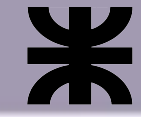


Tuberías

Comunicación entre procesos o IPC (InterProcess Communication)

Frecuentemente los procesos necesitan comunicarse con otros procesos. Los tres puntos a solucionar:

- Cómo pasar información de un proceso a otro.
- Cómo asegurar que dos procesos no se interfieran mientras realizan tareas críticas.
- Cómo secuenciar correctamente cuando existen dependencias.

Diferentes técnicas

Para procesos de una misma PC:

- Tuberías (Pipe, tuberías anónimas): paso de datos entre procesos relacionados.
- FIFO (Tuberías con nombre): paso de datos entre procesos relacionados o no relacionados.
- Cola de mensajes: paso de mensajes (delimitados) entre procesos relacionados o no relacionados.
- Memoria compartida (Objetos de memoria): se utiliza entre procesos relacionados o no relacionados.

Para procesos de una misma PC o de distintas PCs en una red:

- Sockets

Persistencia:

Cuanto tiempo permanece en existencia

Proceso:

- Pipe
- FIFO (los datos de una FIFO tienen persistencia de proceso)
- Socket

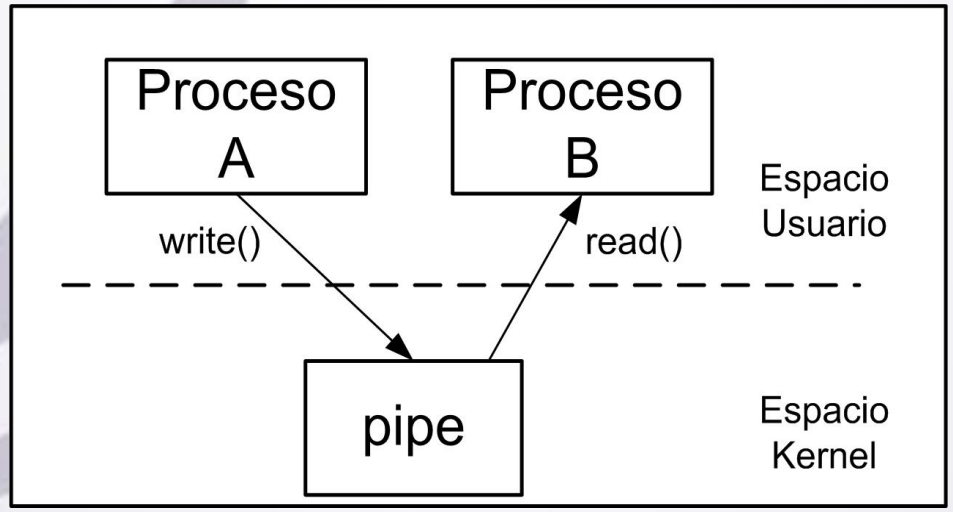
Kernel:

- Cola de mensajes
- Memoria compartida (objetos de memoria)

