Introducción a Sistemas Operativos: Comunicación entre Procesos

Clips xxx
Francisco J Ballesteros

1. 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 1 muestra cómo los procesos en esta última línea de comandos quedan interconectados por un pipe.
En la figura hemos representado los descriptores como flechas y utilizado números para indicar de qué descriptor se trata en cada caso.

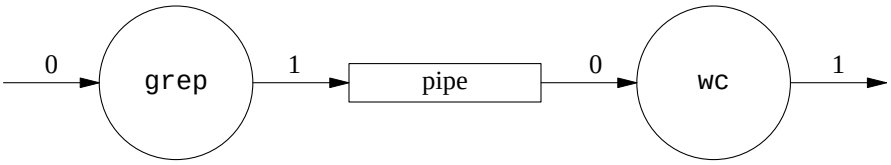

Figura 1: *Utilizando un pipe para enviar la salida de* `grep` *a la entrada de* `wc`.

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.

2. 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.

3. 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 2.

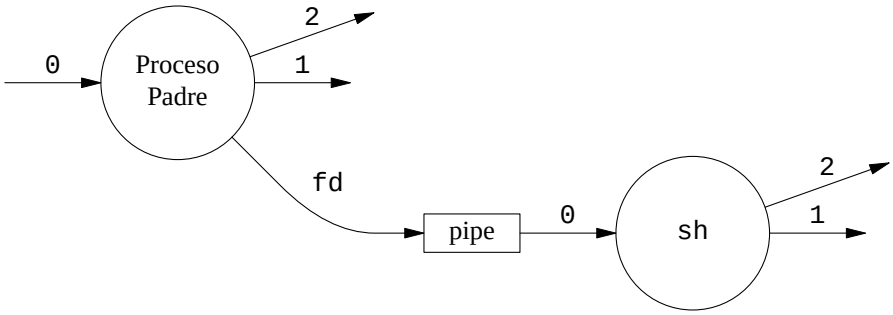

Figura 2: *Descriptores tras la llamada a pipeto mientras ejecutan ambos procesos.*

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`.