Introducción a Sistemas Operativos: Comunicación entre Procesos

Francisco J Ballesteros

1. Redirecciones de Entrada/Salida

Ya vimos cómo pedirle al shell que ejecute un comando haciendo que la salida estándar de dicho comando se envíe a un fichero.

```
unix$ echo hola >/tmp/fich
unix$ cat /tmp/fich
hola
unix$
```

Hemos hecho esto con diversos comandos. En todos los casos, el shell ha llamado a `fork` y, antes de llamar a `exec` para ejecutar el nuevo programa, ha hecho que la salida estándar del nuevo proceso termine en el fichero que hemos especificado (`/tmp/fich` en este ejemplo).

Pero podemos hacer lo mismo con la entrada. Por ejemplo, podemos guardar la salida de *ps(1)* en un fichero y después contar las líneas que contiene con *wc(1)*. Esto nos indica cuántos procesos estamos ejecutando:

```
unix$ ps > /tmp/procs
unix$ wc </tmp/procs
      25     144    1497
unix$
```

Aunque podríamos haber utilizado

```
unix$ wc /tmp/procs
              25     144    1497 /tmp/procs
unix$
```

Hay que decir que en este caso estaríamos ejecutando 24 procesos dado que `ps` escribe una línea de cabecera indicando qué contiene cada columna de su salida.

Aunque la salida de `wc` es similar en ambos casos, en el primer caso `wc` no ha recibido argumentos y se limita a leer su entrada estándar y contar líneas, palabras y caracteres. En el segundo caso hemos utilizado `/tmp/procs` como argumento, por lo que `wc` abre dicho fichero y cuenta sus líneas, palabras y caracteres. De ahí que en el segundo caso `wc` muestre el nombre de fichero tras la cuenta.

Los caracteres "`<`" y "`>`" son sintaxis de shell y se utilizan para indicar **redirecciones de entrada/salida**. Su utilidad es hacer que la entrada o la salida estándar de un comando (del proceso que lo ejecuta) se redirija a un fichero. El comando se limita a leer del descriptor `0` y escribir en el `1`, como cabe esperar.

Podemos también redirigir cualquier otro descriptor, por ejemplo la salida de error estándar.

```
unix$ if ls /blah 2>/dev/null >/dev/null
> then
>     echo /blah existe
> else
>     echo /blah no existe
> fi
/blah no existe
unix$
```

En este caso hemos enviado la salida de error estándar ("2") al fichero `/dev/null` y la salida estándar también a `/dev/null`. Dicho fichero es un dispositivo que ignora los writes (pretendiendo que se han hecho correctamente) y que devuelve *eof* cada vez que se lee del mismo. Así pues, hemos utilizado `ls` sólo por su *exit status*, para ver si un fichero existe o no.

Habría sido más apropiado utilizar

```
unix$ if test -e /blah
...
```

para ver si `/blah` existe, o

```
unix$ if test -f /blah
```

para ver si existe y es un fichero regular, o

```
unix$ if test -d /blah
...
```

para ver si existe y es un directorio. Deberías leer *test(1)* para echar un vistazo a todas las condiciones que permite comprobar. La utilidad de este comando es tan sólo llamar a `exit(0)` o `exit(1)` dependiendo de si la condición que se le pide comprobar es cierta o no.

Volviendo a las redirecciones, también podemos indicar que un descriptor se redirija al sitio al que se refiere otro descriptor, como en

```
unix$ echo houston we have a problem 1>&2
houston we have a problem
unix$
```

que escribe su mensaje en la salida estándar (esto lo hace `echo`) pero haciendo que la salida estándar se diriga al sitio (al fichero) al que se refiere la salida de error estándar (esto lo hace el shell con la redirección).

Podemos comprobar que este es el caso enviando además la salida estándar a `/dev/null`. Si el mensaje sigue saliendo, es que se escribe en la salida de error.

```
unix$ ( echo hola 1>&2 )>/tmp/a
hola
unix$ cat /tmp/a
unix$ ( echo hola 2>&1 )>/tmp/a
unix$ cat /tmp/a
hola
unix$
```

Los paréntesis son sintaxis de shell para agrupar comandos y aplicar una redirección si queremos a un conjunto de comandos. Los hemos utilizado para que resulte obvio lo que está pasando.

Por ejemplo, nuestro script para compilar y ejecutar debiera ser

```
#!/bin/sh
if test $# -eq 0
then
    echo usage: $0 fich 1>&2
    exit 1
fi
if cc $1
then
    ./a.out
fi
```

dado que el mensaje de error "`usage`..." debería escribirse en la salida de error estándar.

Veamos cómo hacer una redirección en C.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int fd;
    int sts;

    switch(fork()){
    case -1:
        err(1, "fork failed");
        break;
    case 0:
        fd = open("iredir.c", O_RDONLY);
        if (fd < 0) {
            err(1, "open: iredir.c");
        }
        dup2(fd, 0);
        close(fd);
        execl("/bin/cat", "cat", NULL);
        err(1, "exec failed");
        break;
    default:
        wait(&sts);
    }
    exit(0);
}
```

En este programa, tras llamar a `fork`, el proceso hijo hace algunos ajustes y después llama a `exec` para cargar y ejecutar el programa `cat`. Y sabemos que cuando `cat` ejecuta sin argumentos lee su entrada y la escribe en la salida. Si ejecutamos este programa vemos algo como esto

```
unix$ iredir
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
...
    exit(0);
}
unix$
```

¡El programa escribe el contenido de `iredir.c`!

Como puedes imaginar, `cat` sigue siendo el mismo binario de siempre. No tiene código (ni argumentos) que le indiquen que ha de leer `iredir.c`. Se limita a leer del descriptor `0` y escribir en el `1`. Eso si, antes de llamar a `exec`, nuestro programa ha ejecutado

```
fd = open("iredir.c", O_RDONLY);
if (fd < 0) {
    err(1, "open: iredir.c");
}
dup2(fd, 0);
close(fd);
```

La parte interesante es la llamada a *dup2(2)*. Primero, hemos abierto `iredir.c` para leer, lo que nos ha dado un descriptor de fichero que vamos a suponer que es `3`. A continuación cuando el programa efectúa la llamada

```
dup2(3, 0);
```

hace que UNIX deje como descriptor `0` lo mismo que tiene el descriptor `3`. Recuerda que un descriptor es un índice en la tabla de descriptores del proceso, que apunta a (el record que representa) un fichero abierto. Tras la llamada, el descriptor `0` corresponde a `iredir.c` (para leer y por el momento con offset `0`). Dado que no necesitamos el descriptor `3` para nada más, el programa lo cierra. Puedes ver en la figura 1 cómo están los descriptores antes y después de *duplicar* el descriptor `3` en el `0`.

Cuando ejecute `cat`, su código leerá de `0` que esta vez consigue leer de `iredir.c`. Eso es todo.

Es preciso seguir los convenios que tenemos en UNIX. Por ejemplo, si redirigimos la salida estándar utilizando un descriptor que hemos abierto en modo lectura, las escrituras fallarán. Vamos a cambiar el código de nuestro programa para que ejecute

```
fd = open("iredir.c", O_RDONLY);
if (fd < 0) {
    err(1, "open: iredir.c");
}
dup2(fd, 0);
dup2(fd, 1);
close(fd);
```

En lugar de lo que hacía antes, y podríamos ver esto

