarán a ser adoptados por el proceso *init* (proceso 1) y este proceso primogénito los liberará del estado *zombie* y desaparecerán del sistema.

Es importante tener en cuenta esta característica de los procesos y evitar, siempre que sea posible, que no quede ningún proceso en estado *zombie* ya que ocupa inútilmente recursos del sistema. Por esta razón, los procesos esperarán con la llamada a sistema *wait* la finalización de sus hijos y, de esta manera, harán desaparecer del sistema los procesos que ya han finalizado.

2.2 La creación de nuevos procesos

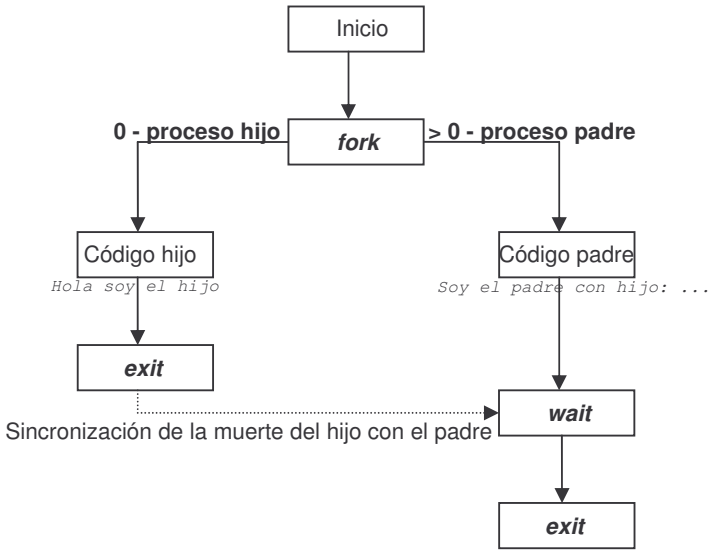
La creación de nuevos procesos es una de las acciones más importantes que permite que se puedan realizar nuevas acciones por parte de los diferentes usuarios. Cada

nuevo proceso se ejecutará de forma concurrente con los otros procesos y tendrá que compartir los recursos (procesador, memoria, ...) del sistema. A diferencia de otros sistemas operativos, la creación de procesos en Unix se realiza de una manera muy sencilla, mediante la llamada a sistema nombrado *fork* que no utiliza ningún parámetro. El resultado de la ejecución de la llamada *fork* es la aparición de un nuevo proceso, hijo del proceso creador y que se ejecutará de forma concurrente con el resto de procesos del sistema, que hereda un gran número de características de su padre. Hay que destacar, por ejemplo, que el nuevo proceso tiene el mismo código, los mismos datos, la misma pila y el mismo valor del contador de programa que el proceso creador. Eso no quiere decir que comparta código y datos, sino que se crea un nuevo proceso donde se copia el código y los datos del proceso que lo ha creado. Sólo hay un dato que es diferente: el valor devuelto por la llamada *fork*.

Efectivamente, la llamada fork acaba dos veces, una vez en el proceso que lo ha invocado y otra, en el nuevo proceso que, como ya se ha dicho, tiene el mismo código que el proceso creador y, por lo tanto, empezará su ejecución justo después de haber acabado la función *fork*. En el proceso padre, la llamada *fork* devuelve el identificador del proceso hijo y en el proceso hijo devuelve el valor 0. Es gracias a los valores diferentes de retorno que se puede distinguir entre padre e hijo y, por lo tanto, se puede modificar el comportamiento de los dos procesos. En el código que sigue se muestra un pequeño programa que crea un nuevo proceso:

```
main()
{
      int process, st, pid;
      char s[80];

      switch (process = fork())
      {
            case -1:

                  /* En caso de error el proceso acaba */

                  sprintf(s, "Error del fork\n");
                  write(2, s, strlen(s));
                  exit(1);

            case 0:

                  /* Proceso hijo - Escribe y acaba */

                  sprintf(s, "Hola soy el hijo\n");
                  write(1, s, strlen(s));
                  exit(0);

            default:

                  /* Proceso padre - Espera fin hijo y escribe */

                  sprintf(s, "Soy el padre con hijo: %d\n", process);
                  write(1, s, strlen(s));
                  pid = wait(&st);
                  exit(0);
      }
}
```

La evolución de los dos procesos se muestra en el siguiente diagrama:



En el código anterior se pueden distinguir claramente los posibles valores devueltos de la llamada *fork*:

- Valor -1: representa un error y que no se ha podido crear el nuevo proceso
- Valor 0: es el valor que se devuelve al nuevo proceso (proceso hijo)
- Valor > 0: es el valor que se devuelve al proceso creador (*pid* del nuevo proceso)

En caso de que la llamada *fork* no haya producido ningún error aparecerá un nuevo proceso que tendrá replicado (no compartido) el mismo código y los mismos datos que el proceso creador y con el mismo valor del contador de programa. Por lo tanto, empezará su ejecución en el punto donde el proceso creador lo ha creado, es decir, justo en el momento de finalizar la llamada *fork*. Por lo tanto, el nuevo proceso empezará su ejecución en el *case* 0 del *switch*, escribirá un mensaje y acabará (*exit*):

```
case 0:

      /* Proceso hijo - Escribe y acaba */

      sprintf(s, "Hola soy el hijo\n");
      write(1, s, strlen(s));
      exit(0);
```

En cambio el padre, después de haber creado al hijo seguirá su ejecución en el *case default* del *switch*, escribirá el identificador del hijo que le ha devuelto la llamada *fork* (variable *process*), esperará la finalización del hijo (*wait*) y acabará (*exit*):

```
default:

      /* Proceso padre - Espera fin hijo y escribe */

      sprintf(s, "Soy el padre con hijo: %d\n", process);
      write(1, s, strlen(s));
      pid = wait(&st);
      exit(0);
```

La salida de los procesos padre e hijo se puede dar en cualquier orden ya que la ejecución es concurrente. Puede ser que escriba primero el padre o que escriba primero el hijo:

```
Soy el padre con hijo: ...
Hola soy el hijo
```

o

```
Hola soy el hijo
Soy el padre con hijo: ...
```

Es importante remarcar la importancia de sincronizarse con la desaparición de los hijos (*wait*). Tal como se ha comentado antes, si el proceso creador no espera la finalización de sus hijos, los procesos que mueren no pueden liberar todos sus recursos y quedan en estado *zombie*. Si en el ejemplo anterior el proceso creador hubiera acabado sin ejecutar la llamada a sistema *wait*, el proceso hijo habría quedado en estado *zombie* hasta que el proceso creador hubiera muerto y el proceso *init* hubiera adoptado el nuevo proceso y hubiera liberado sus recursos.

En otros apartados se irán comentando otros aspectos importantes de la herencia entre procesos relacionados con la entrada/salida y con la comunicación entre procesos.

2.3 La transformación de un proceso (exec)

Como se ha visto en el apartado anterior la creación de procesos de Unix se limita a crear un nuevo proceso idéntico al proceso padre. Sin embargo, cuando se crea un nuevo proceso normalmente se quiere que haga cosas muy diferentes de las que hace su creador. Por esta razón, hay que disponer de una función que nos permita cambiar del todo a un determinado proceso.

La llamada a sistema *exec* permite cambiar todo el código, los datos y la pila de un proceso y cargar un nuevo código y unos nuevos datos almacenados en un fichero ejecutable. Esta llamada es la que realmente permite ejecutar programas después de haber creado un nuevo proceso y, habitualmente, se utilizará inmediatamente después de la llamada *fork*. Una vez acabada la llamada *exec*, el código del proceso que la ha invocado habrá desaparecido del todo y, por lo tanto, la llamada no retornará nunca, es decir, las sentencias del programa que aparezcan después de la llamada nunca no se ejecutarán.

El código que se muestra a continuación crea un nuevo proceso e, inmediatamente después, el proceso hijo ejecuta (llamada a sistema *execlp*) el fichero ejecutable *ls*. Por lo tanto, el nuevo proceso ya no mantiene el mismo código ni los mismos datos que el proceso creador, sino que ha transformado totalmente su código y sus datos según el contenido del fichero ejecutable *ls*. Por esta razón, el proceso no ejecutará nunca las sentencias de después de la llamada, excepto en caso de que *exec* produzca un error y, por lo tanto, el cambio de imagen no se pueda realizar. Esta *mutación* sólo afecta a los datos y al código, en cambio el proceso sigue siendo lo mismo con el mismo identificador (*pid*) y los mismos identificadores de usuario, a menos que el fichero ejecutable tuviera activo el bit *setUID* o el bit *setGID*, tal como se ha explicado previamente.

```
main()
{
      int st;
      char s[80];

      switch (fork())
      {
            case -1:

                  /* En caso de error el proceso acaba */

                  sprintf(s, "Error del fork\n");
                  write(2, s, strlen(s));
                  exit(1);

            case 0:

                  /* Proceso hijo - Ejecutará ls */
                  /* Carga el código de ls */

                  execlp("ls","ls", (char *)0);

                  /* Si llega aquí, ha habido error y acaba */

                  sprintf(s, "Error exec\n");
                  write(2, s, strlen(s));
                  exit(1);

            default:

                  /* Proceso padre - Espera fin hijo y acaba */

                  wait(&st);
                  exit(0);
      }
}
```

2.4 Llamadas a sistema de gestión de procesos

En este apartado se presentan las siguientes llamadas relacionadas con la gestión de procesos:

- *fork*: crea un nuevo proceso
- *exec*: reemplaza la imagen del proceso con una nueva imagen
- *exit*: finaliza un proceso
- *wait*: espera la finalización de un proceso hijo
- *getpid*: obtiene el identificador del proceso
- *getppid*: obtiene el identificador del proceso padre
- *getpgrp*: obtiene el identificador del grupo de procesos del proceso
- *setpgid*: cambia el identificador del grupo de procesos de un proceso

2.4.1 fork

Sintaxis

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

pid_t fork(void);
```

Descripción

La llamada *fork* no tiene ningún parámetro y su función es crear un nuevo proceso. El nuevo proceso (proceso hijo) es una copia exacta del proceso creador (proceso padre). El proceso hijo hereda todo el entorno del proceso padre. Entre los atributos heredados hay que destacar:

- Identificadores de usuario y de grupo, tanto real como efectivo.
- Canales (*file descriptors*) abiertos.
- Programación de los *signals.*
- Segmentos de memoria compartida.
- Bit de *set-user-ID.*
- Bit de *set-group-ID.*
- Identificador de grupo (*process* group Id).
- Directorio de trabajo (*current working directory*).
- Directorio raíz (*root directory*).
- Máscara de creación de ficheros (*umask*).

El proceso hijo se distingue del proceso padre en los siguientes aspectos:

- El identificador de proceso del hijo es diferente del identificador del padre.
- El proceso hijo tiene un identificador de proceso padre diferente (es el identificador del proceso que lo ha creado).
- El proceso hijo tiene su propia copia de los canales (*file descriptors*) del padre. Cada canal del hijo comparte con el canal del padre el mismo puntero en el fichero (puntero de lectura/escritura).
- El conjunto de *signals* pendientes está vacío.
- No se hereda ninguna operación de entrada/salida asíncrona.

Valor devuelto

Si el fork se ejecuta correctamente devolverá el valor 0 al proceso hijo y devuelverá el identificador del proceso hijo (PID) al proceso padre. En caso de error, la llamada devolverá el valor -1 al proceso padre, no se creará ningún proceso y la variable *errno* indicará el error producido.

Errores

La llamada *fork* fallará si:

- *EAGAIN*: se ha superado el número máximo de procesos posibles por usuario o la cantidad total de memoria del sistema disponible es insuficiente.
- *ENOMEM*: no hay espacio de *swap* suficiente.

2.4.2 exec

La llamada a sistema exec se ofrece con diferentes sintaxis que facilitan su utilización en función de los parámetros utilizados: execl, execv, execle, execve, execlp, execvp.

Sintaxis

