Introducción a Sistemas Operativos: Comunicación entre Procesos

Clips xxx
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 "`>>`"

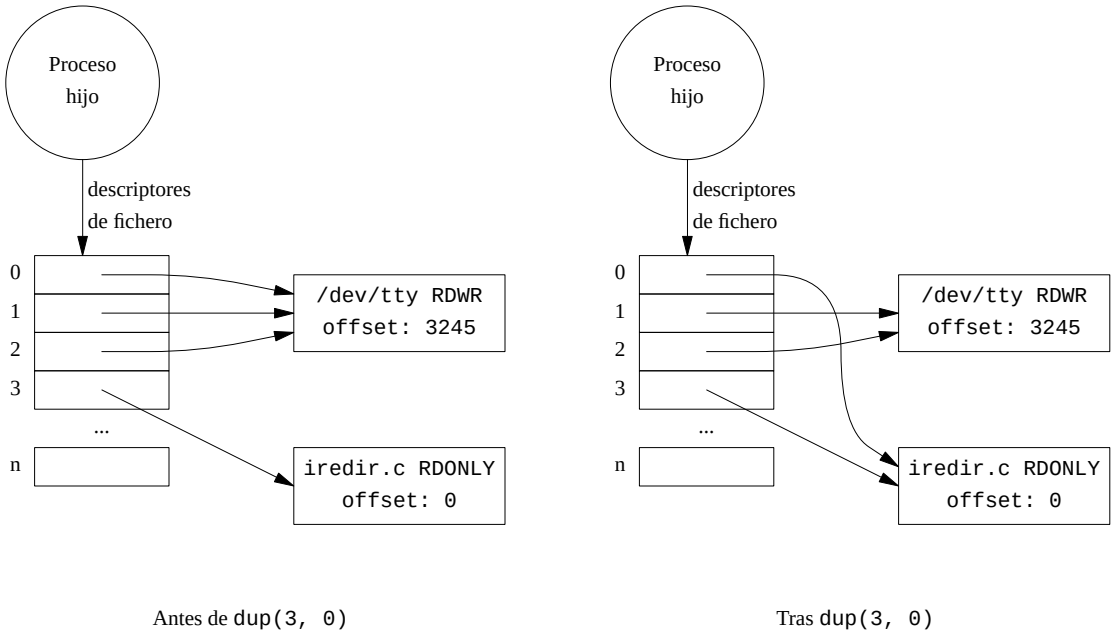

Figura 1: *Procesos antes y después de duplicar el descriptor 3 en el 0.*

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.