```
unix$ iredir2
cat: stdout: Bad file descriptor
unix$
```

¡`cat` no puede escribir en la salida! ¿Puedes ver por qué?

Una redirección de salida hace que el shell llame a `creat` para crear el fichero al que hay que enviar la salida. Una de entrada hace que el shell llame a `open` para leer del fichero. Además, es posible utilizar "`>>`"

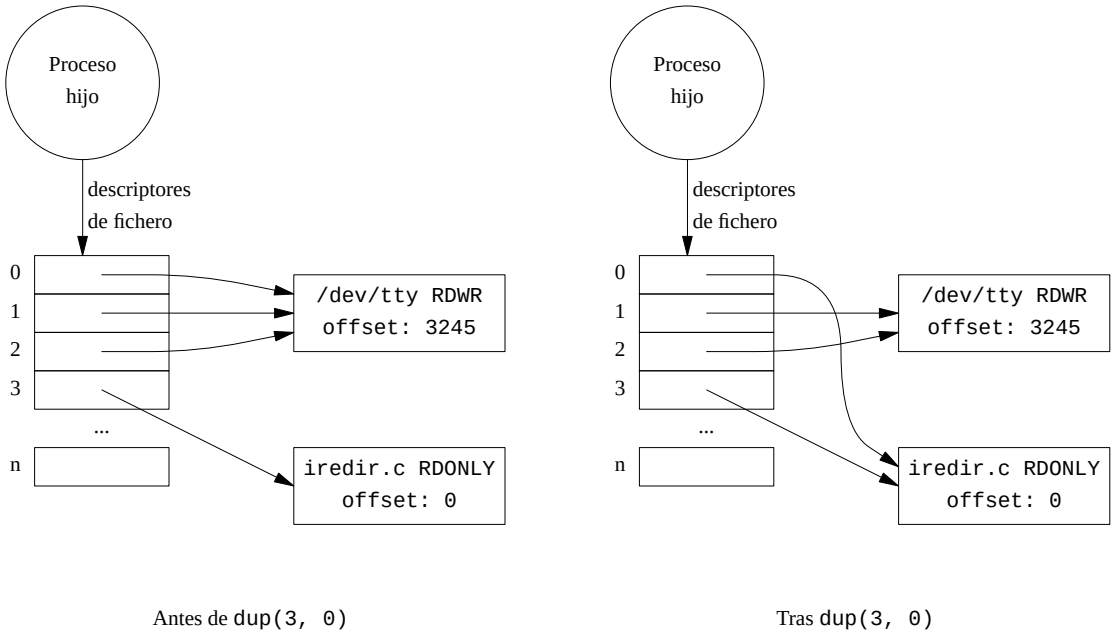

Figura 1: *Procesos antes y después de duplicar el descriptor 3 en el 0.*

para pedirle al shell que envíe la salida a un fichero en modo *append*. En este caso, el shell hará el `open` del fichero en cuestión utilizando el flag `O_APPEND` de `open`, que indica a UNIX que se desea efectuar las escrituras al final del fichero.

Pero es preciso tener cuidado cuando combinamos redirecciones. Ya sabemos lo que hace `creat`. Por ejemplo, supongamos que queremos dejar un fichero de texto con su contenido en mayúsculas. El comando *tr(1)* sabe *traducir* unos caracteres por otros. En particular,

```
tr a-z A-Z
```

cambia los caracteres en el rango "`a-z`" por los del rango "`A-Z`", lo que efectivamente pasa texto a mayúsculas. Por ejemplo,

```
unix$ echo hola >fich
unix$ cat fich
hola
unix$ tr a-z A-Z <fich
HOLA
unix$
```

¡Vamos a utilizar un comando para pasar `fich` a mayúsculas!

```
unix$ tr a-z A-Z <fich >fich
unix$ cat fich
unix$
```

¿Qué ha pasado? !Hemos perdido el contenido de `fich`!

¡Naturalmente! El shell lee la línea y la ejecuta. En este caso sabemos que ejecutará `tr` en un nuevo proceso y que hará dos redirecciones antes de ejecutarlo (tras el `fork`):

- la entrada se redirige a `fich` (abierto para leer)
- la salida se redirige a `fich` (usando `creat`).

En cuanto el shell ha llamado a `creat`, ¡perdemos el contenido de `fich`! Deberíamos haber utilizado en

este caso algo como

```
unix$ tr a-z A-Z <fich >/tmp/tempfich
unix$ mv /tmp/tempfich fich
unix$ cat fich
HOLA
unix$
```

Ahora que conocemos *dup(2)* podemos entender que "`2>&1`" es en realidad un `dup`. ¿Cuál sería? ¿Será

```
dup2(2, 1);
```

o

```
dup2(1, 2);
```

será lo que haga?

Cuando veas una línea de comandos como

```
unix$ cmd >/foo 2>&1
```

piensa en lo que hace cada redirección y en que las del tipo "`2>&1`" son llamadas a `dup2`. Y recuerda que el orden en que hacen dichas redirecciones importa cuando hay llamadas a `dup2` de por medio.

2. Pipelines

Hace tiempo, UNIX disponía de las redirecciones que hemos visto y los usuarios combinaban programas existentes para procesar ficheros. Pero era habitual procesar un fichero con un comando y luego procesar la salida que éste dejaba con otro comando, y así sucesivamente. Por ejemplo, si queremos contar el número de veces que aparece la palabra "failed" en un fichero, sin tener en cuenta si está en mayúsculas o no, podríamos convertir nuestro fichero a minúsculas, quedarnos con las líneas que contienen "failed" y contarlas:

```
unix$ tr A-Z a-z fich >/tmp/out1
unix$ grep failed <tmp/out1 >/tmp/out2
unix$ wc -l </tmp/out2
1
unix$
```

Hemos utilizado el comando *grep(1)* que escribe aquellas líneas que contienen la expresión que hemos indicado como argumento. Más adelante volveremos a usarlo.

Pero a Doug McIlroy se le ocurrió que deberían poderse usar los programas para recolectar datos, como en un jardín, haciendo que los datos pasen de un programa a otro. En ese momento introdujeron en UNIX un nuevo artefacto, el **pipe** o *tubería*, y cambiaron todos los programas para que utilizasen la entrada estándar si no recibían un nombre de fichero como argumento.

El resultado es que podemos escribir

```
unix$ cat fich | tr A-Z a-z | grep failed | wc -l
1
unix$
```

en lugar de toda la secuencia anterior. Cada "`|`" que hemos utilizado es una *tubería* (un pipe) que hace que los bytes que escribe el comando anterior en su salida sean la entrada del comando siguiente. Es como si conectásemos todos estos comandos en una tubería. Lo que vemos en la salida es la salida del último comando (y claro, todo lo que escriban en sus salidas de error estándar).

Por cierto, que si hubiésemos leído *grep(1)*, podríamos haber descubierto el flag `-i` que hace que `grep` ignore la capitalización, consiguiendo el mismo efecto con

```
unix$ grep -i failed fich | wc -l
1
unix$
```

que con el comando anterior. ¡El manual es tu amigo!

La figura 2 muestra cómo los procesos en esta última línea de comandos quedan interconectados por un pipe.

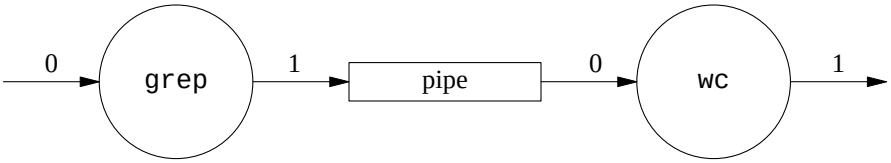

Figura 2: *Utilizando un pipe para enviar la salida de* `grep` *a la entrada de* `wc`.

En la figura hemos representado los descriptores como flechas y utilizado números para indicar de qué descriptor se trata en cada caso.

Debes pensar en el pipe como en un fichero peculiar que tiene dos extremos, uno para leer y otro para escribir. O puedes pensar que los bytes son agua y el pipe es una tubería. Los pipes ni leen ni escriben. Son los procesos los que leen y escriben bytes. Otra cosa es dónde van esos bytes o de dónde proceden.

Para crear un pipe puedes utilizar código como este

```
int fd[2];
if (pipe(fd) < 0) {
    // pipe ha fallado
}
```

que rellena el array `fd` con dos descriptores de fichero. En `fd[0]` tienes el descriptor del que hay que leer para leer de la tubería y en `fd[1]` tienes el que puedes utilizar para escribir en la tubería. Una buena forma de recordarlo es pensar que `0` era la entrada y `1` la salida.

3. Juegos con pipes

Antes de programar algo que consiga el efecto de la línea de comandos que hemos visto, vamos a jugar un poco con los pipes para ver si conseguimos entenderlos correctamente. Aquí tenemos un primer programa que utiliza `pipe`.