```
#include <unistd.h>

int execl(char *path, char *arg0..., char *argn, char * /*NULL*/);

int execv(char *path, char *argv[]);

int execle(char *path, char *arg0..., char *argn, char * /*NULL*/, char *envp[]);

int execve(char *path, char *argv[], char * envp[]);

int execlp(char *file, char *arg0..., char *argn, char * /*NULL*/);

int execvp(char *file, char *argv[]);
```

Descripción

Cada una de las funciones de la familia *exec* reemplaza la imagen (código y datos) del proceso que la invoca con una nueva imagen. La nueva imagen se construye a partir de un fichero ejecutable que se pasa como parámetro. No se devuelve ningún valor en caso de que la llamada se ejecute correctamente, ya que la imagen del proceso que la invoca es eliminada por la nueva imagen.

Los canales (*file descriptors*) abiertos del proceso que invoca la llamada permanecen abiertos después del cambio de imagen, excepto aquéllos que tienen el *flag close-on-exec* activo.

Los *signals* definidos, en el proceso que invoca la llamada, con acción por defecto o que tienen que ser ignorados se mantienen igual después del cambio de imagen. Los *signals* programados con alguna función en el proceso que invoca la llamada cambian su programación a la acción por defecto después del cambio de imagen, ya que el código de las funciones con que habían sido programados desaparece después de la llamada *exec*.

Si el bit *set-user-ID* está activo, el identificador efectivo de usuario del proceso con la nueva imagen tomará como valor el identificador del propietario del fichero de imagen. Igualmente si el bit *set-group-ID* está activo, el identificador efectivo de grupo del proceso con la nueva imagen tomará como valor el identificador del grupo del fichero de imagen. Los identificadores reales de usuario y de grupo se mantendrán.

Los segmentos de memoria compartida del proceso que invoca la llamada se desasignarán de la nueva imagen.

La nueva imagen del proceso mantendrá, entre otros, los siguientes atributos:

- El identificador del proceso (*PID*).
- El identificador del proceso padre (*parent* process ID).

- El identificador del grupo (*process gro*up ID)
- El identificador real de usuario y de grupo (*real user ID y real group* ID)
- El directorio de trabajo (*current working directory*).
- El directorio raíz (*root directory*).
- La máscara de creación de ficheros (*umask*).
- Los *signals* pendientes.

Parámetros

- *path* apunta al nombre completo que identifica al nuevo fichero de imagen.
- *file* se utiliza para construir el nombre completo que identifica al nuevo fichero de imagen. Si el parámetro *file* contiene una barra inclinada ('/'), entonces se utiliza como nombre completo del nuevo fichero de imagen. En caso contrario, el camino completo del nombre del fichero se obtiene mediante una búsqueda en los directorios incluidos en la variable de entorno *PATH*.
- Los parámetros representados por *arg() ...* son punteros a cadenas de caracteres. Estos parámetros son la lista de argumentos que se pasarán a la nueva imagen. La lista se acaba con un puntero nulo. El parámetro *arg0* indicará el nombre que se asociará con el proceso iniciado por la función *exec*.
- *argv* es un puntero a una tabla de cadenas de caracteres. La última entrada de la tabla tiene que ser un puntero nulo. Los elementos de la tabla son la lista de argumentos que se pasarán a la nueva imagen. El valor de la entrada *argv[0]* indicará el nombre que se asociará con el proceso iniciado por la función *exec*.
- *envp* es un puntero a una tabla de cadenas de caracteres. La última entrada de la tabla tiene que ser un puntero nulo. Los valores de este parámetro constituyen el entorno para la nueva imagen.

Valor devuelto

La llamada devolverá el valor -1 en el caso de un error en su ejecución y la variable *errno* indicará el error producido. No se devuelve ningún valor en caso de que la llamada se ejecute correctamente, ya que la imagen del proceso que la invoca es eliminada por la nueva imagen.

Errores

La llamada *exec* fallará, entre de otras razones, si:

- *EACCES*: no se tiene permiso para buscar en uno de los directorios que aparecen en el nombre completo del fichero de imagen, o el fichero con la nueva imagen no es un fichero regular, o el fichero con la nueva imagen no se puede ejecutar.
- *EAGAIN*: la cantidad de memoria disponible del sistema en el momento de leer el fichero imagen es insuficiente.
- *EFAULT*: alguno de los parámetros apunta a una dirección ilegal.
- *EINTR*: ha llegado un *signal* durante la ejecución de la llamada *exec*.
- *ELOOP*: se han encontrado demasiados enlaces simbólicos durante la traducción del parámetro *path* o *file*.
- *ENAMETOOLONG*: la longitud del parámetro *path* o *file* excede el tamaño permitido (PATH_MAX).
- *ENOENT*: algún componente del parámetro *path* o *file* no existe o está vacío.
- *ENOMEM*: la nueva imagen del proceso necesita más memoria de la permitida.
- *ENOTDIR*: algún componente del prefijo del parámetro *path* o *file* no es un directorio.

2.4.3 exit

Sintaxis

```
#include <stdlib.h>

void exit(int status);
```

Descripción

La llamada *exit* tiene como función finalizar el proceso que la invoca con las siguientes consecuencias:

- Se cierran todos los canales (*file descriptors*) del proceso.
- Si el proceso padre está ejecutando la llamada *wait* se le notifica la finalización del hijo y se le pasa el valor de los ocho bits de menor peso del parámetro *status*. Si el proceso padre no está ejecutando la llamada *wait*, el estado del hijo (valor del parámetro *status*) se le pasará en el momento que la ejecute.
- Si el proceso padre no está ejecutando la llamada *wait* en el momento que el proceso invoca la llamada *exit*, entonces el proceso que invoca *exit* se transforma en un proceso *zombie*. Un proceso *zombie* es un proceso inactivo que sólo ocupa una entrada de la tabla de procesos y se eliminará completamente en el momento que su proceso padre ejecute la llamada *wait*.
- Se envía un *signal SIGCHLD* a su padre.
- Se libera toda la memoria asignada al proceso.
- Se desasignan los segmentos de memoria compartida.

Parámetros

El parámetro *status* almacena, en sus 8 bits (bits 0377) de menor peso, el valor que el proceso hijo pasará a su padre cuando el proceso padre ejecute la llamada *wait*.

Valor devuelto

La llamada *exit* no retorna nunca al proceso que la invoca.

Errores

No hay ningún error definido.

2.4.4 wait

Sintaxis

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

pid_t wait(int *stat_loc);
```

Descripción

La llamada *wait* bloquea la ejecución del proceso que la invoca hasta que está disponible la información del estado de finalización de alguno de sus hijos, o hasta que llega un *signal* que tiene una función programada o que provoca la finalización del proceso. Si la información del estado de finalización de algún hijo está disponible antes de invocar la llamada, *wait* retornará inmediatamente.

Si *wait* finaliza porque el estado de finalización de algún hijo está disponible, entonces devolverá el identificador del hijo que ha acabado.

Si un proceso acaba sin haber esperado (*wait*) la finalización de sus hijos, el identificador del proceso padre de todos sus hijos tomará el valor 1. Es decir, sus procesos hijo son heredados por el proceso de inicialización (*init*) que se convierte en su padre.

Parámetros

El parámetro *stat_loc* es un puntero donde se almacenará el estado del proceso hijo que ha finalizado.

Si la llamada finaliza porque está disponible el estado de finalización de algún proceso hijo, entonces el estado del proceso hijo que ha finalizado se almacenará en la dirección apuntada por *stat_loc* de la siguiente manera:

- Si el proceso hijo acaba por la ejecución de la llamada *exit*, los ocho bits de menor peso de la variable apuntada por *stat_loc* valdrán 0 y los ocho bits de mayor peso contendrán el valor de los ocho bits de menor peso del argumento pasado a la llamada *exit*.
- Si el proceso hijo acaba a causa de un *signal*, los ocho bits de mayor peso de la variable apuntada por *stat_loc* valdrán 0 y los ocho bits de menor peso contendrán el número del *signal* que ha causado la finalización del proceso.

Valor devuelto

Si *wait* acaba a causa de la finalización de un proceso hijo, entonces la llamada devuelve el identificador del proceso que ha finalizado. De lo contrario devolverá el valor -1 y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *wait* fallará si:

- *ECHILD*: el proceso que invoca la llamada no tiene ningún hijo vivo.

- *EINTR*: la llamada ha sido interrumpido por un *signal*.

2.4.5 getpid

Sintaxis

```
#include <unistd.h>

pid_t getpid(void);
```

Descripción

La llamada *getpid* no tiene ningún parámetro y devuelve el identificador del proceso que la invoca.

Valor devuelto

La llamada devolverá el identificador del proceso que la invoca. No habrá ningún caso que produzca error.

Errores

No hay ningún error definido.

2.4.6 getppid

Sintaxis

```
#include <unistd.h>

pid_t getppid(void);
```

Descripción

La llamada *getppid* no tiene ningún parámetro y devuelve el identificador del proceso padre del proceso que la invoca.

Valor devuelto

La llamada devolverá el identificador del proceso padre del proceso que la invoca. No habrá ningún caso que produzca error.

Errores

No hay ningún error definido.

2.4.7 getpgrp

Sintaxis

```
#include <unistd.h>

pid_t getpgrp(void);
```

Descripción

La llamada *getpgrp* no tiene ningún parámetro y devuelve el identificador del grupo de procesos (*process gro*up ID) del proceso que la invoca.

Valor devuelto

La llamada devolverá el identificador del grupo de procesos del proceso que la invoca. No habrá ningún caso que produzca error.

Errores

No hay ningún error definido.

2.4.8 setpgid

Sintaxis

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

int setpgid(pid_t pid, pid_t pgid);
```

Descripción

La llamada *setpgid* asigna el valor *pgid* al identificador del grupo de procesos (*process group ID*) del proceso con identificador *pid*.

Si *pgid* es igual a *pid*, el proceso con identificador *pid* se convertirá en el líder del grupo. De lo contrario el proceso con identificador *pid* se convertirá en un miembro de un grupo ya existente.

Si *pid* es igual a 0 se utiliza como *pid* el identificador de proceso del proceso que ha realizado la llamada.

Si *pgid* es igual a 0, el proceso con identificador *pid* se convertirá en el líder del grupo de procesos.

Parámetros

- *pid* es el identificador del proceso al cual se le quiere cambiar su identificador del grupo de procesos.
- *pgid* es el valor del identificador del grupo de procesos que se quiere asignar al proceso.

Valor devuelto

Si la llamada se ejecuta correctamente devolverá el valor 0. De lo contrario devolverá el valor -1 y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *setpgid* fallará, entre de otras razones, si:

- *EACCES*: el *pid* corresponde al identificador de un hijo del proceso que invoca la llamada y que ha ejecutado una llamada a sistema *exec*.
- *EINVAL*: el parámetro *pgid* tiene un valor negativo o superior al máximo permitido.
- *ESRCH*: el parámetro *pid* no coincide con el identificador del proceso que invoca la llamada ni con el identificador de ninguno de sus hijos.

2.5 Ejemplos de las llamadas de gestión de procesos

2.5.1 Ejemplo 1

Este ejemplo muestra la creación de un nuevo proceso (*fork*) y la sincronización del proceso padre con la finalización del proceso hijo (*exit* y *wait*). El nuevo proceso se ejecutará de forma concurrente con el proceso padre.

El proceso padre realizará las acciones siguientes:

a) Crea el proceso hijo (*fork*)
b) Escribe su identificador de proceso (*write* y *getpid*)
c) Espera la finalización del proceso hijo (*wait*)
d) Escribe el identificador del hijo que ha muerto (*write*)
e) Acaba (*exit*)

El proceso hijo, por su parte, realizará de forma concurrente con el proceso padre las acciones siguientes:

a) Escribe su identificador de proceso (*write* y *getpid*)
b) Escribe el identificador de proceso de su padre (*write* y *getppid*)
c) Acaba (*exit*)

La escritura b) del padre y las escrituras a) y b) del hijo pueden aparecer en cualquier orden en función de la ejecución concurrente de los dos procesos:

```
Hola soy el hijo: ...          {salida a) del hijo}
Mi padre es: ...               {salida b) del hijo}
Hola soy el padre: ...         {salida b) del padre}
```

o

```
Hola soy el padre: ...         {salida b) del padre}
Hola soy el hijo: ...          {salida a) del hijo}
Mi padre es: ...               {salida b) del hijo}
```

o

```
Hola soy el hijo: ...          {salida a) del hijo}
Hola soy el padre: ...         {salida b) del padre}
Mi padre es: ...               {salida b) del hijo}
```

En cambio, la escritura d) del padre siempre saldrá al final de todo ya que el padre, antes de escribir, espera la finalización del hijo. En la última escritura el padre devolverá el identificador del hijo que ha muerto y que le ha sido devuelto por la llamada a sistema *wait*. Por lo tanto, la última escritura será:

```
El hijo finalizado es: ...     {salida d) del padre}
```

```
#include <errno.h>


void error(char *m)
{
      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}


main()
{
      int st, pid;
      char s[80];

      switch (fork())
      {
            case -1:

                  /* En caso de error el proceso acaba */

                  error("Fork");

            case 0:

                  /* Proceso hijo - Escribe y acaba */

                  sprintf(s, "Hola soy el hijo: %d\n", getpid());
                  write(1, s, strlen(s));
                  sprintf(s, "Mi padre es: %d\n", getppid());
                  write(1, s, strlen(s));
                  exit(0);

            default:

                  /* Proceso padre - Escribe y espera fin hijo */

                  sprintf(s, "Hola soy el padre: %d\n", getpid());
                  write(1, s, strlen(s));
                  pid = wait(&st);

                  /* Escribe y acaba */

                  sprintf(s, "El hijo finalizado es: %d\n", pid);
                  write(1, s, strlen(s));
                  exit(0);
      }
}
```

2.5.2 Ejemplo 2

Este ejemplo muestra la creación de un nuevo proceso (*fork*), el cambio de imagen (*exec*) y la sincronización del padre con la finalización del hijo (*exit* y *wait*). El proceso hijo ejecuta (*exec*) la orden `ls -l /usr/bin` y acaba (*exit*). El proceso padre espera la finalización de su hijo (*wait*) y acaba (*exit*).