Tuberías sin nombre - PIPE

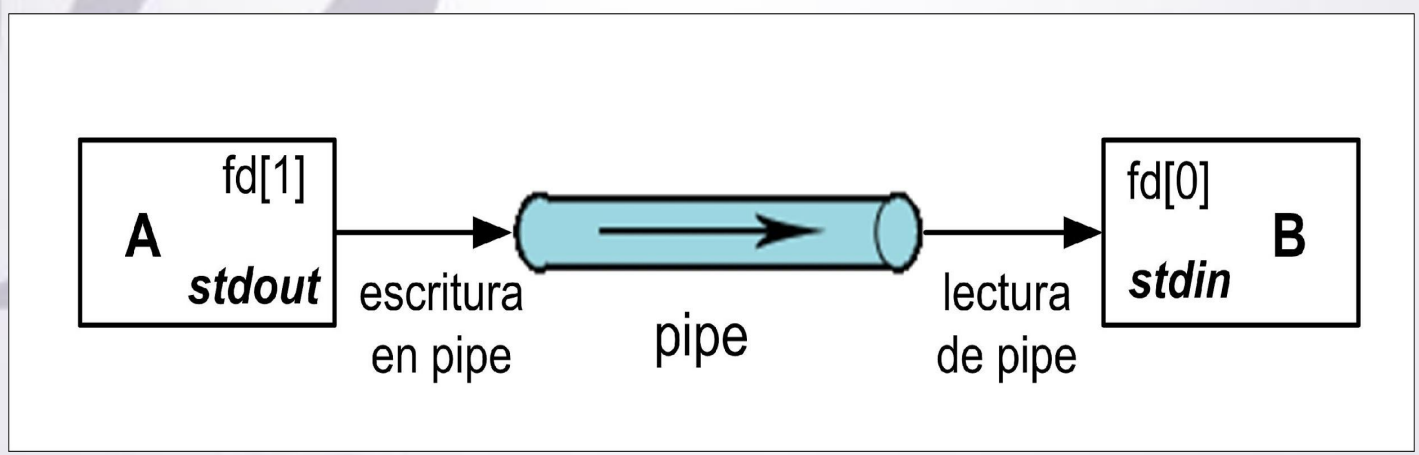
- Usualmente se usa para pasar datos entre procesos relacionados (padre, hijo).
- Buffer tipo FIFO, mantenido en memoria del Kernel. Capacidad limitada.
- Es unidireccional.
- Tienen persistencia de proceso.

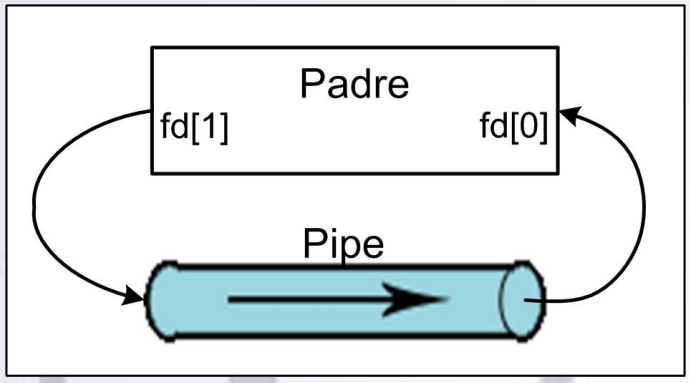
Tubería (pipeline, pipe)



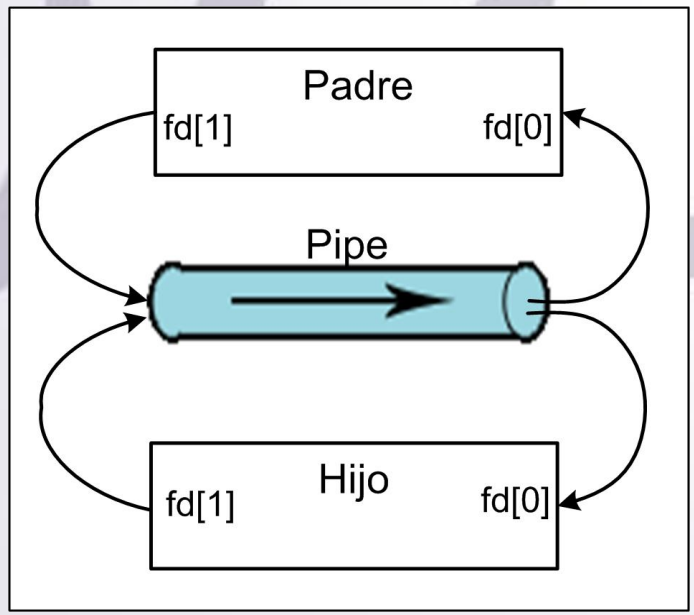
Es un método de conexión que une la salida estándar de un proceso a la entrada estándar de otro. Para esto se utilizan "descriptores de archivos" reservados, los cuales en forma general son:

- 0: entrada estándar (stdin).
- 1: salida estándar (stdout).
- 2: salida de error (stderr).