```
#include <stdlib.h>
#include <unistd.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    int fd[2], nr;
    char buf[1024];

    if (pipe(fd) < 0) {
        err(1, "pipe failed");
    }
    write(fd[1], "Hello!\n", 7);
    nr = read(fd[0], buf, sizeof(buf));
    write(1, buf, nr);
    exit(0);
}
```

Cuando lo ejecutamos, sucede lo siguiente:

```
unix$ pipe1
Hello!
unix$
```

Tras llamar a `pipe`, el programa escribe 7 bytes en el pipe y luego lee del pipe. Como puedes ver, ha leído lo mismo que ha escrito. Eso quiere decir que lo que escribes en un pipe es lo que se lee del mismo.

Cambiemos ahora el programa para que haga dos writes en el pipe usando

```
if (pipe(fd) < 0) {
    err(1, "pipe failed");
}
write(fd[1], "Hello!\n", 7);
write(fd[1], "Hello!\n", 7);
nr = read(fd[0], buf, sizeof(buf));
write(1, buf, nr);
```

¿Qué escribirá ahora? Si lo ejecutamos podremos verlo:

```
unix$ pipe2
Hello!
Hello!
unix$
```

¡Un sólo `read` ha leído lo que escribimos con los dos `write`s! Dicho de otro modo, los pipes de UNIX (en general) no delimitan mensajes. O, no preservan los límites de los writes. Sucede igual que en conexiones de red. Una vez los bytes están en el pipe da igual si se escribieron en un único `write` o en varios. Cuando un `read` lea del pipe, leerá lo que pueda.

Vamos a intentar escribir todo lo que podamos dentro de un pipe en este otro programa:

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

int
main(int argc, char* argv[])
{
    int fd[2], nw;
    char buf[1024];

    if (pipe(fd) < 0)
        err(1, "fork failed");
    for(;;){
        nw = write(fd[1], buf, sizeof buf);
        fprintf(stderr, "wrote %d bytes\n", nw);
    }
    exit(0);
}
```

Cuando lo ejecutamos

```
unix$ fillpipe
wrote 1024 bytes
wrote 1024 bytes
...
wrote 1024 bytes
```

vemos 64 mensajes impresos y el programa no termina. El programa está dentro de una llamada a `write`, intentando escribir más en el pipe, ¡pero no puede!

Los pipes tienen algo de buffer (son sólo un buffer en el kernel que tiene asociados dos descriptores). Cuando escribimos en un pipe los bytes se copian al buffer del pipe. Cuando leemos de un pipe los bytes proceden de dicho buffer. Pero si llenamos el pipe, UNIX detiene al proceso que intenta escribir hasta que se lea algo del pipe y vuelva a existir espacio libre en el buffer del pipe. Como puedes ver, en nuestro sistema UNIX resulta que los pipes pueden almacenar 64KiB, pero no más.

Y aún nos falta por ver un último efecto curioso que puede producirse si escribimos en un pipe. Observa el siguiente programa.

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

int
main(int argc, char* argv[])
{
    int fd[2];

    if (pipe(fd) < 0) {
        err(1, "pipe failed");
    }
    close(fd[0]);
    fprintf(stderr, "before\n");
    write(fd[1], "Hello!\n", 7);
    fprintf(stderr, "after\n");
    exit(0);
}
```

Si esta vez lo ejecutamos...

```
unix$ closedpipe
before
15131: signal: sys: write on closed pipe
unix$
```

UNIX mata el proceso en cuanto intenta escribir. Veremos cómo cambiar este comportamiento, pero es el comportamiento normal en UNIX cuando escribimos en un pipe del que nadie puede leer.

Piensa en una línea de comandos en que utilizas un pipeline y el último comando termina pronto. Por ejemplo, escribiendo los dos primeros strings que contiene el disco duro y que son imprimibles:

```
unix# cat /dev/rdisk0s1 | strings | sed 2q
BSD  4.4
gEFI        FAT32
unix#
```

¿Querrías que `cat` continuase leyendo *todo* el disco una vez has encontrado lo que buscas? (El comando *strings(1)* escribe en la salida los bytes de la entrada que corresponden a strings imprimibles, ignorando el resto de lo que lee).

Una vez `sed` imprime las dos primeras líneas que lee, termina. Esto tiene como efecto que el segundo pipe deja de tener descriptores abiertos para leer del mismo. El efecto es que cuando `strings` intenta escribir tras la muerte de `sed`, UNIX mata a `strings`. A su vez, esto hace que el primer pipe deje de tener abiertos descriptores para leer del mismo. En ese momento, si `cat` intenta escribir, UNIX lo mata y termina la ejecución de nuestra línea de comandos.

Nos falta por ver qué sucede si leemos repetidamente de un pipe. Podemos modificar uno de los programas anteriores para verlo de forma controlada:

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

int
main(int argc, char* argv[])
{
    int fd[2], nr;
    char buf[5];

    if (pipe(fd) < 0) {
        err(1, "pipe failed");
    }
    write(fd[1], "Hello!\n", 7);
    close(fd[1]);
    do {
        nr = read(fd[0], buf, sizeof(buf)-1);
        if (nr < 0) {
            err(1, "pipe read failed");
        }
        buf[nr] = 0;
        printf("got %d bytes '%s'\n", nr, buf);
    } while(nr > 0);
    exit(0);
}
```

¡Vamos a ejecutarlo!

```
unix$ piperd
got 4 bytes 'Hell'
got 3 bytes 'o!
'
got 0 bytes ''
unix$
```

El primer `read` obtiene 4 bytes (que es cuanto le dejamos leer por el tamaño del buffer). Observa que terminamos los bytes que leemos con un byte a cero para que C lo pueda entender como un string.

El segundo `read` consigue leer los 3 bytes restantes que habíamos escrito. Pero el tercer `read` recibe una indicación de *EOF* (0 bytes leídos). Esto es natural si pensamos que nadie puede escribir en el pipe (hemos cerrado el descriptor para escribir en el pipe y nadie más lo tiene) y que hemos vaciado ya el buffer del pipe.

Así pues, cuando ningún proceso tiene abierto un descriptor para poder escribir en un pipe y su buffer está vacío, `read` siempre devuelve una indicación de EOF. Es importante por esto que cierres todos los descriptores en cuanto dejen de ser útiles. En este ejemplo ves que si hubiésemos dejado abierto el descriptor de `fd[1]` el programa nunca terminaría.

4. Pipeto

Vamos a utilizar ahora los pipes para hacer un par de funciones útiles. La primera nos dejará (en un programa en C) ejecutar un comando externo de tal forma que podamos escribir cosas en su entrada estándar. Hay mucho usos para esta función. Uno de ellos es enviar correo electrónico.