```
#include <errno.h>

void error(char *m)
{
      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}


main()
{
      int st;

      switch (fork())
      {
            case -1:

                  /* En caso de error el proceso acaba */

                  error("Fork");

            case 0:

                  /* Proceso hijo - Ejecutará ls */
                  /* Carga el código de ls */
                  /* Recibe como parámetros -l y /usr/bin */

                  execlp("ls","ls","-l",/usr/bin", (char *)0);

                  /* Si llega aquí, execlp ha fallado */

                  error("Ejecutando ls ");

            default:

                  /* Proceso padre */
                  /* Espera que acabe el hijo */

                  wait(&st);
                  exit(0);
      }
}
```

3 La entrada/salida

A lo largo de este capítulo se describirá la estructura general del sistema de ficheros, los diferentes tipos de ficheros, la independencia de dispositivos y la redirección de la entrada/salida, así como las principales llamadas para gestionar las entradas y salidas.

3.1 La entrada/salida y el sistema de ficheros

La entrada/salida de Unix y su sistema de ficheros están íntimamente relacionados tanto por el conjunto de llamadas a sistema uniformes e idénticas para acceder a cualquier dispositivo o fichero, como también por la existencia de diversos tipos de ficheros que incluyen los propios dispositivos. Todo dispositivo de Unix se reconoce en el sistema como un fichero de tipo dispositivo, que puede ser utilizado con las mismas llamadas a sistema (*open, close, read, write* ...) que para los ficheros regulares que contienen información de los usuarios.

Tipo de ficheros

El sistema de ficheros de Unix incluye diversos tipos de ficheros que representan tanto los ficheros convencionales, que contienen información de los usuarios, como los propios dispositivos de entrada/salida y otros recursos del sistema. Los tipos de ficheros más destacados de Unix son:

- *Directorio*: fichero que contiene referencias de otros ficheros y que constituye el elemento fundamental de la jerarquía de ficheros de Unix. Dentro de los ficheros de tipo directorio hay entradas que relacionan el número de un *inode* (el elemento que describe cada fichero) con el nombre (*link*) que se le da al fichero en aquel directorio.
- *Fichero regular*: fichero que contiene información general sin ninguna estructura en particular.
- *Dispositivo*: fichero especial que representa a cada uno de los dispositivos del sistema. Hay dos tipos de dispositivo: los dispositivos de carácter (por ejemplo terminales) y los dispositivos de bloque (por ejemplo discos).
- *Enlace simbólico* (*soft link*): fichero que contiene el nombre de otro fichero al cual representa, de manera que el acceso al enlace simbólico es como acceder al fichero que enlaza.
- *Named pipes*: dispositivo de comunicación entre procesos.

Los nombres de los ficheros (*hard links* y *soft links*)

Una característica del sistema de ficheros de Unix es la posibilidad de que los ficheros puedan tener más de un nombre (*hard links*). Incluso, pueden existir ficheros que no tengan nombre y que sean accesibles y utilizables por los procesos que lo tienen abierto. Gracias a esta característica, se puede hacer visible y accesible un fichero desde diversos puntos del sistema de ficheros. Y eso es posible por la separación existente entre la información del fichero (*inode*) y su nombre.

Los *inodes* son unas estructuras que contienen toda la información de un fichero excepto el nombre, o nombres del fichero, y sus datos, que están almacenados en los

bloques de datos del sistema de ficheros. Dentro del *inode* se puede encontrar, entre otras, las siguientes informaciones:

- Identificador del propietario del fichero
- Identificador del grupo de usuarios del fichero
- Tamaño del fichero
- Fecha de creación, fecha del último acceso y fecha de la última modificación
- Tipo de fichero
- Permisos de acceso al usuario, al grupo y al resto de usuarios
- Número de nombres *(hard links*) del fichero
- Punteros a los bloques de datos

Los *inodes* están numerados del 1 al número máximo de *inodes* de un determinado sistema de ficheros.

El nombre del fichero está almacenado dentro de los directorios. Cada entrada a un directorio es una asociación entre un número de *inode* y un nombre. De esta manera se pueden tener tantos nombres como se quiera de cada fichero. Sólo hay que asociar el mismo número de *inode* (que representa al fichero) con diferentes nombres dentro de los directorios que se quiera.

Entrada de un directorio

Número de *inode*	Nombre del fichero

El número de nombres *(hard links*) de un fichero se guardará en el *inode* correspondiente y servirá para que el sistema operativo pueda detectar cuando un fichero ya no tiene ningún nombre y, por lo tanto, cuando un fichero se puede eliminar del sistema. Sin embargo, que un fichero no tenga ningún nombre no es una condición suficiente para que desaparezca su *inode* y la información que contiene. Para poder borrar del todo un fichero hace falta, además de no tener ningún nombre, que no haya ningún proceso que lo esté utilizando (fichero abierto). Por lo tanto, en Unix es posible que un proceso tenga un fichero abierto, borre el último nombre y siga trabajando con él sin que ningún otro proceso pueda acceder, ya que no dispone de ningún nombre visible en el sistema de ficheros. Esta manera de trabajar es usual entre los programas que utilizan ficheros temporales durante su ejecución. En este caso, el fichero desaparece cuando lo cierra el último proceso que lo tenía abierto (ved el ejemplo 2 de este capítulo).

Unix utiliza la palabra *link* (enlace) para referirse a los nombres de un fichero y distingue entre los *hard links* (que son los que se acaban de explicar) que están contabilizados dentro del *inode* de cada fichero y los *soft links* (enlaces simbólicos). Los *soft links* son un tipo especial de fichero que se utiliza para poder acceder a otro fichero desde cualquier lugar del sistema de ficheros. Los *soft links* contienen el nombre del fichero al cual apuntan y tienen un conjunto de características que los diferencian de los *hard links*:

1. Los *soft links* son ficheros, en cambio los *hard links* no son nada más que una entrada en un directorio que asocia un nombre con un número de *inode*.
2. Los *soft links* no son conocidos por el fichero que representan. A diferencia de los *hard links* que son contabilizados dentro del inode de cada fichero,

los *soft links* se crean y se destruyen sin que el fichero apuntado tenga ninguna constancia. Por esta razón, es posible que puedan existir *soft links* que apunten a ficheros inexistentes, porque los ficheros apuntados hayan desaparecido del sistema después de la creación del *soft link*.

3. Los *soft links* se pueden situar en un sistema de ficheros diferente al del fichero que representan. En cambio los *hard links* (las parejas "*inode* - nombre" dentro de los directorios) tienen que estar situadas todas dentro del mismo sistema de ficheros, para evitar así las ambigüedades con los números de *inode* que coinciden entre sistemas de ficheros diferentes.

3.2 La independencia de dispositivos y la redirección de la entrada/salida

Para garantizar la independencia de dispositivos de las aplicaciones, es necesario disponer de un conjunto de llamadas homogéneas de entrada/salida y de dispositivos virtuales (canales) que puedan ser asociados con cualquier dispositivo real o fichero. Unix ofrece un conjunto de llamadas idénticas para gestionar la entrada/salida independientemente del dispositivo o fichero que se esté utilizando. Estas llamadas utilizan dispositivos virtuales que pueden asociarse a cualquier dispositivo real o fichero.

Los dispositivos virtuales (o canales) de Unix llamados *file descriptors* se agrupan en una tabla independiente para cada proceso. Cada dispositivo virtual se identifica con el valor de entrada en la tabla, de manera tal que el primer dispositivo virtual será el canal 0 y así sucesivamente. Las llamadas a sistema de Unix de lectura (*read*) y escritura (*write*) utilizan exclusivamente los dispositivos virtuales y, de esta manera, se independizan de los dispositivos reales o ficheros que estén asociados en cada *file descriptor*. Para asociar los dispositivos virtuales con un determinado dispositivo o fichero se utiliza la llamada *open* (abrir). Para liberar un determinado dispositivo se utiliza la llamada a sistema *close* (cerrar). Por lo tanto, antes de poder utilizar un determinado dispositivo o fichero habrá que abrirlo (*open*) para así asociar un determinado dispositivo virtual con el fichero o dispositivo real. Una vez tengamos el dispositivo virtual asociado ya podremos escribir (write) o leer (read) sobre este dispositivo virtual. Una vez hayamos finalizado con el uso del dispositivo real o fichero podremos liberarlo junto con el canal virtual utilizando la llamada *close*.

En la página siguiente se muestra el código de un proceso que abre un fichero existente y lee todo su contenido carácter a carácter. Una vez ha acabado la lectura, se cierra el canal y se liberan los recursos. Hay que fijarse en que la llamada *open* devuelve el valor del primer dispositivo virtual libre de la tabla de canales del proceso que, en este caso, seguramente será el canal 3 tal como se explica en las siguientes líneas.

Por convenio, Unix considera que los canales 0, 1 y 2 se utilizarán como canales estándares de entrada/salida:

- 0: entrada estándar (*stdin*)
- 1: salida estándar (*stdout*)
- 2: salida estándar de error (*stderr*)

En el ejemplo de la página siguiente se puede ver que la función *error* escribe los mensajes de error por el canal de salida estándar de error (dispositivo virtual 2 -

stderr). En cambio, el contenido del fichero se escribe carácter a carácter por el canal estándar de salida (dispositivo virtual 1 - *stdout*).

Todos los procesos que se ejecutan desde el intérprete de órdenes de Unix tienen abiertos los canales 0, 1 y 2, ya que los nuevos procesos heredan de sus padres una tabla de canales con los mismos canales abiertos que el proceso creador. El intérprete de órdenes tiene abiertos los canales estándares y, por lo tanto, todos sus hijos también los tendrán abiertos y asociados con los mismos dispositivos en el momento de su creación. Asumiendo este hecho, se podría afirmar que la llamada *open* del ejemplo devolvería el canal 3 (primera entrada libre del proceso) si fuera el intérprete de órdenes quien creara aquel proceso.

```
#include <fcntl.h> /* definiciones O_RDONLY, O_WRONLY ... */
#include <errno.h>

void error(char *m)
{

      /* Escribe los errores por el canal estándar de errores (2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}


main()
{
      int fd, n;
      char c;

      if ((fd = open("Datafile.dat, O_RDONLY)) < 0)
            error("Apertura del fichero");

      /* Lee del fichero mientras haya información */
      /* y escribe el contenido por la salida estándar */

      while ((n=read(fd, &c, 1)) > 0)
            write(1, &c, 1);

      if (n<0)
            error("Lectura del fichero");

      close (fd);
}
```

Vamos a suponer que el ejemplo anterior está almacenado en un fichero ejecutable llamado *testprogram* y que queremos ejecutarlo pero con la salida estándar redireccionada hacia otro fichero llamado *output.dat*. Desde el intérprete de órdenes habría que invocar la línea siguiente:

```
$ testprogram > output.dat
```

El símbolo > representa la redirección de la salida estándar del proceso hacia el fichero indicado. Para realizar esta redirección desde el programa se podría hacer con el código siguiente:

```
#include <fcntl.h> /* definiciones O_RDONLY, O_WRONLY ...         */
#include <errno.h>

void error(char *m)
{
      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}

main()
{
      int st;
      char s[80];

      switch (fork())
      {
            case -1:

                  /* En Caso de error el proceso acaba */

                  error("Error del fork");

            case 0:

                  /* Se cierra el canal estándar de salida */

                  close (1);

                  /* Se crea fichero que se asignará al dispositivo */
                  /* virtual 1 (stdin) que es el primero libre */

                  if (open("output.dat", O_WRONLY|O_CREAT, 0600) < 0)
                        error("Apertura del fichero");

                  /* A partir de este momento el stdout del nuevo */
                  /* proceso ya está redireccionado hacia el fichero */

                  /* Proceso hijo - Ejecuta testprogram */

                  execlp("testprogram "testprogram", (char *)0);

                  /* Si llega aquí, execlp ha fallado */

                  error("Ejecutando testprogram");

            default:

