Pipes en Linux

Introducción

En este pequeño artículo voy a explicar lo que son los pipes, primero desde el punto de vista de su uso en el shell y luego desde el punto de vista del programador en C. Primero voy a contar cómo eran en DOS, para los memoriosos. Más adelante cuento cómo se usan en C y qué opciones hay.

Pipes en la línea de comandos

Sin pipes, ¿cómo haría uno para llevar información de un comando a otro? Lo primero que se puede pensar podría ser guardar esa información en un archivo temporal. Esto tiene varias desventajas:

- Una acción a la vez, desaprovechando la capacidad del procesador.
- Es necesario esperar a que el primer proceso termine de procesar todo para que empiece el otro y quizá el segundo proceso solamente necesitaba un cachito de información.
- Para grandes volumenes de información es un sistema lento y usa mucho disco.

A cualquiera con un mínimo conocimiento sobre manejo de DOS desde la línea de comandos le suena familiar el comando `dir | more`. Y algunos otros saben incluso qué quiere decir eso y explican: "La salida del comando dir es reusada como entrada para el comando more". Y no se equivocan.

En el viejo y obsoleto DOS eso se implementaba guardando toda la salida de **dir** en un archivo temporario, y luego usando ese archivo temporario como entrada para **more**. Es decir, era equivalente a escribir `dir > file.tmp` y luego `more < file.tmp`.

Todas estas limitaciones hicieron que este mecanismo del SO pase desapercibido, y que se conozca solamente como una forma de pausar la salida del comando dir.

Pero el mundo de DOS quedó atrás, gracias a Dios, hace mucho tiempo. El lector podrá preguntar "¿Y? ¿Acaso ahora las cosas son diferentes?". Y yo responderé "Sí". Y el lector preguntará "¿Por qué?". Y yo responderé "¡Seguí leyendo!".

Pipes en Linux

Pipes a nivel shell

Cuando en el shell de UNIX alguien escribe "`gunzip -c zapato.tar.gz | tar xf -`" lo que sucede es lo siguiente:

1. El shell construye un *pipe*, que es un par de "archivos inexistentes", que tienen la cualidad de que lo que se escribe en uno se lee en el otro.
2. Después el shell crea dos procesos diferentes, uno con **gzip** y otro con **tar**. Esos procesos son procesos completamente independientes y corren al mismo tiempo, aprovechando la multitarea del sistema operativo. El proceso con el gzip tiene redireccionada su salida

estándar hacia uno de los "archivos inexistentes" del pipe, y el otro "archivo inexistente" oficia como entrada estándar para el tar.

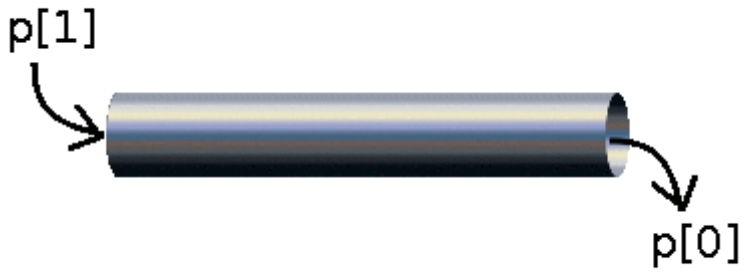
Esto quiere decir que gzip y tar se ensamblan mágicamente en un nuevo utilitario que descomprime y "desarchiva" al mismo tiempo.

¿Y cómo se usa todo esto en un programa?

Para usar pipes desde C sólo hay que llamar a la función **pipe** y pedirle los dos descriptores de archivo que serán los extremos del "tubo" que se habrá construído. Algo así:

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

Después de la invocación a **pipe** el arreglo de enteros p contiene en `p[0]` un descriptor de archivo para *leer*, y en `p[1]` un descriptor de archivo para *escribir*. Esto quiere decir que los pipes funcionan en una sola dirección.



¿Cómo se usan estos “descriptores de archivo” que nos devolvió la función? Bueno, se pueden usar para leer y escribir bytes mediante las las funciones **read**, **write** y **close**, pero usualmente es mucho más conveniente crear un `FILE*` y de esta manera poder usar funciones más cómodas tales como **fread**, **fwrite**, **fprintf**, etc. Esto se logra mediante la función **fdopen**. Por ejemplo para leer de `p[0]` podemos simplemente hacer `FILE *f=fdopen(p[0], "r+")`, y ya disponemos de un `FILE*` listo para leer los bytes que emergen del pipe. Para escribir (por ejemplo en p[1]), se hace lo mismo pero se debe pasar como segundo parámetro "w" (al igual que en un **fopen** normal).