El comando *mail(1)* es capaz de leer un mensaje de correo (texto) de su entrada y enviarlo. Podemos

utilizar el flag `-s` para indicar un *subject* y suministrar como argumento la dirección de *email* a que queremos enviar el mensaje. Por ejemplo, si tenemos las notas de una asignatura en un fichero llamado "NOTAS" y en cada línea tenemos la dirección de email y las notas de un alumno, podríamos ejecutar

```
unix$ EMAIL=geek@geekland.com
unix$ grep $EMAIL NOTAS | mail -s 'tus notas' $EMAIL
unix$
```

para enviar las notas al alumno con su email en `$EMAIL`.

Estaría bien poder hacer lo mismo desde C y poder programar algo como

```
fd = pipeto("mail -s 'tus notas' geek@geekland.com");
if (fd < 0) {
    // pipeto failed
    return -1;
}
nw = write(fd, mailtext, strlen(mailtext));
...
close(fd);
```

para enviar el mensaje desde un programa en C. En este caso queremos que la función `pipeto` nos devuelva un descriptor que podamos utilizar para escribir algo que llegue a la entrada estándar del comando que ejecuta `pipeto`.

Esta es la función:

```
int
pipeto(char* cmd)
{
    int fd[2];

    pipe(fd);
    switch(fork()){
    case -1:
        return -1;
    case 0:
        close(fd[1]);
        dup2(fd[0], 0);
        close(fd[0]);
        execl("/bin/sh", "sh", "-c", cmd, NULL);
        err(1, "execl");
    default:
        close(fd[0]);
        return fd[1];
    }
}
```

Como puedes ver, llamamos a `pipe` antes de hacer el `fork`. Esto hace que tras el `fork` tanto el padre como el hijo tengan los descriptores para leer y escribir en el pipe. El padre cierra el descriptor por el que se lee del pipe (no lee nunca del pipe) y retorna el descriptor que se usa para escribir. En cambio, el hijo cierra el descriptor por el que se escribe en el pipe y a continuación ejecuta

```
dup2(fd[0], 0);
close(fd[0]);
execl("/bin/sh", "sh", "-c", cmd, NULL);
```

para ejecutar `cmd` como un comando en un shell cuya entrada estándar procede del pipe.

El efecto de ejecutar, por ejemplo,

```
fd = pipeto("grep foo");
```

puede verse en la figura 3.

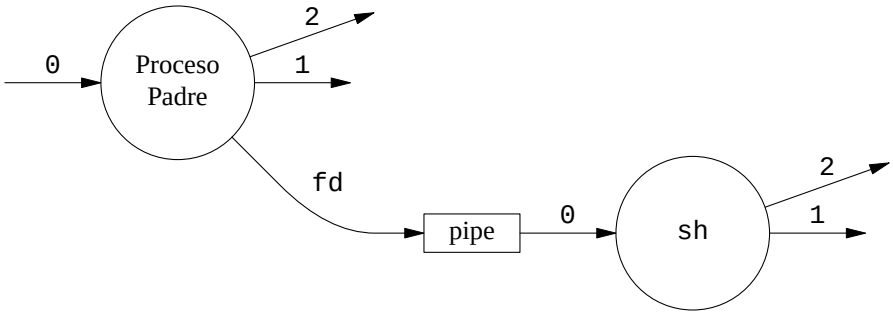

Figura 3: *Descriptores tras la llamada a pipeto mientras ejecutan ambos procesos.*

Para poder ejecutar comandos de shell la función ejecuta un shell al que se le indica como argumento el comando que queremos ejecutar. Si se desea ejecutar un sólo comando o no se requiere poder utilizar sintaxis de shell podríamos ejecutar directamente el programa que deseemos.

Un detalle importante es que si no hubiésemos cerrado en el hijo el descriptor por el que se escribe en el pipe, el comando nunca terminaría si lee la entrada estándar hasta EOF. ¿Puedes ver por qué?

Otro detalle curioso es que redirigimos la entrada del proceso hijo (para que lea del pipe) pero *no* redirigimos la salida del padre para escribir en el pipe. ¿Qué te parece esto?

¡Naturalmente!, el hijo hará un `exec` y el programa que ejecutemos *no sabe* que ha de leer de ningún pipe. Simplemente va a leer de su entrada estándar. Por ello hemos de conseguir que el descriptor `0` en dicho proceso sea el extremo del pipe por el que se lee del mismo. Pero el código del padre es harina de otro costal. El padre *sabe* que tiene que escribir en el descriptor que devuelve `pipeto`. Así pues, ¿por qué habríamos de redirigir nada para escribir en el pipe?

Además, una vez rediriges la salida estándar, has perdido el valor anterior del descriptor y no puedes volver a recuperar la salida estándar anterior. Ni siquieres podrías abriendo `/dev/tty`, dado que quizá tu salida estándar no era `/dev/tty`.

5. Pipefroms

Otra función de utilidad realiza el trabajo inverso, permite leer la salida de un comando externo en un programa escrito en C. Este podría ser el código de ficha función.

```
int
pipefrom(char* cmd)
{
    int fd[2];

    if (pipe(fd) < 0) {
        return -1;
    }
    switch(fork()){
    case -1:
        return -1;
    case 0:
        close(fd[0]);
        dup2(fd[1], 1);
        close(fd[1]);
        execl("/bin/sh", "sh", "-c", cmd, NULL);
        err(1, "execl");
    default:
        close(fd[1]);
        return fd[0];
    }
}
```

En este caso redirigimos la salida estándar del nuevo proceso al pipe (en el proceso hijo) y retornamos el descriptor para leer del pipe.

Podemos utilizar esta función en un programa para leer la salida de un comando. Por ejemplo, este programa ejecuta el comando `who` que muestra qué usuarios están utilizando el sistema y lee su salida. En nuestro caso nos limitamos a escribir toda la salida del comando en nuestra salida estándar. En un caso real podríamos procesar la salida de `who` para hacer con ella algo más interesante.

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

int
pipefrom(char* cmd)
{
    // como se muestra arriba
}

static int
readall(int fd, char buf[], int nbuf)
{
    int tot, nr;

    for (tot = 0; tot < nbuf; tot += nr) {
        nr = read(fd, buf+tot, nbuf-tot);
        if (nr < 0) {
            return -1;
        }
        if (nr == 0) {
            break;
        }
    }
    return tot;
}

int
main(int argc, char* argv[])
{
    int    fd, nr;
    char    buf[1024];

    fd = pipefrom("who");
    if (fd < 0)
        err(1, "pipefrom");
    nr = readall(fd, buf, sizeof buf - 1);
    close(fd);
    if (nr < 0) {
        err(1, "read");
    }
    buf[nr] = 0;
    printf("command output:\n%s--\n", buf);
    exit(0);
}
```

El programa incluye una función auxiliar, `readall`, que lee todo el contenido desde el descriptor indicado dejándolo en un buffer. Esta podría ser una ejecuciæon del programa:

```
unix$ pwho
nemo     console  Jul 13 07:30
nemo     ttys000  Jul 13 07:31
nemo     ttys001  Aug 18 15:59
nemo     ttys002  Aug 20 18:48
unix$
```

6. Sustitución de comandos

El shell incluye sintaxis que realiza un trabajo similar al código que acabamos de ver. Se trata de la la llamada *sustitución de comandos* o *interpolación de salida* de comandos. La idea es utilizar un comando *dentro* de una línea de comandos para que dicho comando escriba el texto que debiéramos escribir nosotros en caso contrario.

Por ejemplo, podríamos querer guardar en una variable de entorno la fecha actual. Para ello deberíamos ejecutar `date` y luego escribir una línea de comandos que asigne a una variable de entorno lo que `date` ha escrito. Pero, por un lado, esto no es práctico y, por otro, si queremos ejecutar estos comandos como parte de un script, no queremos que un humano tenga que editar el script cada vez que lo ejecutamos.

Por ejemplo, tal vez queremos hacer un script llamado `mkvers` que genere un fichero fuente en C que declare un array que defina la versión del programa con la fecha actual.

Veamos cómo hacerlo. Queremos tener un fichero que contenga, por ejemplo,