                  /* Proceso padre - Espera fin hijo y acaba */

                  wait(&st);
                  exit(0);
      }
}
```

En el programa anterior, para redireccionar la salida estándar del nuevo proceso se procede de la forma siguiente:

1. Se cierra (*close*) el canal de salida estándar (*file descriptor* 1) a fin de que quede libre y, por lo tanto, se pueda asignar en el momento de hacer una llamada *open*.
2. Con la llamada *open* se crea el fichero hacia al cual se quiere redireccionar la salida estándar. Esta llamada buscará el primer canal (dispositivo virtual) libre de la tabla de canales del proceso. El canal 1 se ha cerrado previamente y, por lo tanto, será el primer canal libre. Así pues, a partir de este momento el canal estándar de salida estará asociado al fichero *output.dat* y todo lo que se escriba por la salida estándar del proceso (*file descriptor* 1), se escribirá sobre este fichero.
3. Una vez redireccionada la salida estándar del nuevo proceso, se invoca la llamada a sistema *exec* con el fichero *testprogram* como parámetro, con lo cual el proceso cambiará su código por el código definido en el fichero. La llamada *exec* no produce ninguna modificación en la tabla de canales del proceso y, por lo tanto, el proceso leerá todo el fichero *Datafile.dat* y lo escribirá carácter a carácter por su salida estándar que, en este caso, está redireccionada hacia el fichero *output.dat*.

3.3 La tabla de ficheros abiertos y el puntero de lectura/escritura

En el apartado anterior se ha explicado la manera de redireccionar la entrada/salida de un proceso gracias a la existencia de los dispositivos virtuales y de un conjunto de llamadas homogéneas. También se ha visto que la tabla de canales o dispositivos virtuales se hereda entre padres e hijos, de manera tal que los canales abiertos en el proceso padre también están abiertos y apuntando a los mismos dispositivos en el proceso hijo.

Cada canal está asociado a un dispositivo o a un fichero a través de una entrada en una tabla llamada tabla de ficheros abiertos. Esta tabla es global para todo el sistema y, por lo tanto, es la misma para todos los procesos. Dentro de esta tabla de ficheros abiertos se almacena, entre otras cosas, el puntero de lectura/escritura de los ficheros abiertos.

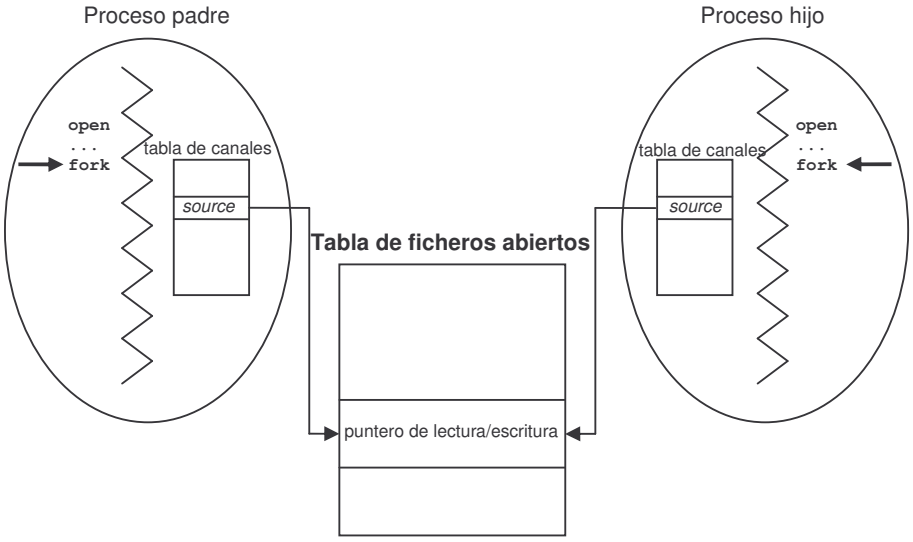
El puntero de lectura/escritura indica la posición donde se hará la siguiente lectura o escritura sobre el fichero correspondiente. En el momento de abrir un fichero, el puntero de lectura/escritura se sitúa, por defecto, al inicio del fichero. A medida que se lee o se escribe en un fichero, el puntero de lectura/escritura va avanzando automáticamente tantas posiciones como bytes se hayan leído o se hayan escrito. También se puede cambiar el puntero de lectura/escritura con un llamada a sistema específica para este propósito (*lseek*).

Cada vez que se abre o se crea un fichero, además de ocupar una entrada de la tabla de canales (*file descriptors*) del proceso correspondiente, también se ocupa una nueva entrada de la tabla de ficheros abiertos que se asocia con el canal abierto. Dentro de la entrada de la tabla de ficheros abiertos se guardará el puntero de lectura/escritura situado al inicio del fichero abierto, a menos que se indique otra opción.

En la creación de procesos, el nuevo proceso no sólo hereda el contenido de la tabla de canales del proceso padre, sino que sus canales abiertos apuntan también a las mismas entradas de la tabla de ficheros abiertos que el padre. Por lo tanto, padre e hijo compartirán el mismo puntero de lectura/escritura de los ficheros abiertos que el hijo haya heredado en el momento de su creación.

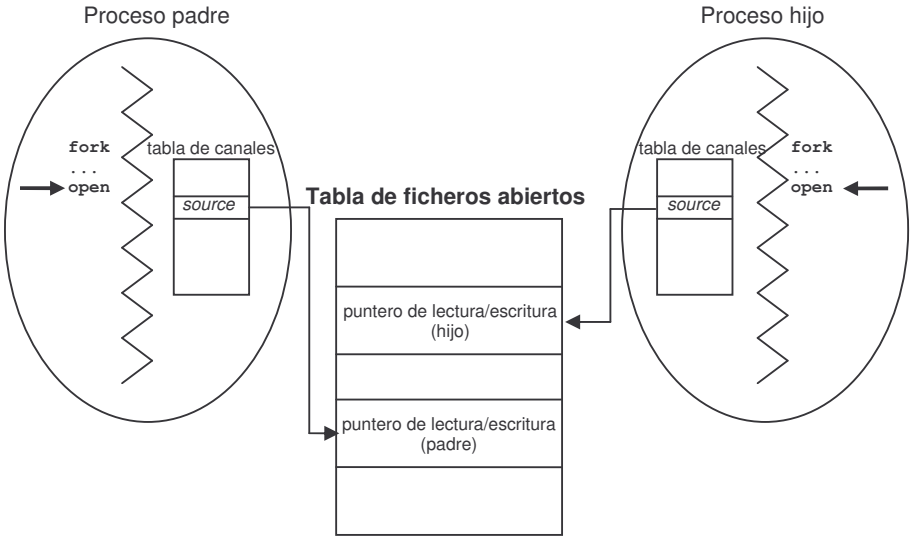
En ejemplo 4 y 5 de este capítulo se pueden ver dos programas que muestran la diferencia entre compartir un fichero ya abierto entre proceso padre e hijo (ejemplo 4), o abrir dos veces y de forma independiente el mismo fichero entre padre e hijo (ejemplo 5).

En el ejemplo 4, el proceso padre abre un fichero antes de la creación del hijo (canal *source*). Por lo tanto, cuando se crea el nuevo proceso, los dos procesos comparten el mismo puntero de lectura/escritura.



En la figura anterior se puede ver el estado de la tabla de canales de los dos procesos y el estado de la tabla de ficheros abiertos después de la creación del proceso hijo. El proceso hijo hereda la misma tabla de canales que el padre y cada canal heredado apunta a la misma entrada compartida de la tabla de ficheros abiertos. Por esta razón, los punteros de lectura/escritura de los ficheros heredados son compartidos entre padre e hijo.

En el ejemplo 5, el proceso padre abre un fichero (canal *source*) después de haber creado el proceso hijo. El proceso hijo abre también el mismo fichero. El resultado es que los dos procesos acceden al mismo fichero con punteros de lectura/escritura independientes:



En la figura anterior se puede ver el estado de la tabla de canales de los dos procesos y el estado de la tabla de ficheros abiertos, después de la creación del proceso hijo y después de que los dos procesos hayan abierto el mismo fichero de forma independiente. En este caso, proceso padre e hijo tienen una entrada en la tabla de ficheros abiertos diferente y, por lo tanto, no comparten el puntero de lectura/escritura en el acceso al fichero.

3.4 Llamadas a sistema de entrada/salida

En este apartado se presentan las siguientes llamadas relacionadas con la entrada/salida y el sistema de ficheros:

- *creat*: crea un nuevo fichero
- *open*: abre un fichero
- *close*: cierra un canal (*file descriptor*)
- *read*: lee de un fichero
- *write*: escribe en un fichero
- *lseek*: sitúa el puntero de lectura/escritura
- *dup*: duplica un canal abierto
- *unlink*: borra una entrada (*link*) de un directorio

3.4.1 creat

Sintaxis

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

int creat(const char *path, mode_t mode);
```

Descripción

La llamada *creat* crea un nuevo fichero regular o rescribe un fichero existente.

Si el fichero existe, su longitud pasa a ser 0 y no se modifica su propietario. Si el fichero no existe, el identificador del propietario del nuevo fichero será el identificador de usuario efectivo del proceso que invoca la llamada y el identificador de grupo del fichero será el identificador efectivo de grupo del proceso.

Los valores de los bits correspondientes a los permisos de acceso serán los indicados en el parámetro *mode* y modificados según la máscara de creación (*umask*): se hará una *AND* bit a bit con la máscara de creación complementada y, por lo tanto, todos los bits con valor 1, en la máscara de creación del proceso, tomarán el valor 0 en la máscara de permisos.

Si la llamada acaba correctamente, se devuelve un canal (*file descriptor*) de sólo escritura con el puntero de lectura/escritura situado al inicio del fichero y el fichero queda abierto para escritura, aunque el parámetro *mode* no lo permita.

Esta llamada es equivalente a:

open(path, O_WRONLY | O_CREAT | O_TRUNC, mode)

Parámetros

- *path* apunta al nombre completo del fichero.
- *mode* representa una máscara de bits que describe los permisos con que se creará el fichero, una vez modificada según el valor de la máscara de creación (*umask*).

Valor devuelto

Si la llamada acaba correctamente se devuelve un valor entero no negativo que corresponde al primer canal (*file descriptor*) libre disponible del proceso. De lo contrario se devuelve el valor negativo -1, no se crea ni modifica ningún fichero y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *creat* fallará, entre otras razones, si:

- *EACCES*: el proceso no tiene permiso para buscar en uno de los componentes directorio del nombre del fichero, el fichero no existe y el directorio donde se

tiene que crear el nuevo fichero no permite escribir o el fichero existe pero no se tienen permisos de escritura.

- *EDQUOT*: no se puede crear el fichero porque el usuario no dispone de más cuota de espacio de bloques de disco o no dispone de más cuota de *inodes* en el sistema de ficheros.
- *EFAULT*: el parámetro *path* apunta a una dirección ilegal.
- *EINTR*: ha llegado un *signal* durante la ejecución de la llamada *creat*.
- *EISDIR*: existe un directorio con el mismo nombre que el fichero que se quiere crear.
- *EMFILE*: el proceso tiene demasiados ficheros abiertos.
- *ENOENT*: algún componente directorio del parámetro *path* no existe o el parámetro está vacío.
- *ENOTDIR*: algún componente del prefijo del parámetro *path* no es un directorio.

3.4.2 open

Sintaxis

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

int open(const char *path, int oflag /* mode_t mode */...);
```

Descripción

La llamada *open* abre un fichero mediante el establecimiento de una conexión entre el fichero (*path*) y un canal (*file descriptor*) del proceso.

Se crea una nueva entrada en la tabla de ficheros abiertos que representa al fichero abierto y el nuevo canal (*file descriptor*) apunta a esta entrada.

Si la llamada acaba correctamente se devuelve un canal (*file descriptor*) con el puntero de lectura/escritura situado al inicio del fichero. La modalidad de acceso del fichero se establece según el valor del parámetro *oflag*. El parámetro *mode* sólo se utiliza si el parámetro *oflag* incluye el valor O_CREAT.

El valor del parámetro *oflag* se construye con una *OR* bit a bit de todos los valores incluidos como parámetros.

Parámetros

- *path* apunta al nombre completo de un fichero.
- *oflag* indicará la modalidad de acceso del fichero abierto y puede tomar, entre otros, los siguientes valores:
  - Tiene que tener siempre únicamente uno de estos tres valores:
    - *O_RDONLY*: fichero abierto sólo para lectura.
    - *O_WRONLY*: fichero abierto sólo para escritura
    - *O_RDWR*: fichero abierto para lectura y escritura.
  - Puede utilizarse cualquier combinación de los siguientes valores:
    - *O_APPEND*: el puntero de lectura/escritura se situará al final del fichero
    - *O_CREAT*: crea el fichero si no existe. Si se utiliza este valor hay que añadir el parámetro *mode* a la llamada.
    - *O_EXCL*: si se incluyen los valores *O_CREAT* y *O_EXCL*, la llamada *open* fallará si el fichero ya existe.
    - *O_NONBLOCK o O_NDELAY*: si se incluye alguno de estos dos valores las lecturas (*read*) y escrituras (*write*) posteriores no provocarán bloqueo. Si aparecen los dos valores *O_NONBLOCK* tiene preferencia.
    - *O_TRUNC*: si el fichero existe, es un fichero regular y se abre con *O_RDWR* o *O_WRONLY*, entonces su longitud pasa a ser 0. El resultado de utilizar *O_TRUNC* con *O_RDONLY* no está definido.
- *mode* representa una máscara de bits que describe los permisos con que se abrirá el fichero, una vez modificada según el valor de la máscara de creación (*umask*).

Valor devuelto

Si la llamada acaba correctamente se devuelve un valor entero no negativo que corresponde al primer canal (*file descriptor*) libre disponible del proceso. De lo contrario se devuelve el valor negativo -1, no se crea ni modifica ningún fichero y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *open* fallará, entre otras razones, si:

- *EACCES*: el proceso no tiene permiso para buscar en uno de los componentes directorio del nombre del fichero, o el fichero existe y los permisos especificados por *oflag* no están permitidos, o el fichero no existe y no se permite escribir en el directorio donde tiene que crearse, o se ha especificado *O_TRUNC* y no se tienen permisos de escritura.
- *EDQUOT*: el fichero no existe, se ha especificado *O_CREAT* y no se puede crear porque el usuario no dispone de más cuota de espacio de bloques de disco, o no dispone de más cuota de *inodes* en el sistema de ficheros.
- *EEXIST*: se han incluido los valores *O_CREAT* y *O_EXCL* y el fichero ya existe.
- *EFAULT*: el parámetro *path* apunta a una dirección ilegal.
- *EINTR*: ha llegado un *signal* durante la ejecución de la llamada *open*.
- *EISDIR*: el fichero es un directorio y se intenta acceder con *O_WRONLY* o *O_RDWR*.
- *EMFILE*: el proceso tiene demasiados ficheros abiertos.
- *ENOENT*: no se ha incluido el valor *O_CREAT* y el fichero no existe, o el valor *O_CREAT* se ha incluido pero algún componente directorio del parámetro *path* no existe o el parámetro está vacío.
- *ENOTDIR*: algún componente del prefijo del parámetro *path* no es un directorio.

3.4.3 close

Sintaxis