> En verdad los pipes no son exclusivos de Linux, en Windows también se pueden crear mediante la función CreatePipe.

Pipes y procesos hijo

Todo lo que escribí hasta ahora muestra como usar un pipe dentro de un único proceso. Esto raramente tiene sentido. Un pipe pasa a ser más útil en el caso de comunicar un proceso padre con un proceso hijo. Pero `p[0]` y `p[1]`, cuando son creados existen solamente en el proceso en el que fueron creados: esos file descriptors son solamente válidos allí. El truco es aprovechar que la función fork, que permite crear nuevos procesos, copia todo el proceso del llamador. Y eso incluye a los file descriptors del pipe, pero... el pipe no se duplica! Y esto da como resultado que si el padre escribe en `p[1]`, el hijo puede leer en `p[0]`, o viceversa.

Un pipe sirve solamente para una comunicación unidereccional, si es necesario que padre e hijo se comuniquen en ambos sentidos, deberán usarse dos pipes.

Un último comentario: Si el pipe es para que el padre escriba en `p[1]` se recomienda que cierre `p[0]` (y que el cliente cierre `p[1]`). Es decir que cada parte cierre el extremo del pipe que no va a usar. Una de las razones para hacer esto es que mientas no se haga, el lector del pipe no recibirá EOF cuando el otro lado cierre el pipe, ya que todavía habrá un extremo abierto (aunque no utilizado ni tenido en cuenta).

popen

Existe una función de más alto nivel para el caso en el que se quiera ejecutar algo con su entrada o salida redireccionada desde o hacia un pipe. Esta función es popen. Veamos un ejemplo: Supongamos que tenemos una aplicación llamada fortune, que cada ve que se ejecuta escupe a la salida estándar una simpática frase. También supongamos que desde nuestra aplicación queremos disponer de simpáticas frases. Podemos usar popen de la siguiente manera:

```
FILE *f = popen("fortune", "r");

/* leemos de f la frase */
[ ... ]

pclose(f);
```

Lo de pclose no es un error de tipeo. Si un "stream" se abre con popen, se cierra solamente con pclose. Esto es así porque esta función se ocupa no sólo de cerrar en pipe sino de hacer las limpiezas relacionadas con el haber creado un proceso hijo.

Detalles extra

Nótese que los pipes son unidireccionales, en el último caso, el del popen, nótese también que se puede o tomar la salida o enviar a la entrada de un proceso, pero no ambas cosas. Para manipular entrada y salida estándares es necesario hacerlo a mano, combinando invocaciones a pipe, close, fork, etc. Queda como ejercicio para el lector =).

El mecanismo de pipes tiene incluido un manejo de las congestiones. Es decir que si el lado escritor del pipe se zafa y escribe más de lo que el lado lector lee, el escritor se *bloquea* al intentar escribir nuevamente. El escritor queda bloqueado hasta que el lector se pone al día con lo que tiene pendiente de leer. Ejemplo:

```
grep algo archivo.txt | gzip > archivo
```

El comando grep filtra de archivo todas las líneas que tengan *algo* y las envía mediante un pipe al gzip. Es evidente que el proceso de comprimir es mucho más lento que el de filtrar. Gracias al mecanismo que expliqué antes el proceso escritor, es decir el grep, va a ir pausando automáticamente para esperar al gzip. Simpático ¿no?

Alternativas a los pipes

En un sistema UNIX hay varias otras cosas que son primos cercanos de los pipes. Están los sockets, que son muy similares pero que tienen las siguientes diferencias:

- Son bidireccionales. Con un solo par de file descriptor se puede establecer una comunicación en ambos sentidos.
- Cada extremo del "pipe" (que no lo es en verdad) tiene una "dirección de red". Esa red no necesariamente es de protocolo IP. Si el protocolo de esas direcciones lo admite, la conexión puede estar comunicando dos máquinas diferentes.

Hay otro pariente de los pipes al que se le llama "fifo". Es también una conexión bidireccional. Se obtiene cuando dos procesos abren un mismo archivo especial que fue creado con **mknod** (ver la página de manual de esta función).

Mis otros artículos sobre programación en Linux.

Por Nicolás Lichtmaier. Cualquier comentario o pedido de mayor claridad o extensión será bien recibido.