```
char vers[] = "vers: Fri Aug 26 17:05:14 CEST 2016";
```

para utilizarlo junto con otros ficheros al construir nuestro ejecutable. El plan es que cada vez que hagamos un cambio significativo, ejecutaremos

```
unix$ mkvers
```

y luego compilaremos el programa. De ese modo el programa podría imprimir (si así se le pide) la fecha que corresponde a la versión del programa.

Este podría ser el script:

```
#!/bin/sh
DATE=‘date‘

(
    echo -n ’char vers[] = "vers: ’
    echo -n "$DATE"
    echo  ’";’
) > vers.c
```

La linea que nos interesa es

```
DATE=‘date‘
```

que es similar a haber escrito

```
DATE="Fri Aug 26 17:12:10 CEST 2016"
```

si es que esa era la fecha.

Lo que hace el shell cuando una línea de comandos contiene "‘...‘" es ejecutar el comando que hay entre "‘...‘" y sustituir ese texto de la línea de comandos por la salida de dicho comando. También se dice que el

shell interpola la salida de dicho comando en la línea de comandos. Para conseguir eso, el shell ejecuta código similar a `pipefrom()` y lee la salida del comando. Una vez la ha leído por completo, la utiliza como parte de la línea de comandos y continúa ejecutándola.

Por ejemplo, el comando *seq(1)* se utiliza para contar. Resulta muy útil para numerar cosas. Por ejemplo,

```
unix$ seq 4
1
2
3
4
unix$
```

Como puedes ver en la salida de este comando

```
unix$ echo x ‘seq 4‘ y
x 1 2 3 4 y
unix$
```

el shell ha ejecutado `echo` con "`1 2 3 4`" en la línea de comandos: ha reemplazado el comando entre comillas invertidas por la salida de dicho comando. ¡Y ha reemplazado los "`\n`" en dicha salida por espacios en blanco! Piensa que es una *línea* de comandos y no queremos fines de línea en ella.

Es *muy* habitual utilizar la sustitución de comandos, sobre todo a la hora de programar scripts. Por ejemplo, este libro tenía inicialmente ficheros llamados `ch1.w` para el primer capítulo, `ch2.w` para el segundo, etc.

Podemos crear estos ficheros utilizando un bucle `for` en el shell y la sustitución de comandos. Lo primero que haremos será guardar en una variable los números para los 6 capítulos que pensábamos escribir inicialmente.

```
unix$ caps=‘seq 6‘
unix$ echo $caps
1 2 3 4 5 6
unix$
```

Y ahora podemos ejecutar el siguiente comando que crea cada uno de los ficheros:

```
unix$ for c in $caps
> do
>     echo creating chap $c
>     touch ch$c.w
> done
creating chap 1
creating chap 2
creating chap 3
creating chap 4
creating chap 5
creating chap 6
unix$ echo ch*.w
ch1.w ch2.w ch3.w ch4.w ch5.w ch6.w
unix$
```

Como puedes ver, el comando `for` es de nuevo parte de la sintaxis del shell, similar al comando `if` que vimos anteriormente. En este caso, este comando ejecuta las líneas de comandos entre `do` y `done` (el cuerpo del bucle) para cada uno de los valores que siguen a `in`, haciendo que la variable de entorno cuyo

nombre precede a `in` contenga el valor correspondiente en cada iteración. Si no has entendido este párrafo, mira la salida del comando anterior y fíjate en qué es `$c` en cada iteración.

Podríamos haber escrito este otro comando

```
unix$ for c in ‘seq 6‘ ; do
>     echo creating chap $c
>     touch ch$c.w
> done
```

y el efecto habría sido el mismo.

Poco a poco vemos que podemos utilizar el shell para combinar programas existentes para hacer nuestro trabajo. ¡Eso es UNIX!. Recuerda...

- La primera opción es utilizar el manual y encontrar un programa ya hecho que puede hacer lo que deseamos hacer
- La segunda mejor opción es combinar programas existentes utilizando el shell para ello
- La última opción es escribir un programa en C para hacer el trabajo.

7. Pipes con nombre

En ocasiones deseamos poder conectarnos con la entrada/salida de un proceso *después* de que dicho proceso comience su ejecución. Un pipe resulta útil cuando podemos crearlo antes de crear un proceso, y hacer que dicho proceso "herede" los descriptores del pipe. Pero una vez creado, ya no es posible crear un pipe compartido con el proceso.

Existe otra abstracción en UNIX que consiste en un pipe con un nombre en el sistema de ficheros. Se trata de otro tipo de *i-nodo*, llamado **fifo**. Es posible crear un fifo con la llamada *mkfifo(2)*, que tiene el mismo aspecto que *creat(2)*, pero crea un fifo en lugar de un fichero regular. Igualmente, podemos utilizar el comando *mkfifo(1)* para crearlo.

Veamos una sesión de shell que utiliza fifos. Primero vamos a crear uno en `/tmp/namedpipe`:

```
unix$ mkfifo /tmp/namedpipe
unix$ ls -l /tmp/namedpipe
prw-r--r--  1 nemo  wheel  0 Aug 27 09:03 /tmp/namedpipe
unix$
```

Observa que `ls` utiliza una "`p`" para mostrar el tipo de fichero. El fichero es en realidad un pipe. Si utilizamos *stat(2)*, podemos utilizar la constante `S_IFIFO` para comprobar el campo `st_mode` de la estructura `stat` y ver si tenemos un fifo entre manos.

Una vez creado, el fifo se comporta como un pipe. Todo depende de si lo abrimos para leer o para escribir. Por ejemplo, en esta sesión

```
unix$
unix$ cat /tmp/namedpipe &
[1] 16443
unix$ echo hola >/tmp/namedpipe
hola
[1]+  Done                    cat /tmp/namedpipe
unix$
```

vemos como `cat` comienza a ejecutar y queda bloqueado intentando leer del fifo. Una vez ejecutamos `echo` y hacemos que el shell abra el fifo para escribir, `cat` puede leer. En realidad tenemos a `cat` leyendo del extremo de lectura del pipe y a `echo` escribiendo del extremo de escritura del pipe. Una vez `echo`

termina y cierra su salida estándar, `cat` recibe una indicación de fin de fichero (lee `0` bytes) y termina.

Si utilizamos el mismo fifo de nuevo, vemos que funciona de modo similar una vez más:

```
unix$ echo hola > /tmp/namedpipe &
[1] 16462
unix$ cat /tmp/namedpipe
hola
[1]+  Done                    echo hola > /tmp/namedpipe
unix$
```

En este caso, `echo` (en realidad el shell al procesar el ">") abre el fifo para escribir y se bloquea hasta que algún proceso lo tenga abierto para leer. Una vez `cat` lee el fichero, `echo` puede escribir y termina.

Igual que sucede con un pipe creado con *pipe(2)*, UNIX se ocupa de bloquear a los procesos que leen y escriben en el pipe para que todo funcione como cabe esperar.

El siguiente programa muestra como podríamos utilizar un fifo para leer comandos. Podríamos utilizar algo similar para dotar a una aplicación de una consola a la que nos podemos conectar abriendo un fifo.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <err.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>

int
main(int argc, char* argv[])
{
    char buf[1024];
    int fd, nr;

    if(mkfifo("/tmp/fifo", 0664) < 0)
        err(1, "mkfifo");
    for(;;){
        fprintf(stderr, "opening...\n");
        fd = open("/tmp/fifo", O_RDONLY);
        if(fd < 0) {
            err(1, "open");
        }
```

```
        for(;;){
            nr = read(fd, buf, sizeof buf - 1);
            if(nr < 0){
                err(1, "read");
            }
            if(nr == 0){
                break;
            }
            buf[nr] = 0;
            fprintf(stderr, "got [%s]\n", buf);
            if(strcmp(buf, "bye\n") == 0){
                close(fd);
                unlink("/tmp/fifo");
                exit(0);
            }
        }
        close(fd);
    }
    exit(0);
}
```

El programa abre una y otra vez `/tmp/fifo` (tras crearlo) y lee cuanto puede del mismo. Si consigue leer "`bye\n`", entiende que queremos que el programa termine y así lo hace.