```
#include <unistd.h>

int close(int fildes);
```

Descripción

La llamada *close* cierra el canal (*file descriptor*) indicado en el parámetro *fildes*. Cerrar el canal significa hacerlo disponible para que pueda ser asignado posteriormente por otras llamadas como *open*.

Una vez cerrados todos los canales asociados a una *pipe*, los datos que todavía pudieran estar en la *pipe* son eliminados. Una vez cerrados todos los canales asociados a un fichero, si el número de enlaces (*links*) del fichero es cero, el espacio ocupado por el fichero en el sistema de ficheros se libera y el fichero ya no será accesible nuevamente.

En el momento que se cierran todos los canales asociados a una entrada de la tabla de ficheros abiertos, esta entrada se liberará.

Parámetros

fildes indica el canal (*file descriptor*) que se quiere cerrar.

Valor devuelto

Si la llamada acaba correctamente devuelve el valor 0. De lo contrario devolverá el valor -1 y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *close* fallará, entre otras razones, si:

- *EBADF*: el parámetro *fildes* no es un canal (*file descriptor*) válido.
- *EINTR*: ha llegado un *signal* durante la ejecución de la llamada *close*.
- *EIO*: se ha producido un error de entrada/salida mientras se leía o se escribía en el sistema de ficheros.

3.4.4 read

Sintaxis

```
#include <unistd.h>

ssize_t read(int fildes, void *buf, size_t nbyte);
```

Descripción

La llamada *read* lee de un fichero. La llamada *read* intenta leer el número de bytes indicados en el parámetro *nbyte* del fichero asociado al canal abierto (*file descriptor*) *fildes*. Los bytes leídos se almacenan en el buffer apuntado por el parámetro *buf*.

Si el parámetro *nbyte* es 0, la llamada devolverá el valor 0 y no tendrá ningún otro efecto.

En los ficheros donde se permite situar (*lseek*) el puntero de lectura/escritura (por ejemplo los ficheros regulares), la llamada *read* empezará la lectura en la posición indicada por el desplazamiento del puntero de lectura/escritura asociado al canal *fildes*. Al final de la lectura, el desplazamiento del puntero de lectura/escritura del fichero se incrementará el número de bytes que se hayan leído.

En los ficheros donde no se permite situar el puntero de lectura/escritura (por ejemplo los terminales), la llamada *read* siempre leerá de la posición actual. En este caso el desplazamiento del puntero de lectura/escritura asociado al fichero estará indefinido.

Si se intenta leer del final de fichero (*end-of-file*) o más allá del final de fichero, no se producirá ninguna lectura.

La lectura de una *pipe* vacía producirá los siguientes efectos:

- Si no hay ningún proceso que tenga la *pipe* abierta para escritura, la llamada *read* devolverá el valor 0 para indicar final de fichero (*end-of-file*).
- Si algún proceso tiene la *pipe* abierta para escritura y el *flag O_NDELAY* de la *pipe* está activo (*open*), la llamada *read* devolverá el valor 0.
- Si algún proceso tiene la *pipe* abierta para escritura y el *flag O_NONBLOCK* de la *pipe* está activo (*open*), la llamada *read* devolverá el valor -1 y la variable *errno* valdrá *EAGAIN*.
- Si los *flags O_NDELAY* y *O_NONBLOCK* de la *pipe* no están activos (*open*), la llamada *read* se bloquea hasta que algún proceso escriba datos o hasta que todos los procesos que la tienen abierta para escritura la cierren.

La lectura de un fichero asociado con un terminal que no tenga datos disponibles producirá los siguientes efectos:

- Si el *flag O_NDELAY* del terminal está activo (*open*), la llamada *read* devolverá el valor 0.
- Si el *flag O_NONBLOCK* del terminal está activo (*open*), la llamada *read* devolverá el valor -1 y la variable *errno* valdrá *EAGAIN*.
- Si los *flags O_NDELAY* y *O_NONBLOCK* del terminal no están activos (*open*), la llamada *read* se bloqueará hasta que haya datos disponibles.

La llamada *read* lee datos que se han escrito previamente. Sin embargo, si se lee alguna parte de un fichero anterior al final de fichero (*end-of-file*) que no ha sido nunca escrita, *read* devolverá bytes con el valor 0. Esta situación puede pasar al situar (*lseek*) el puntero de lectura/escritura más allá del final de fichero y escribir alguna información. El espacio situado en medio devolverá, al ser leído, bytes con el valor 0 hasta que se escriba alguna cosa explícitamente.

Si la llamada acaba correctamente y el parámetro *nbyte* es mayor que 0, entonces devolverá el número de bytes leídos. Este número nunca será superior al valor del parámetro *nbyte,* pero sí que puede ser inferior si no hay tantos bytes disponibles para ser leídos o la llamada ha sido interrumpida por un *signal*.

Si la llamada *read* es interrumpida por un *signal* antes de que haya leído nada, retornará el valor -1 y la variable *errno* indicará el error que se ha producido.

Si la llamada *read* es interrumpida por un *signal* después de haber leído alguna cosa, devolverá el número de bytes leídos.

Parámetros

- *fildes* canal abierto asociado al fichero del cual se quiere leer.
- *buf* puntero a un espacio de memoria donde se guardarán los bytes leídos.
- *nbyte* número de bytes que se quieren leer.

Valor devuelto

Si la llamada *read* acaba correctamente, devolverá un valor no negativo que indicará el número de bytes realmente leídos del fichero asociado con el canal (*file descriptor*) *fildes*. Este número nunca será superior al valor del parámetro *nbytes*. De lo contrario devolverá el valor -1 y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *read* fallará, entre otras razones, si:

- *EBADF*: el parámetro *fildes* no es un canal abierto (*file descriptor*) válido para lectura.
- *EFAULT*: el parámetro *buf* apunta a una dirección ilegal.
- *EINTR*: ha llegado un *signal* durante la ejecución de la llamada *read* y no ha habido ninguna lectura.
- *EIO*: se ha producido un error del dispositivo físico de entrada/salida.

3.4.5 write

Sintaxis

```
#include <unistd.h>

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

Descripción

La llamada *write* escribe en un fichero. La llamada *write* intenta escribir el número de bytes indicados en el parámetro *nbyte*, que están almacenados en el buffer apuntado por el parámetro *buf*, en el fichero asociado al canal abierto (*file descriptor*) *fildes*.

Si el parámetro *nbyte* es 0, la llamada devolverá el valor 0 y no tendrá ningún otro efecto.

En los ficheros donde se permite situar (*lseek*) el puntero de lectura/escritura (por ejemplo los ficheros regulares), la llamada *write* empezará la escritura en la posición indicada por el desplazamiento del puntero de lectura/escritura asociado al canal *fildes*. Al final de la escritura, el desplazamiento del puntero de lectura/escritura del fichero se incrementará el número de bytes que se hayan escrito.

En los ficheros donde no se permite situar el puntero de lectura/escritura (por ejemplo los terminales), la llamada *write* siempre escribirá en la posición actual. En este caso el desplazamiento del puntero de lectura/escritura asociado al fichero estará indefinido.

Si el *flag O_APPEND* (*open*) está activo, el puntero de lectura/escritura se situará al final de fichero antes de cada escritura.

En caso de que no se puedan escribir el número de bytes indicado al parámetro *nbyte*, porque no hay espacio disponible o se ha superado algún límite indicado por el sistema, la llamada *write* sólo escribirá el número de bytes que sea posible y devolverá este número.

Si la llamada *write* es interrumpida por un *signal* antes de que haya escrito nada, retornará el valor -1 y la variable *errno* indicará el error que se ha producido.

Si la llamada *write* es interrumpida por un *signal* después de haber escrito algo, devolverá el número de bytes leídos.

La ejecución correcta de la llamada *write* sobre un fichero regular tendrá las siguientes consecuencias:

- Si se hace uno lectura (*read*) de las posiciones del fichero modificadas por la llamada *write*, devolverá los datos escritos por la llamada *write* en estas posiciones.
- Si se hacen escrituras sobre posiciones previamente ya escritas, los valores de la última escritura sustituirán los valores existentes.

Las escrituras en *pipes* tienen las mismas consecuencias que en los ficheros regulares salvo las siguientes excepciones:

- No hay ningún puntero de lectura/escritura asociado con la *pipe* y, por lo tanto, todas las escrituras se añaden al final de la *pipe*.
- La escritura en una *pipe* es indivisible y, por lo tanto, garantiza que los datos de ninguna otra escritura realizada por otros procesos, en la misma *pipe*, pueda mezclarse con los datos escritos por la llamada *write*.
- Si los *flags O_NDELAY* y *O_NONBLOCK* de la *pipe* no están activos (*open*), la llamada *write* puede bloquear al proceso (por ejemplo si la *pipe* está llena), pero al final de la escritura habrá escrito todos los bytes indicados y, por lo tanto, devolverá el valor *nbyte*.
- Si los *flags O_NDELAY* y *O_NONBLOCK* de la *pipe* están activos (*open*), la llamada *write* no bloqueará nunca al proceso. Si la llamada puede escribir los bytes indicados, devolverá *nbyte*. De lo contrario, si el *flag O_NONBLOCK* está activo, devolverá el valor -1 y la variable *errno* valdrá *EAGAIN*, o si el *flag O_NDELAY* está activo, devolverá el valor 0.

Parámetros

- *fildes* canal abierto asociado al fichero en el cual se quiere escribir.
- *buf* puntero a un espacio de memoria donde se guardan los bytes que se quieren escribir.
- *nbyte* número de bytes que se quieren escribir.

Valor devuelto

Si la llamada *write* acaba correctamente devolverá un valor no negativo que indicará el número de bytes realmente escritos en el fichero asociado con el canal (*file descriptor*) *fildes*. Este número nunca será superior al valor del parámetro *nbytes*. De lo contrario devolverá el valor -1 y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *write* fallará, entre otras razones, si:

- *EBADF*: el parámetro *fildes* no es un canal abierto (*file descriptor*) válido para escritura.
- *EDQUOT*: la cuota de bloques de disco del usuario ha sido superada.
- *EFAULT*: el parámetro *buf* apunta a una dirección ilegal.
- *EFBIG*: se intenta escribir en un fichero que supera el tamaño máximo de fichero permitida al proceso, o que supera el tamaño máximo de fichero del sistema.
- *EINTR*: ha llegado un *signal* durante la ejecución de la llamada *write* y no ha habido ninguna escritura.
- *EPIPE*: se intenta escribir en una *pipe* que ya no está abierta para lectura por ningún proceso. Un *signal SIGPIPE* se enviará al proceso al generarse este error.

