Introducción a Sistemas Operativos: Comunicación entre Procesos

Clips xxx
Francisco J Ballesteros

1. 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$
```