¡Vamos a ejecutarlo!

```
unix$ ./rdfifo &
[1] 16561
unix$ opening...
```

A la vista de los mensajes, `rdfifo` está en la llamada a `open`. Estará bloqueado hasta que otro proceso abra el fifo para escribir en el mismo. Si hacemos tal cosa

```
unix$ echo hola >/tmp/fifo
got [hola
]
opening...
unix$
```

vemos que `read` lee lo que `echo` ha escrito en el fifo, lo que quiere decir que `open` consiguió abrir el fifo para leer del mismo y que `read` pudo obtener varios bytes. Puede verse también que una siguiente llamada a `read` obtuvo `0` bytes y el programa ha vuelto a intentar abrir el fifo. Eso sucede en cuanto `echo` termina y cierra su descriptor.

Podemos ver esto en esta otra sesión de shell, en la que vamos a escribir varias veces utilizando en mismo descriptor de fichero:

```
unix$ (echo hola ; echo caracola) > /tmp/fifo
got [hola
]
got [caracola
]
opening...
unix$
```

Esta vez, varios `read` han podido leer del fifo en nuestro programa. Cuando el último `echo` termina, se cierra el fifo para escribir, lo que hace que nuestro programa reciba una indicación de EOF e intente

reabrirlo de nuevo.

Naturalmente, podemos hacer que nuestro programa termine utilizando

```
unix$ echo bye >/tmp/fifo
got [bye
]
[1]+  Done                    ./8.fifo
unix$
```

8. Señales

Otro mecamismo de intercomunicación de procesos es la posibilidad de enviar mensajes a un proceso dado y la posibilidad de actuar cuando recibimos un mensaje en un proceso. Si estás pensando en la red o en TCP/IP, no nos referimos a eso.

UNIX permite enviar mensajes consistentes en números enteros concretos a un proceso dado. A estos mensajes se los denomina **señales** (*signals* en inglés). Cada mensaje tiene un significado concreto y tiene asociada una acción por defecto.

Cuando un proceso entra al kernel (o sale) UNIX comprueba si tiene señales pendientes y el código del kernel hace que cada proceso procese sus propias señales. Así es como funciona, aunque puedes ignorar este detalle. La recepción de una señal

- puede ignorarse,
- puede hacer que ejecute un manejador para la misma, o
- puede matar al proceso.

En breve vamos a ver qué supone esto en realidad.

El comando de shell capaz de enviar una señal a otro proceso es *kill(1)* y la llamada al sistema para hacerlo desde C es *kill(2)*. Estos son algunos de los valores según indica *kill(1)*:

```
unix$ man kill
...
              Some of the more commonly used signals:
     1        HUP (hang up)
     2        INT (interrupt)
     3        QUIT (quit)
     6        ABRT (abort)
     9        KILL (non-catchable, non-ignorable kill)
     14       ALRM (alarm clock)
     15       TERM (software termination signal)
...
```

Por ejemplo, tras ejecutar

```
unix$ sleep 3600 &
[1] 15984
unix$
```

podemos ejecutar `kill` para enviarle la señal `TERM` (simplemente "`15`") al proceso con pid `15984`:

```
unix$ kill 15984
[1]+  Terminated: 15          sleep 3600
unix$
```

Como puedes ver, por omisión, la señal `TERM` hace que UNIX mate el proceso que la recibe (salvo que dicho proceso cambie la acción por defecto para esta señal).

Incluso si un comando pide a UNIX que ignore la señal `TERM`, es imposible ignorar la señal `KILL`. Así pues,

```
unix$ kill -9 15984
```

es más expeditivo que el comando anterior, y pide a UNIX que mate el proceso con pid `15984` en cualquier caso (supuesto que tengamos permisos para hacer tal cosa, claro está).

Cuando UNIX inicia un *shutdown* (el administrador decide apagarlo, o pulsas el botón de encendido/apagado) primero envía la señal `TERM` a todos los procesos. Esto permite a dichos procesos enterarse de que el sistema está terminando de operar y podrían salvar ficheros que necesiten salvar o terminar ordenadamente si es preciso. Aquellos procesos a los que no importa esto simplemente habrán dejado que `TERM` los mate.

Pasados unos segundos, UNIX envía la señal `KILL` a todos los los procesos vivos, lo que efectivamente los mata. Puedes pensar que `TERM` significa "*por favor, muérete*" y que `KILL` es una puñalada trapera por la espalda.

Otra señal que seguramente has utilizado es `INT`. Se utiliza para interrumpir la ejecución de un proceso. Es de hecho la señal que se utiliza cuando pulsas *Control* y la tecla `c` antes de soltar *Control. Normalmente decimos "*control-c" o escribimos `^C` para representar esto.

Veámoslo. Pero primero hagamos un programa que podamos modificar luego para ver qué hacen las señales.

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

int
main(int argc, char* argv[])
{
    fprintf(stderr, "pid %d sleeping...\n", getpid());
    if(sleep(3600) != 0) {
        warn("sleep");
    }
    fprintf(stderr, "done");
    exit(0);
}
```

Este programa duerme una hora tras imprimir un mensaje informando de su *pid*. Además, hemos comprobado si `sleep` ha fallado e imprimimos un mensaje tras dormir. Si lo ejecutamos

```
unix$ signal1
pid 16056 sleeping...
```

vemos que no obtenemos el prompt del shell. El proceso está en `sleep`. Si pulsamos ahora *control-c* vemos que la ejecución del programa se interrumpe.

```
unix$ signal1
pid 16056 sleeping...
^C
unix$
```

Obtenemos igual resultado si desde otro shell ejecutamos

```
unix$ kill -INT 16056
```

El programa termina igualmente. Si en cambio le envíamos una señal `TERM`, se nos informa de que el proceso ha terminado, aunque el proceso muere igualmente.

```
unix$ signal1
pid 16102 sleeping...
```

Y ejecutando en otro shell

```
unix$ kill -TERM 16102
```

vemos que en shell original el proceso muere:

```
unix$ signal1
pid 16102 sleeping...
Terminated: 15
unix$
```

Cambiemos nuestro programa para pedirle a UNIX que la acción al recibir la señal `INT` no sea morir, sino ignorar la señal.

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

int
main(int argc, char* argv[])
{
    signal(SIGINT, SIG_IGN);
    fprintf(stderr, "pid %d sleeping...\n", getpid());
    if(sleep(3600) != 0) {
        warn("sleep");
    }
    fprintf(stderr, "done");
    exit(0);
}
```

Esto lo hacemos utilizando *signal(3)* para pedir que acción por defecto sea ignorar la señal que indicamos (`SIGINT` simplemente es un `2`). La constante `SIG_IGN` indica que queremos ignorar dicha señal.

Ahora volvemos a ejecutar el programa en un shell y pulsar *control-c* varias veces...

```
unix$ signal2
pid 16058 sleeping...
^C^C
```

Esta vez el programa sigue ejecutando como si tal cosa. Podemos utilizar otro shell para ejecutar

```
unix$ kill  16058
```

y matar el proceso (por omisión, `kill` envía la señal `TERM`).