3.4.6 lseek

Sintaxis

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

off_t lseek(int fildes, off_t offset, int whence);
```

Descripción

La llamada *lseek* sitúa, en una posición determinada, el puntero de lectura/escritura del fichero especificado por el canal (*file descriptor*) abierto *fildes*. La posición final del puntero dependerá del valor de los parámetros *offset* y *whence* de la siguiente manera:

- Si *whence* vale *SEEK_SET*, el puntero se sitúa a la posición del valor de *offset*.
- Si *whence* vale *SEEK_CUR*, el puntero se sitúa a la posición resultante de sumar el valor *offset* con la posición actual del puntero.
- Si *whence* vale *SEEK_END*, el puntero se sitúa a la posición resultante de sumar el valor *offset* al final del fichero.

La llamada *lseek* permite situar el puntero de lectura/escritura más allá del final del fichero. Si se escribe en esta posición, las lecturas que se hagan en las posiciones donde no hay datos devolverán el valor cero hasta que se escriba alguna cosa diferente.

Parámetros

- *fildes* indica un canal (*file descriptor*) abierto de un fichero.
- *offset* indica el desplazamiento relativo que se quiere aplicar al puntero de lectura/escritura.
- *whence* indica la posición a partir de la cual se añadirá el desplazamiento.

Valor devuelto

Si la llamada acaba correctamente se devolverá el *offset* resultante, medido en bytes, del puntero de lectura/escritura respecto del origen del fichero. De lo contrario se devolverá el valor negativo -1, el *offset* no se modificará y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *lseek* fallará, entre otras razones, si:

- *EBADF*: el parámetro *fildes* no es un canal (*file descriptor*) abierto.
- *ENVAL*: el parámetro *whence* no es *SEEK_SET*, *SEEK_CUR* o *SEEK_END*.
- *ESPIPE*: el parámetro *fildes* está asociado a una *pipe*.

3.4.7 dup

Sintaxis

```
#include <unistd.h>

int dup(int fildes);
```

Descripción

La llamada *dup* devuelve un nuevo canal (*file descriptor*) que tiene las siguientes características comunes con el canal indicado en el parámetro *fildes*:

- Mismo fichero abierto o pipe.
- Mismo puntero de lectura/escritura, es decir, los dos canales compartirán un solo puntero.
- Misma modo de acceso (lectura, escritura o lectura/escritura).

Parámetros

fildes indica el canal (*file descriptor*) que se quiere duplicar.

Valor devuelto

Si la llamada acaba correctamente se devolverá un valor entero no negativo que corresponderá al primer canal (*file descriptor*) libre disponible del proceso. De lo contrario se devolverá el valor negativo -1 y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *dup* fallará, entre otras razones, si:

- *EBADF*: el parámetro *fildes* no es un canal (*file descriptor*) válido.
- *EINTR*: ha llegado un *signal* durante la ejecución de la llamada *dup*.
- *EMFILE*: el proceso tiene demasiados ficheros abiertos.

3.4.8 unlink

Sintaxis

```
#include <unistd.h>

int unlink(const char *path);
```

Descripción

La llamada *unlink* elimina un nombre (*link)* de un fichero y reduce el contador de nombres (*links*) del fichero referenciado. Si el parámetro *path* representa un enlace simbólico (*symbolic link*), la llamada borra el enlace simbólico pero no afecta al fichero o al directorio apuntado por el enlace.

Si el contador de enlaces de un fichero llega a cero y ningún proceso tiene el fichero abierto, entonces el espacio ocupado por el fichero se libera y el fichero ya no podrá ser accesible nuevamente. Si algún proceso tiene el fichero abierto cuándo se borra el último nombre (*link*), entonces los contenidos del fichero no se eliminarán hasta que todos los procesos hayan cerrado el fichero.

Parámetros

path apunta al nombre completo de la entrada (*link*) que se quiere eliminar.

Valor devuelto

Si la llamada acaba correctamente devolverá el valor 0. De lo contrario devolverá el valor -1 y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *unlink* fallará, entre otras razones, si:

- *EACCES*: no está permitida la búsqueda en alguno de los componentes directorio del *path*, no está permitida la escritura en el directorio que contiene el nombre (*link*) que se quiere borrar, o el usuario no es el propietario del directorio que contiene el nombre (*link*) ni es el propietario del fichero.
- *EFAULT*: el parámetro *path* apunta a una dirección ilegal.
- *EINTR*: ha llegado un *signal* durante la ejecución de la llamada *unlink*.
- *ENOENT*: no existe el nombre del fichero (*link*) o el parámetro *path* está vacío.
- *ENOTDIR*: algún componente del prefijo del parámetro *path* no es un directorio.
- *EROFS*: la entrada que se quiere borrar forma parte de un sistema de ficheros de sólo lectura.

3.5 Ejemplos de las llamadas de entrada/salida

3.5.1 Ejemplo 1

Este ejemplo copia, carácter a carácter, un fichero fuente en otro fichero destino. Los nombres de los ficheros se pasan como parámetros del programa. El programa abre el fichero fuente (*open*), crea el fichero destino (*open*) y va leyendo (*read*) del fichero fuente carácter a carácter y lo va escribiendo (*write*) en el fichero destino.

```
#include <fcntl.h> /* definiciones O_RDONLY, O_WRONLY ...         */
#include <errno.h>

void error(char *m)
{

      /* Escribe los errores por el canal estándar de errores (2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}


main(int argc, char *argv[])
{
      int source, dest, n;
      char c;

      if (argc != 3)
            error("Número de argumentos erróneo");

      if ((source = open(argv[1], O_RDONLY)) < 0)
            error("Apertura del fichero fuente");

      /* Si el fichero destino existe, lo sobreescribe (O_TRUNC)  */
      /* Si el fichero destino no existe, lo crea (O_CREAT, 0600) */

      if ((dest = open(argv[2], O_WRONLY|O_TRUNC|O_CREAT, 0600)) < 0)
            error("Creación del fichero destino");

      /* Lee del fichero fuente mientras haya información   */
      /* Escribe en el fichero destino todo el que ha leído */

      while ((n=read(source, &c, 1)) > 0)
            if (write(dest, &c, 1) < 0)
                  error("Escritura en el fichero destino");

      if (n<0)
            error("Lectura del fichero fuente");

      close (source);
      close (dest);
}
```

3.5.2 Ejemplo 2

Este ejemplo muestra la creación de un fichero temporal que no aparece en el sistema de ficheros mientras es utilizado. Para crearlo se utiliza la llamada a sistema *creat* e inmediatamente después se borra el nombre del fichero (*unlink*). A partir de este momento, el fichero sigue existiendo pero sin acceso por parte de ningún otro proceso. En el momento que el proceso cierra el canal abierto asociado al fichero (*close*), el fichero desaparece físicamente del sistema de ficheros.

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

void error(char *m)
{
      /* Escribe los errores por el canal estándar de errores (2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}


main()
{
      int fildes;

      /* Crea el fichero temp.dat con permisos para el propietario */

      if ((fildes = creat("temp.dat", 0700)) < 0)
            error("Error en la creación del fichero");

      /* Se borra el nombre (link) del fichero creado */

      if (unlink("temp.dat") < 0)
            error("Error borrando el nombre (link) del fichero");

      /* A partir de este momento, el fichero sólo es       */
      /* accesible mediante el canal fildes de este proceso */



      /* ... */



      /* Una vez cerrado el canal fildes, el fichero desaparece */

      close(fildes);
      exit(0);
}
```

3.5.3 Ejemplo 3

Este ejemplo escribe el contenido de un fichero en sentido inverso, es decir el primer carácter de un fichero (*source*) pasa a ser el último carácter de otro fichero (*dest*). Abre el fichero origen (*source*) pasado como primer parámetro (*open*) y crea el fichero destino (*dest*) pasado como segundo parámetro (*open*). Lee el fichero origen (*read*) en orden inverso (*lseek*) y escribe los datos leídos (*write*) en el fichero destino.

```
#include <fcntl.h>  /* definiciones O_RDONLY, O_WRONLY ...          */
#include <unistd.h> /* definiciones de SEEK_SET, SEEK_CUR, SEEK_END */
#include <errno.h>

void error(char *m)
{

      /* Escribe los errores por el canal estándar de errores (2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}

main(int argc, char*argv[])
{
      int source, dest;
      long index;
      char c;

      if (argc != 3)
            error("Argumentos insuficientes");
      if ((source = open(argv[1], O_RDONLY)) < 0)
            error("Abrir el fichero origen");
      if ((dest = open(argv[2], O_WRONLY|O_TRUNC|O_CREAT, 0600)) < 0)
            error("Crear el fichero destino");

      /* Se lee el fichero origen desde el final hacia el inicio  */

      /* Se calcula el tamaño del fichero con el valor de retorno */
      /* de lseek que posiciona el puntero al final del fichero   */

      if ((index = lseek(source, 0, SEEK_END)) < 0)
            error("Cálculo tamaño fichero con lseek");
      index--;
      while (index >= 0) {

            if (lseek(source, index, SEEK_SET) < 0)
                  error("Posición con lseek");
            if (read(source, &c, 1) < 0)
                  error("Lectura del fichero origen");
            if (write(dest, &c, 1) < 0)
                  error("Escritura del fichero destino");
            index--;

      }

      close(source);
      close(dest);
}
```

3.5.4 Ejemplo 4

Este ejemplo muestra la compartición del puntero de lectura/escritura entre un proceso padre y su proceso hijo que ha heredado la tabla de canales (*file descriptors*) del padre. El proceso padre abre el fichero (*open*) antes de crear al proceso hijo, crea (*fork*) al proceso hijo que hereda la tabla de canales y, por lo tanto, comparte el puntero de lectura/escritura asociado al canal del fichero. Los dos procesos van leyendo (*read*) del fichero y escribiendo (*write*) por la salida estándar hasta llegar al final de fichero. La salida mostrará el fichero una sola vez gracias a la compartición del puntero de lectura/escritura.

```
#include <fcntl.h> /* definiciones O_RDONLY, O_WRONLY ...         */
#include <errno.h>

void error(char *m)
{

      /* Escribe los errores por el canal estándar de errores (2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}

main(int argc, char*argv[])
{
      int source, n, st;
      char c;

      if (argc != 2) error("Faltan argumentos");
      if ((source = open(argv[1], O_RDONLY)) < 0)
            error("Abrir el fichero origen");

      switch (fork())
      {
            case -1: /* en Caso de error el proceso acaba */
                  error("Fork");

            case 0: /* Proceso hijo - lee del fichero y escribe */
                  while ((n=read(source, &c, 1)) > 0)
                        if (write(1, &c, 1) < 0)
                              error("Escritura salida estándar");
                  if (n<0)
                        error("Lectura del fichero fuente");
                  close(source);
                  exit(0);

            default: /* Proceso padre - lee del fichero y escribe */
                  while ((n=read(source, &c, 1)) > 0)
                        if (write(1, &c, 1) < 0)
                              error("Escritura salida estándar");
                  if (n<0)
                        error("Lectura del fichero fuente");
                  close(source);
                  wait(&st);
                  exit(0);
      }
}
```

3.5.5 Ejemplo 5

Si cogemos el mismo código que en el ejemplo 5 pero movemos la apertura (*open*) del fichero y la ponemos después de la creación (*fork*) del proceso hijo, entonces tanto el proceso padre como el proceso hijo escribirán el fichero completo y, por lo tanto, el contenido aparecerá dos veces. En este programa el hijo no ha heredado el canal abierto del fichero y, por lo tanto, cada proceso crea su propio canal con su propia entrada en la tabla de ficheros y con punteros de lectura/escritura independientes para cada proceso.

```
#include <fcntl.h> /* definiciones O_RDONLY, O_WRONLY ...         */
#include <errno.h>

void error(char *m)
{

      /* Escribe los errores por el canal estándar de errores (2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}

main(int argc, char*argv[])
{
      int source, n, st;
      char c;

      if (argc != 2) error("Faltan argumentos");
      switch (fork())
      {
            case -1: /* En caso de error el proceso acaba */
                  error("Fork");

            case 0: /* Proceso hijo - abre, lee y escribe */
                  if ((source = open(argv[1], O_RDONLY)) < 0)
                        error("Abrir el fichero origen");
                  while ((n=read(source, &c, 1)) > 0)
                        if (write(1, &c, 1) < 0)
                              error("Escritura salida estándar");
                  if (n<0)
                        error("Lectura del fichero fuente");
                  close(source);
                  exit(0);

            default: /* Proceso padre - abre, lee y escribe */
                  if ((source = open(argv[1], O_RDONLY)) < 0)
                        error("Abrir el fichero origen");
                  while ((n=read(source, &c, 1)) > 0)
                        if (write(1, &c, 1) < 0)
                              error("Escritura salida estándar");
                  if (n<0)
                        error("Lectura del fichero fuente");
                  close(source);
                  wait(&st);
                  exit(0);
      }
}
```

4 La comunicación y sincronización entre procesos

Todos los sistemas operativos que permiten la ejecución concurrente de procesos es indispensable que ofrezcan herramientas de comunicación y sincronización. Unix ofrece diversos mecanismos de comunicación y sincronización y en este capítulo presentaremos dos: los *signals* y las *pipes*. Los *signals* son señales que permiten notificar un acontecimiento concreto, pero que no incluyen contenido aparte de la propia señal. Las *pipes*, en cambio, permiten enviar mensajes con cualquier tipo de contenido entre procesos que mantengan una relación de parentesco (padre/hijo).

4.1 Los signals

Los *signals* son señales asíncronas que pueden ser enviadas a los procesos por las siguientes causas:

- Un error en la ejecución del proceso: por ejemplo ejecutar una instrucción ilegal *(signal SIGILL*), intentar acceder a una dirección de memoria inválida (*signal SIGSEGV*) *...*
- Un aviso del sistema operativo sobre algún recurso relacionado con el proceso: indicación de un cambio en el estado de algún hijo (*signal SIGCHLD*), aviso de escribir en una *pipe* que no tiene lectores (*signal SIGPIPE*) ...
- Un aviso de otro proceso o del mismo proceso por algún acontecimiento que el proceso considera adecuado: señal de la alarma del propio proceso (*signal SIGALRM*), señal de algún acontecimiento definido por el propio proceso o conjunto de procesos (*signal SIGUSR1*) ... Este envío explícito de *signals* entre procesos se hace mediante la llamada a sistema *kill*.

Cada proceso puede programar su respuesta a cada uno de los *signals*. La acción a tomar al recibir un *signal* determinado dependerá de esta programación y se pueden programar las siguientes acciones:

- Acción por defecto (*SIG_DFL*): en la mayoría de los *signals* la acción por defecto será la muerte del proceso que recibe el *signal*. Esta acción es la que se toma si el proceso no ha heredado ninguna otra programación, ni ha programado el *signal*.
- Ignorar el *signal* (*SIG_IGN*): en este caso el *signal* no llegará al proceso y, por lo tanto, no se tomará ninguna acción.
- Programar una rutina de servicio del *signal* (*signal handler*): en el momento de recibir el *signal*, la ejecución del proceso se interrumpirá y se ejecutará la rutina de servicio programada. Una vez acabada la ejecución de la rutina de servicio se volverá a la ejecución del proceso en el punto donde se había interrumpido. Esta manera de actuar se asimila a la gestión de interrupciones. Algunas versiones de Unix reprograman los *signals* en la acción por defecto una vez se sirve un *signal*. Por esta razón, es conveniente reprogramar nuevamente el *signal* dentro de la rutina de servicio.

Algunos *signals* no son reprogramables y, al recibirlos, siempre se toma la acción por defecto: por ejemplo el *signal SIGKILL* (siempre mata el proceso) y el *signal SIGSTOP* (siempre detiene el proceso). Una excepción es el *signal SIGCHLD* que por defecto se ignora.

La programación de los *signals* se hereda (después de un *fork*) de padres a hijos. En caso de que un proceso haga un cambio de imagen (llamada a sistema *exec*) sólo se

mantendrán los *signals* programados para ser ignorados (*SIG_IGN)* o para tomar la acción por defecto (*SIG_DFL*). En cambio los *signals* que se han programado con una rutina de servicio (*signal handler*) pasarán a la programación por defecto, ya que con la nueva imagen desaparecerá la rutina de servicio programada. La programación de los *signals* se realiza con la llamada a sistema del mismo nombre (*signal*).

A continuación se muestra la ejecución de un proceso que recibe un *signal* programado (con la llamada a sistema *signal*) con una rutina de servicio. En el momento que el *signal* llega, se interrumpe la ejecución del código del proceso y se ejecuta la rutina de servicio. Una vez la rutina de servicio acaba, se vuelve al punto de ejecución donde se había interrumpido previamente:

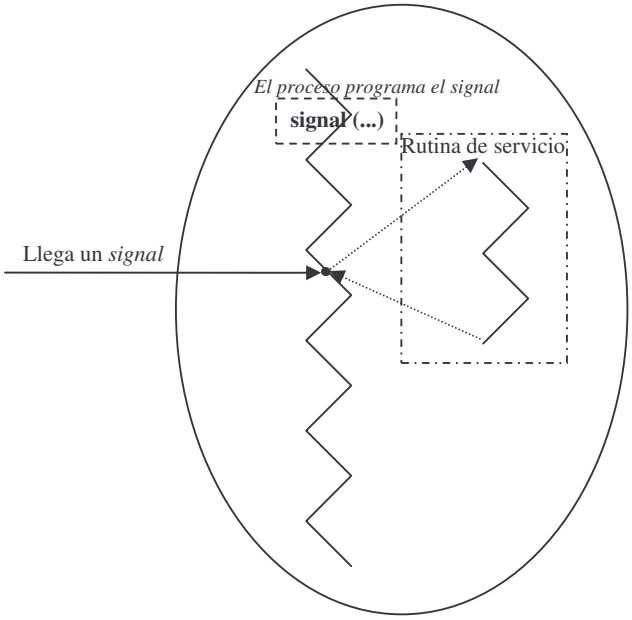


Tabla de *signals* según el estándar *POSIX*:

Señal	**Significado de la señal**	**Número**
SIGABRT	Fin anormal de un proceso	6
SIGALRM	Señal producida por la alarma del proceso	14
SIGFPE	Excepción aritmética	8
SIGHUP	Corte de comunicación con la línea (terminal, módem ...)	1
SIGILL	Instrucción ilegal	4
SIGINT	Interrupción producida desde el terminal (combinación de teclas)	2
SIGQUIT	Fin producido desde el terminal (combinación de teclas)	3
SIGKILL	Matar el proceso (no se puede programar ni ignorar)	9
SIGPIPE	Se ha escrito en una *pipe* sin lectores	13
SIGSEGV	Acceso a memoria inválido	11
SIGTERM	Señal de fin de un proceso	15
SIGUSR1	*Signal 1* definido por el usuario	10
SIGUSR2	*Signal 2* definido por el usuario	12
SIGCHLD	El estado de un hijo ha cambiado	17
SIGCONT	Señal de continuación si el proceso está parado (*stop*)	18
SIGSTOP	Señal de *stop* (no es puede programar ni ignorar)	19
SIGTSTP	Señal de *stop* producido desde el terminal (combinación de teclas)	20
SIGTTIN	Proceso en segundo plano que intenta leer del terminal	21
SIGTTOU	Proceso en segundo plano que intenta escribir en el terminal	22

Todos los *signals* del 1 al 15 provocan, por defecto, la muerte del proceso. Hace falta destacar que el *signal SIGKILL* no se puede programar ni ignorar y se utiliza para matar definitivamente a los procesos.

Los *signals 17* a 22 se ofrecerán sólo en el caso que el control de trabajos (*job control*) esté definido en la versión del sistema. Por defecto *SIGCHLD* se ignora, en cambio *SICONT* provoca la continuación de un proceso parado y el resto (*SIGSTOP*, *SIGTSTP*, *SIGTTIN*, *SIGTTOU*) provocan el paro del proceso que los recibe.

4.2 Las pipes

Las *pipes* son dispositivos lógicos que permiten la comunicación entre procesos con una relación de parentesco entre ellos, es decir, entre procesos que compartan este dispositivo por herencia. Las *pipes* son canales unidireccionales de comunicación y, al ser creadas (llamada a sistema *pipe*), devuelven dos *file descriptors* dentro de la tabla de canales del proceso, uno para lectura y otro para escritura:

```
int p[2];

...

/* Se crea una pipe y se reciben los dos canales: */
/* el de lectura y el de escritura (p[0] y p[1])  */

if (pipe(p) < 0) error("Creación pipe");

...
```

En el código anterior el proceso crea un *pipe* que es accesible a través de los canales (*file descriptors*) *p[0]* y *p[1*]. El canal *p[0]* es de lectura y el canal *p[1]* es de escritura.

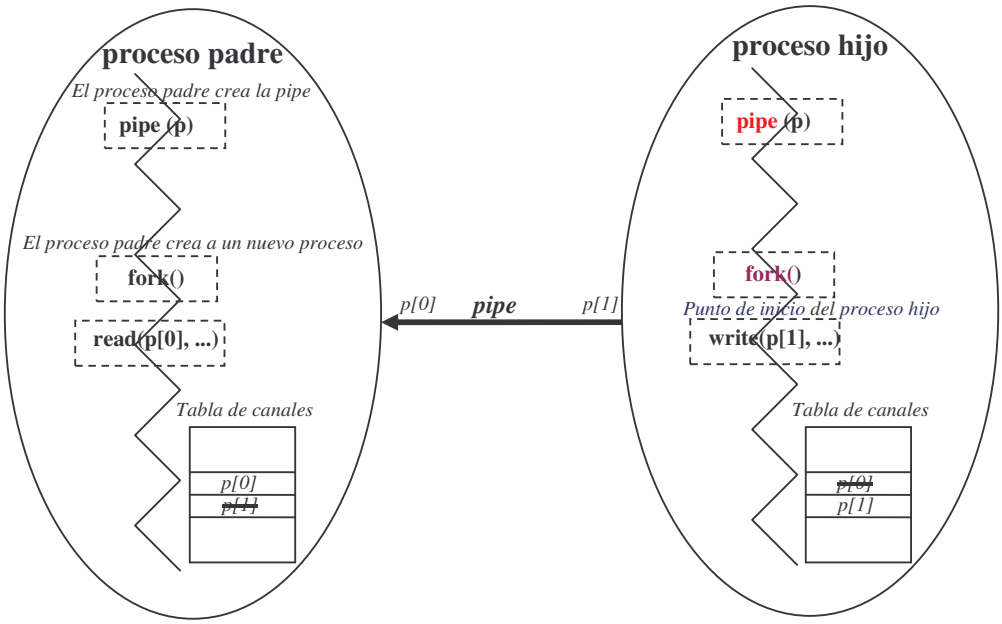
Para poder utilizar la *pipe* para comunicarse con otro proceso, hace falta que los dos procesos la compartan en sus respectivas tablas de canales. Por esta razón, es necesario que el proceso que ha creado la *pipe* cree un nuevo proceso que herede los canales de la *pipe*, con el fin que pueda comunicarse con el proceso creador.

En la próxima figura se muestran dos procesos (padre e hijo) que se comunican mediante una *pipe*. El proceso padre crea la *pipe* (llamada a sistema *pipe*) y acto seguido crea el proceso hijo (*fork*).

Después de la creación de la *pipe*, en la tabla de canales del proceso padre aparecen dos nuevas entradas*, p[0]* y *p[1*], que corresponden a los canales de lectura y de escritura de la *pipe*. De momento, sin embargo, la *pipe* sólo la puede utilizar el proceso que la ha creado, ya que la *pipe* sólo existe en su tabla de canales. Después de la creación del nuevo proceso, el proceso hijo dispone de una tabla de canales idéntica a la del padre y, por lo tanto, comparte los dos canales de acceso a la *pipe*. A partir de este momento, padre e hijo podrán comunicarse mediante la misma *pipe*. En el ejemplo presentado el hijo escribe un mensaje en la *pipe* mediante su canal de escritura en la *pipe p[1*]. Este mensaje es leído por el padre mediante su canal de lectura de la *pipe p[0*].

Con el fin de poder aprovechar todas las características de las *pipes*, como se explica en el texto que sigue, los procesos tendrían que cerrar los canales que no utilicen antes de iniciar la comunicación. En el ejemplo presentado, el proceso padre sólo utilizará el canal de lectura de la *pipe* (*p[0*]) y, por lo tanto, cerrará su canal de escritura (*p[1*]). Por su parte, el proceso hijo

sólo utilizará el canal de escritura de la *pipe* (*p[1]*) y, por lo tanto, cerrará su canal de lectura (*p[0]*).



El código presentado a continuación muestra las acciones que harán padre e hijo para poder comunicarse mediante una *pipe* según el esquema anterior.

```
#include <errno.h>

void error(char *m)
{
      /* Escribe los errores por el canal estándar de errores (2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}

main()
{
      int p[2];
      int st;
      char buff[20];

      /* Se crea una pipe y se reciben los dos canales: */
      /* el de lectura y el de escritura (p[0] y p[1])  */

      if (pipe(p) < 0) error("Creación pipe");

      switch (fork())
      {
            case -1: error("Fork 1");

            case 0: /* Proceso hijo */
```

```
                /* Cierra el canal de la pipe que no utiliza */

                close(p[0]);

                /* Escribe un mensaje a la pipe, la cierra y acaba */

                write(p[1],"Hola\n", 5);
                close(p[1]);
                exit(0);

          default: /* Proceso padre */
                break;
    }

    /* El padre cierra el canal de la pipe que no utiliza */

    close(p[1]);

    /* Lee el mensaje de la pipe y lo escribe por stdout */

    read(p[0], buff, 5);
    write(1, buff, 5);

    /* Espera que acabe el hijo y finaliza su ejecución */

    wait(&st);
    exit(0);
}
```

La lectura de una *pipe* provoca el bloqueo del proceso que intenta leer hasta que la *pipe* disponga del total de bytes que se quieren leer. Asimismo, la escritura de una *pipe* escribe el mensaje y retorna inmediatamente a menos que la *pipe* esté llena. En este caso, el proceso que intenta escribir quedará bloqueado hasta que haya espacio suficiente en la *pipe* para escribir todo el mensaje. Éste es el comportamiento habitual de las *pipes* en lectura y escritura, siempre y cuando haya algún proceso que tenga abierto algún canal de lectura y algún canal de escritura sobre la *pipe*. De lo contrario, el comportamiento es bastante diferente:

- Si no hay ningún proceso que tenga un canal de escritura abierto de una *pipe*, es decir, si ya no es posible que nadie escriba nunca más nada en la *pipe*, entonces la lectura de la *pipe* nunca bloqueará el proceso y retornará inmediatamente con el número de bytes leídos disponibles en la *pipe*. Y si la *pipe* está vacía la lectura devolverá 0 bytes leídos.

- Si no hay ningún proceso que tenga un canal de lectura abierto de una *pipe*, es decir, si ya no es posible que nadie lea nunca más nada de la *pipe*, entonces un intento de escritura sobre la *pipe* provocará que el proceso reciba un *signal SIGPIPE* y que no se escriba nada en la *pipe*. El *signal SIGPIPE* avisará, al proceso que intenta escribir, de que ya no es posible que ningún proceso lea de la *pipe*.