No es muy amable por parte del programa ignorar la señal de interrupción. Seguramente el usuario acabe frustrado sin poder interrumpir el programa y lo mate de todos modos.

Algo más habitual es utilizar *signal* para pedir que cuando se nos envíe la señal `INT` el programa ejecute un manejador para atender dicha señal. Esto es prácticamente lo que sucede con una interrupción software, pero las señales son una abstracción de UNIX para hablar con los procesos, no son interrupciones

realmente.

Esta nueva versión del programa pide a UNIX que cuando el proceso reciba la señal `INT` se ejecute la función `handleint`.

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

static void
handleint(int no)
{
    fprintf(stderr, "hndlr got %d\n", no);
}

int
main(int argc, char* argv[])
{
    signal(SIGINT, handleint);
    fprintf(stderr, "pid %d sleeping...\n", getpid());
    if(sleep(3600) != 0) {
        warn("sleep");
    }
    fprintf(stderr, "done\n");
    exit(0);
}
```

Si lo ejecutamos y pulsamos *control-c* o utilizamos `kill` desde otro shell para enviar la señal `INT`, esto es lo que sucede:

```
unix$ signal3
pid 16150 sleeping...
^C
hndlr got 2
signal3: sleep: Interrupted system call
done
unix$
```

Lo primero que vemos es que *mientras* el programa estaba dentro de `sleep`, el proceso ha recibido la señal. Considerando la llamada a `signal` que hemos hecho, UNIX ajusta la pila (de usuario) del proceso para que cuando continúe ejecutando ejecute en realidad `handleint`. Por eso vemos que lo siguiente que sucede es que `handleint` imprime su mensaje. Una vez `handleint` (el manejador de la señal) termina, su retorno hace que volvamos a llamar a UNIX. Esto lo ha conseguido UNIX ajustando la pila de usuario, nosotros no hemos hecho nada especial en el código como puedes ver. El kernel hace ahora que el programa continúe por donde estaba cuando recibió la señal. En nuestro caso estábamos dentro de `sleep` y, dado que lo hemos interrumpido, `sleep` falla y retorna `-1`. Puedes ver el mensaje de error que imprime `warn` en nuestro código justo detrás del mensaje que ha impreso el manejador. Por último, el programa escribe `done` y termina.

Es preciso tener cuidado con el código que programamos en los manejadores de señal. Dado que el programa podría estar ejecutando cualquier cosa cuando UNIX hace que se interrumpa y ejecute el manejador de la señal, sólo deberíamos utilizar funciones **reentrantes** dentro de los manejadores de señal. Decimos que una función es *reentrante* si es posible llamarla antes de que una llamada en curso termine. Por ejemplo, las funciones que utilizan variables globales (o almacenamiento *estático* en el sentido de `static` en

C) no son reentrantes. En pocas palabras, cuanto menos haga un manejador de señal mejor. La página de manual *sigaction(2)* y la de *signal(3)* suelen tener mas detalles respecto a qué funciones puede utilizarse dentro de un manejador y cuales no.

Cada vez que se llama a `signal` para una señal se cambia el efecto de dicha señal en nuestro proceso. Para hacer que `INT` recupere su comportamiento por defecto tras haber instalado un manejador, podríamos ejecutar la llamada

```
signal(SIGINT, SIG_DFL);
```

y UNIX olvidaría que antes teníamos un manejador instalado. La constante `SIG_DFL` indica que queremos que la señal en cuestión tenga la acción por defecto.

Los manejadores de señal que tenemos instalados (incluyendo si consisten en ignorar la señal) son atributos del proceso. Cada proceso tiene los suyos. Un cambio en un proceso a este respecto no afecta a otros procesos.

La página de manual *signal(3)* detalla la lista completa de señales, qué acción por defecto tienen y si se pueden ignorar o no.

Las llamadas al sistema interrumpidas por una señal suelen devolver una indicación de error y dejan `errno` al valor `EINTR` (normalmente), que significa "*interrupted*".

Pero `read` y `write` tienen un comportamiento especial. Cuando se interrumpen por el envío de una señal, UNIX re-inicia las llamadas al sistema tras la interrución (y tras la ejecución del manejador si lo hay). Naturalmente, salvo que la acción de la señal en cuestión sea matar al proceso. Eso quiere decir que nuestro programa continuaría ejecutando `read` si es que `read` se ha interrumpido, tras ejecutar el manejador que hemos instalado antes para `INT`.

Para cambiar este comportamiento, podemos utilizar *siginterrupt(3)* y pedir que una señal interrumpa las llamadas al sistema que pueden re-iniciarse tras una señal, esto es, `read` y `write` principalmente. Por ejemplo, ejecutando

```
siginterrupt(SIGINT, 1);
```

incluso `read` se interrumpirá y dejará en `errno` el valor `EINTR`. En cambio, tras

```
siginterrupt(SIGINT, 0);
```

las llamadas a `read` interrumpidas volverán a reiniciarse automáticamente.

Dos señales útiles son `USR1` y `USR2`. Suelen estar disponibles para lo que nos plazca. Un uso habitual es hacer que el programa vuelque información de depuración, por ejemplo, cuando se le envíe `USR1`, o que relea la configuración. También suele utilizarse la señal `HUP` para que el programa relea la configuración, aunque esta señal no es precisamente para eso.

9. Alarmas

Podemos combinar las señales con un temporizador que tiene cada proceso para implementar time-outs. El temporizador es otra abstracción que suministra UNIX para cada proceso. Podemos programarlo para que envíe la señal `ALRM` pasado un tiempo, e instalar el manejador que queramos para dicha señal.

Veamos un programa que hace esto:

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

static void
tout(int no)
{
    fprintf(stderr, "interrupted\n");
}

int
main(int argc, char* argv[])
{
    char buf[1024];
    int nr;

    signal(SIGALRM, tout);
    fprintf(stderr, "> ");
    siginterrupt(SIGALRM, 1);    // try with 0
    alarm(15);
    nr = read(0, buf, sizeof buf - 1);
    alarm(0);
    if(nr > 0){
        write(1, buf, nr);
    } else {
        warn("read");
    }
    exit(0);
}
```

Si lo ejecutamos y escribimos una línea antes de que pasen 15 segundos esto es lo que sucede:

```
unix$ 8.alrm
> hola
hola
unix$
```

El programa termina la llamada a `read` y escribe lo que ha leído. Pero si lo ejecutamos y pasan 15 segundos...

```
unix$ 8.alrm
> interrupted
8.alrm: read: Interrupted system call
unix$
```

Primero se interrumpe `read`, y ejecuta el manejador `tout` que hemos instalado (que imprime "`interrupted`"). Luego `read` termina indicando como error `EINTR`, y el programa continúa.

La primera llamada a `alarm` instala el temporizador para que envíe la señal `ALRM` pasados 15 segundos. La segunda llamada a `alarm` (tras `read`) cancela el temporizador.

Observa la llamada a `siginterrupt`, ¿Qué sucede si la quitamos?

Una última advertencia. Los temporizadores han de usarse con sumo cuidado. De hecho, si es posible, es mejor no utilizarlos. Hacen los programas impredecibles y difíciles de depurar. Por ejemplo, ¿Qué pasa si el

usuario tardó 16 segundos en escribir su línea? ¿Por qué falla el programa en esos casos? Tal vez sería mejor dejar el programa bloqueado leyendo en nuestro caso...

10. Terminales y sesiones

El *terminal* es el dispositivo en UNIX que representa la consola, ventana o acceso remoto al sistema utilizado para acceder al sistema. Ya hemos visto que `/dev/tty` representa el terminal en cada proceso y que hay otros dispositivos habitualmente llamados `/dev/tty*` o `/dev/pty*` que representan terminales concretos.