Este comportamiento especial de las *pipes* que no disponen de los dos canales (el de escritura y el de lectura) permite un control detallado del final de la comunicación entre los procesos implicados. Por esta razón es muy importante que siempre se cierren todos los canales que no se utilicen de las *pipes*. De lo contrario la finalización de alguno de los procesos, que se comunican con la *pipe*, no sería detectado por el otro proceso y provocaría un bloqueo permanente del proceso, ya sea en el intento de leer de una *pipe* vacía o en el intento de escribir en una *pipe* llena.

4.3 Llamadas a sistema de comunicación y sincronización entre procesos

En este apartado se presentan las siguientes llamadas relacionadas con la comunicación y sincronización entre procesos:

- *pipe*: crea una *pipe* de comunicación y devuelve los canales (*file descriptors*) de lectura y escritura
- *signal*: programa la acción que se tomará al llegar un determinado *signal*
- *kill*: envía un *signal* a un proceso o a un grupo de procesos
- *alarm*: programa el reloj del proceso para que le envíe un *signal SIGALRM* al cabo de un cierto tiempo
- *pause*: bloquea al proceso hasta que llega un *signal*

4.3.1 pipe

Sintaxis

```
#include <unistd.h>

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

Descripción

La llamada *pipe* crea una mecanismo de comunicación entre procesos llamado *pipe* y devuelve dos canales abiertos (*file descriptors*) que corresponderán al canal de lectura sobre la *pipe* (*fildes[0*]) y al canal de escritura sobre la *pipe* (*fildes[1*]).

La lectura de la *pipe* por el canal *fildes[0*] accede a los datos escritos en la *pipe* por el canal *fildes[1]* siguiendo una estructura *FIFO* (*first-in-first-out*).

Parámetros

fildes se una tabla donde la llamada devolverá los dos canales (*file descriptors*) de acceso a la *pipe*.

Valor devuelto

Si la llamada acaba correctamente devolverá el valor 0. De lo contrario devolverá el valor -1 y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *pipe* fallará si:

- *EMFILE*: el número de canales abiertos supera el máximo permitido al proceso.
- *ENFILE*: no hay espacio para una nueva entrada en la tabla de ficheros.

4.3.2 signal

Sintaxis

```
#include <signal.h>

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

Descripción

Esta llamada permite programar la acción que tomará el proceso ante el recibimiento de un *signal*. Los signals se pueden programar de tres formas diferentes:

- *SIG_DFL*: se tomará la acción por defecto definida para el *signal* indicado en el parámetro *sig*.
- *SIG_IGN*: se ignorará la llegada del *signal* indicado en el parámetro *sig*.
- La dirección de la rutina de servicio (*signal handler*): cada vez que llegue un *signal* del tipo definido en el parámetro *sig* se ejecutará la rutina de servicio indicada.

El sistema garantiza que si se envía más de un *signal* del mismo tipo a un proceso, el proceso recibirá al menos uno de estos *signals*. No se garantiza, sin embargo, la recepción de cada uno de los *signals* enviados.

Parámetros

- *sig* indica el *signal* que se quiere programar y no puede valer ni *SIGKILL* ni *SIGSTOP*.
- *disp* indica la programación del *signal* que puede ser *SIG_DFL*, *SIG_IGN* o la dirección de la rutina de servicio del *signal* (*signal handler*).

Valor devuelto

Si la llamada *signal* acaba correctamente, devolverá la programación previa del *signal* correspondiente. De lo contrario devolverá *SIG_ERR* y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *signal* fallará si:

- *EINTR*: ha llegado un *signal* durante la ejecución de la llamada *signal*.
- *EINVAL*: el valor del parámetro *sig* no es un valor de *signal* válido o es igual a *SIGKILL* o *SIGSTOP*.

4.3.3 kill

Sintaxis

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

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

Descripción

La llamada *kill* envía un *signal* a un proceso o a un grupo de procesos (*process group*). El proceso o el grupo de procesos se especifica en el parámetro *pid* y el *signal* enviado se especifica en el parámetro *sig*. Si el parámetro *sig* vale 0 (*null signal*), entonces la llamada *kill* verifica la validez del identificador indicado en el *pid* pero no se envía ningún *signal*.

El identificador de usuario real o efectivo (*real* o *effective user ID*) del proceso que envía el *signal* tiene que coincidir con el identificador real del proceso que recibe el *signal*, excepto en el caso que el proceso que envía el *signal* pertenezca al *super-user*.

Si el parámetro *pid* es mayor que 0, el *signal* se enviará al proceso que tenga este identificador.

Si el parámetro *pid* es negativo y diferente de -1, el *signal* se enviará a todos los procesos que pertenezcan al grupo con identificador igual al valor absoluto de *pid* y para los cuales se tenga permiso para enviar un *signal*.

Si el parámetro *pid* vale 0, el *signal* se enviará a todos los procesos que pertenezcan al mismo grupo que el proceso que envía el *signal*.

Si el parámetro *pid* vale -1 y el usuario efectivo (*effective user*) del proceso que envía el *signal* no es *super-user*, entonces el *signal* se enviará a todos los procesos que tengan un identificador de usuario real (*real user ID*) igual al identificador de usuario efectivo (*effective u*ser ID) del proceso que envía el *signal*.

Si el parámetro *pid* vale -1 y el usuario efectivo (*effective user*) del proceso que envía el *signal* es *super-user*, entonces el *signal* se enviará a todos los procesos

El sistema garantiza que si es se envía más de un *signal* del mismo tipo a un proceso, el proceso recibirá al menos uno de estos *signals*. No se garantiza, sin embargo, la recepción de cada uno de los *signals* enviados.

Parámetros

- *pid* indica el identificador del proceso o del grupo de procesos (*process group*) a quien se quiere enviar el *signal*.
- *sig* indica el *signal* que se quiere enviar.

Valor devuelto

Si la llamada acaba correctamente devolverá el valor 0. De lo contrario devolverá el valor -1, no se envíará ningún *signal* y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *kill* fallará si:

- *EINVAL*: el valor del parámetro *sig* no es un valor de *signal* válido.
- *EPERM*: si no se tienen permisos para enviar un *signal* a un determinado proceso.
- *ESRCH*: no existe ningún proceso ni ningún grupo de procesos (*process group*) con el identificador indicado en el parámetro *pid*.

4.3.4 alarm

Sintaxis

```
#include <unistd.h>

unsigned int alarm(unsigned int sec);
```

Descripción

La llamada *alarm* programa el reloj del proceso que la invoca con el fin de que envíe al propio proceso un *signal SIGALRM*, después de un determinado número de segundos especificados en el parámetro *sec*.

Las peticiones hechas con la llamada *alarm* por un mismo proceso no se acumulan, sino que una nueva llamada anula el anterior.

Si el parámetro *sec* vale 0, se cancela cualquier petición de alarma hecha previamente.

La llamada *fork* anula en el hijo la programación de la alarma que tenga el proceso padre. En cambio la llamada *exec* mantiene la alarma que se haya programado antes de su invocación.

Parámetros

sec indica el número de segundos a partir de los cuales se enviará un *signal SIGALRM* al proceso que ha invocado la llamada.

Valor devuelto

La llamada *alarm* devuelve el tiempo que todavía falta para que llegue la próximo alarma del proceso.

Errores

No hay ningún error definido.

4.3.5 pause

Sintaxis

```
#include <unistd.h>

int pause(void);
```

Descripción

La llamada *pause* suspende la ejecución del proceso que la ha invocado hasta que el proceso recibe un *signal* que no haya sido programado para ser ignorado.

Si el *signal* provoca la finalización del proceso, entonces la llamada *pause* ya no retorna.

Valor devuelto

La llamada *pause* siempre devuelve error con el valor -1 y la variable *errno* indicará el error que se ha producido.

Errores

La llamada *pause* fallará si:

- *EINTR*: ha llegado un *signal* durante la ejecución de la llamada *pause*.

4.4 Ejemplos de las llamadas de comunicación y sincronización entre procesos

4.4.1 Ejemplo 1

Este ejemplo muestra la programación (*signal*) de una alarma (*alarm*) que se envía al proceso cada 3 segundos. El proceso se bloquea (*pause*) y espera la llegada de la alarma.

```
#include <signal.h>

#define TIEMPO 3


/* Rutina de atención del SIGALRM */

void rutina_alarma(int foo)
{
      char s[20];

      /* Redefine la programación del signal SIGLARM */

      signal(SIGALRM, rutina_alarma);

      /* Escribe un mensaje cada vez que llega la alarma */

      write(1, "Alarma 3 segundos\n", strlen("Alarma 3 segundos \n"));

      /* Solicita un signal SIGLARM de aquí a TIEMPO segundos */

      alarm(TIEMPO);
}


main()
{
      /* Define la programación del signal SIGALRM) */

      signal(SIGALRM, rutina_alarma);

      /* Solicita un SIGLARM de aquí a TIEMPO segundos */

      alarm(TIEMPO);

      /* Bucle infinito a la espera de la alarma*/

      while(1)
      {
            /* Bloquea al proceso hasta que llega el signal */

            pause();
      }
}
```

4.4.2 Ejemplo 2

Este ejemplo muestra la creación (*fork*) de dos procesos hijos que se comunican con una *pipe*. El proceso padre crea la *pipe* (*pipe*) y acto seguido crea (*fork*) al primer proceso hijo. Este primer proceso hijo hereda la *pipe* del padre, redirecciona (*close* y *dup*) su salida estándar hacia la *pipe* y cierra (*close*) los canales que no utiliza de la *pipe*. Acto seguido ejecuta (*exec*) el fichero ejecutable *ls*. El padre crea (*fork*) el segundo proceso hijo. El segundo proceso hijo hereda la *pipe* del padre, redirecciona (*close* y *dup*) su entrada estándar hacia la *pipe*, cierra (*close*) los canales que no utiliza de la *pipe* y ejecuta (*exec*) el fichero ejecutable *grep*. El padre cierra (*close*) los canales de la *pipe* que no utiliza, espera (*wait*) la finalización de los dos hijos y acaba (*exit*). La ejecución de los dos hijos equivale a la orden:

```
ls -l /urs/bin | grep cat
```

```
#include <errno.h>

void error(char *m)
{

      /* Escribe los errores por el canal estándar de errores (2) */

      write(2, m, strlen(m));
      write(2, "\n", 1);
      write(2, strerror(errno), strlen(strerror(errno)));
      exit(1);
}

main()
{
      int p[2];
      int st1, st2;

      /* Se crea una pipe y se reciben los dos canales: */
      /* el de lectura y el de escritura (p[0] y p[1])  */

      if (pipe(p) < 0) error("Creación pipe");
      switch (fork())
      {
            case -1: error("Fork 1");

            case 0: /* Hijo 1 - Redirecciona, cierra y ejecuta */

                  /* Redirecciona salida estándar hacia la pipe */

                  close(1); dup(p[1]);

                  /* Cierra los canales de la pipe que no utiliza */

                  close(p[0]); close(p[1]);

                  /* Carga el código de ls con dos parámetreos */

                  execlp("ls","ls","-l",/usr/bin", (char *)0);
                  error("Ejecución ls ");

            default: /* Proceso padre */
                  break;
      }
```

```
        switch (fork())
        {
              case -1: error("Fork 2");

              case 0: /* Hijo 2 - Redirecciona, cierra y ejecuta */

                    /* Redirecciona entrada estándar hacia la pipe */

                    close(0); dup(p[0]);

                    /* Cierra los canales de la pipe que no utiliza */

                    close(p[0]); close(p[1]);

                    /* Carga el código de grep con un parámetro */

                    execlp("grep", "grep", "cat", (char *)0);
                    error("Ejecución grep");

              default: /* Proceso padre */
                    break;
        }

        /* El padre cierra los canales que no utiliza */

        close(p[0]); close(p[1]);

        /* Espera que acaben los dos hijos */

        wait(&st1);
        wait(&st2);

        /* Finaliza su ejecución */

        exit(0);
}
```

Si alguno de los procesos no cerrara los canales de la *pipe* que no utiliza la finalización de los procesos no sería correcta. Por ejemplo, si el padre no cierra los dos canales de la *pipe* antes de esperar la finalización de los hijos, provocará que el proceso *grep* se quede bloqueado indefinidamente leyendo (*read*) de su entrada estándar una vez se haya acabado el proceso *ls*. Consecuentemente el proceso padre tampoco acabará porque se quedará bloqueado esperando (*wait*) la finalización de su segundo hijo.

5 Bibliografía

- Manual en línea de Unix: orden `man`.
- Kernighan, Brian W. y Pike, Rob. *El entorno de programación UNIX*. Prentice-Hall Hispanoamericana. México, 1987.
- Robbins, Kay A. y Robbins, Steven. UNIX programación práctica. Prentice-Hall Hispanoamericana. México, 1997.
- Stevens W.Richard. *Advanced Programming in the Unix Environment*. Reading, Mass.: Addison-Wesley, 2001.