Nos interesa ser conscientes de este dispositivo puesto que es capaz de enviar señales a los procesos y además suministra la entrada/salida en la mayoría de programas interactivos. La idea es que el terminal controla los procesos que lo utilizan y, por ejemplo, es capaz de enviar la señal `INT` si se pulsa *control-c* y es capaz de enviar la señal `HUP` si el usuario sale del terminal (cierra la ventana, desconecta la conexión de red que ha utilizado para conectarse, lo hace un log-out de la consola).

La abstracción es sencilla: UNIX agrupa los procesos en **grupos de procesos** y cada uno pertenece a una **sesión**. A su vez, cada sesión tiene un **terminal de control** (en realidad cada proceso lo tiene).

Por ejemplo, si abrimos una ventana con un shell obtenemos un nuevo terminal. Podemos ver qué terminal estamos utilizando con el comando *tty(1)*. Por ejemplo, en una ventana tenemos

```
unix$ tty
ttys002
unix$
```

y en otra

```
unix$ tty
ttys007
unix$
```

tenemos un terminal distinto.

Pues bien, la *sesión* es la abstracción que representa la sesión en el sistema en cada uno de estos terminales. En nuestro ejemplo (y si no hay ninguna otra consola ni conexión remota, lo cual no es cierto) tendríamos dos sesiones.

Dentro de una sesión tenemos ejecutando un shell y podemos ejecutar una línea de comandos tal como

```
unix$ ls | wc -l
```

o cualquier otra. En este caso, tanto `ls` como `wc` terminarán perteneciendo al mismo *grupo* de procesos. Y si ejecutamos

```
unix$ (sleep 3600 ; echo hi there) &
unix$ ls | wc -l
```

tendremos dos grupos de procesos. La intuición es que podemos utilizar los grupos para agrupar los procesos (de una línea de comandos, por ejemplo).

Así pues tenemos procesos agrupados en grupos que están agrupados a su vez en sesiones. Cada una de estas abstracciones es simplemente otro tipo de datos más implementado por el kernel y como todo lo demás tendrá su propio record con los atributos que deba tener dentro del kernel. Una vez más es tan sencillo como eso.

Un proceso puede iniciar una nueva sesión llamando a

```
setsid();
```

lo que a su vez hace que también cree un nuevo grupo de procesos y se convierta en su líder. La función *setpgid(2)* es la que se utiliza para hacer que un proceso cree un nuevo grupo de procesos y pase a ser su líder (aunque seguirá dentro de la misma sesión).

Es poco habitual que tengas que llamar a estas funciones. Podrías desear que tu programa se deshaga del terminal de control para que continue ejecutando como un proceso en background sin que ningún terminal le envíe señal alguna (para que ejecute como un **demonio**, que es como se conoce a estos procesos). Esto sucederá si estás programando un servidor o cualquier otro programa que se desea que continue su ejecución independientemente de la sesión.

Pero incluso en este caso, lo normal es llamar a *daemon(3)* que crea un proceso utilizando *fork(2)* y hace que su directorio actual sea "`/`" y que no tenga terminal de control. De ese modo puede ejecutar sin molestar. Eso si, si dicho programa vuelve a utilizar un terminal para leer o escribir, seguramente adquiera dicho terminal como terminal de control. Aunque los detalles exactos dependen del tipo de UNIX concreto que utilices. Consulta tu manual.

El comportamiento del terminal está descrito en *tty(4)* y puede cambiarse. Desde el shell disponemos del comando *stty(1)* (*set tty*) y desde C disponemos de una llamada al sistema *ioctl(2)* que es en realidad una forma de efectuar múltiples llamadas que no tienen llamada al sistema propia (simplemente se utiliza una constante para indicar qué operación de control de Entrada/Salida queremos hacer y otros argumentos empaquetan los parámetros/resultados de la llamada en un record).

Por ejemplo, estos son los ajustes de nuestro terminal

```
unix$ stty -a
speed 9600 baud; rows 24; columns 58; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D;
eol = M-^?; eol2 = M-^?; swtch = <undef>; start = ^Q;
stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V;
flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint ignpar -parmrk -inpck -istrip -inlcr -igncr
icrnl ixon -ixoff -iuclc -ixany imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0
cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase
-tostop -echoprt echoctl echoke
unix$
```

Conviene no intimidarse por la cantidad de ajustes. Hemos mostrado todos utilizando el flag `-a` de `stty`. Los terminales eran artefactos mas o menos simples cuando UNIX se hizo allá por los 70, pero hoy día acarrean toda una historia de dispositivos pintorescos.

Una cosa que vemos es que la opción `intr` tiene como valor `^C`. Esto quiere decir que si pulsamos *control-c* el terminal envía la señal *INT* al grupo de procesos que está ejecutando en **primer plano** o **foreground**. Dicho de otro modo, a los involucrados en la línea de comandos que estamos ejecutando en el momento de pulsar `^C`.

Aquellos procesos que hemos ejecutado el **background** o en **segundo plano**, esto es, aquellos que hemos ejecutado con un "`&`" en su línea de comandos pertenecen a otro grupo de procesos y no reciben la señal `INT` cuando pulsamos `^C`.

Podemos cambiar la combinación de teclas que produce la señal de interrumpir el grupo en primer plano.

Esto podemos hacerlo por ejemplo como en

```
unix$ stty intr ^h
unix$
```

Aquí hemos escrito literalmente "^h", no hemos pulsado *control-h*. Tras la ejecución de este comando, *control-c* no interrumpe la ejecución de ningún comando en este terminal. En cambio, *control-h* sí que puede utilizarse para interrumpir al grupo que está ejecutando en foreground.

Para restaurar los valores por defecto o dejarlos en un estado razonable podemos ejecutar

```
unix$ stty sane
```

y volvemos a utilizar ^C para interrumpir.

Otro ajuste interesante es el flag de echo. Si ejecutamos

```
unix$ ssty -echo
```

(quitar el eco) veremos que en las siguientes líneas de comandos no vemos nada de lo que escribimos. No obstante, si escribimos el nombre de un comando y pulsamos *enter* vemos que UNIX lo ejecuta normalmente.

```
unix$ stty -echo
unix$ Sat Aug 27 11:23:24 CEST 2016
```

El programa que escribe en pantalla lo que nosotros escribimos en el teclado es el terminal, y claro, si le pedimos que no haga eco de lo que escribimos, deja de hacerlo. Podría ser útil para leer contraseñas y para alguna otra cosa...

Para dejar el terminal en su estado normal podemos ejecutar

```
unix$ stty echo
```

aunque no veamos por el momento lo que escribimos. Aunque si es una ventana, es más simple cerrarla y abrir otra.

10.1. Modo crudo y cocinado

Un ajuste importante es el modo de **disciplina de línea** del terminal. Se trata de un parámetro que puede tener como valor *modo crudo* (o *raw*) o bien *modo cocinado* (*cooked*).

Como ya sabes el terminal procesa caracteres según los escribes pero no suministra esos caracteres a ningún programa que lea del terminal hasta que pulsas *enter*. A esto se le llama *modo cocinado*. El terminal "cocina" la línea para permitir que puedas editarla.

Pero en algunas ocasiones querrás procesar directamente los caracteres que escribe el usuario (por ejemplo, si estás implementando un juego o un editor y quieres atender cada carácter por separado). El *modo crudo* sirve justo para esto. Si lo activas, el terminal no cocina nada y se limita a darte los caracteres según estén disponibles. Naturalmente, no podrás borrar y, de hecho, el carácter que utilizabas para borrar pasará a comportarse como cualquier otro carácter.

Con que sepas que estos modos existen tenemos suficiente. Puedes utilizar el manual y las llamadas que hemos mencionado recientemente para aprender a utilizar estos modos, aunque es muy poco probable que lo necesites.