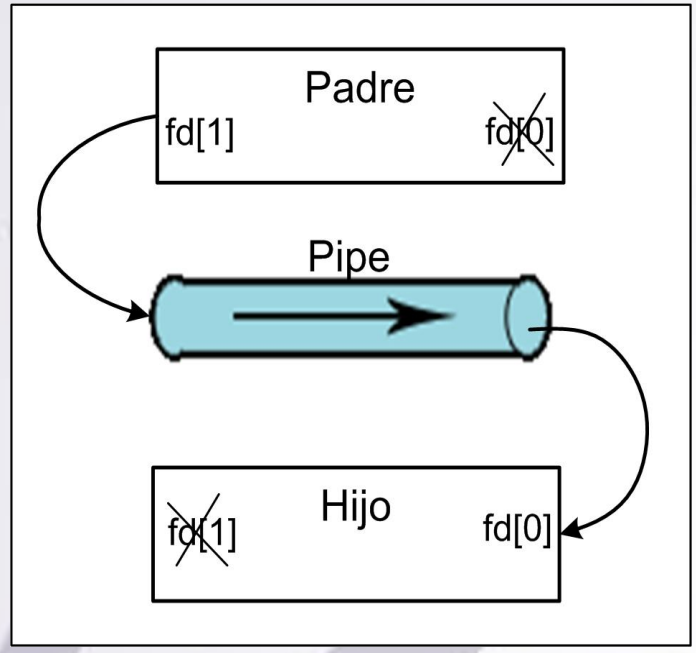




1) Al crear una tubería el proceso se comunica con sí mismo.



2) Para que la tubería pueda comunicarse con otro proceso debemos hacer un fork y crear un hijo. Luego de hacer un fork, la tubería se hereda al proceso hijo.



3) Se recomienda que el padre haga un close() de p[0] (el lado de lectura de la tubería), y el hijo haga un close() de p[1] (el lado de escritura de la tubería) o viceversa. Así cerramos el extremo del pipe que no utiliza.

Creación de tuberías en C

Para crear una tubería **simple** con C, se usa la llamada al sistema *pipe()*. Toma un argumento que es un arreglo de dos enteros, y si tiene éxito, la tabla contendrá dos nuevos descriptores de archivos para ser usados por la tubería.

```
#include <unistd.h>

int pipe(int fd)
```

pipe() devuelve -1 en caso de error, o 0 si tuvo éxito.

Donde donde fd es un arreglo de dos enteros , esos enteros son los descriptores:

- fd[0] es para leer
- fd[1] es para escribir

Escrituras de tuberías

La escritura de hasta `PIPE_BUF` bytes está garantizada de ser atómica.

Para **escribir a la entrada de una tubería** se usa la función **write()**

```
#include <unistd.h>

int write(int fd[1], void *buffer, size_t count);
```

Devuelve el número de bytes escritos o –1 si hubo error.

La función write( ) se bloquea hasta que todos los datos se han escrito en la tubería.

Cuando un proceso intenta escribir en una tubería que tienen los descriptores de lectura cerrados, el kernel envía la señal SIGPIPE al proceso que intente la escritura.

Lectura de tuberías

Para **leer a la salida de una tubería** se usa la función **read()**

```
#include <unistd.h>

int read(int fd[0], void *buffer, size_t count);
```

Devuelve el número de bytes leídos, 0 si EOF o –1 si hubo error.

Cierre de una tubería

Para **cerrar los lados de una tubería** se usa la función **close()**

```
#include <unistd.h>
int close(int fd[x]);
```

Devuelve 0 si hubo éxito o –1 hubo error.

fd[x]: es alguno de los descriptores (fd[0] para leer, o fd[1] para escribir).

Si se cierran todos los descriptores de la tubería, esta se destruye.

Situaciones conflictivas:

- Un proceso que lee una tubería vacía se bloquea.
- Un proceso que escribe en una tubería llena se bloquea.
- Si dos procesos quieren leer desde una misma tubería no se puede determinar quien leyó primero.
- Un proceso trata de escribir en una tubería en la cual ningún proceso tiene un descriptor de lectura abierto. El kernel envía señal SIGPIPE al proceso, por defecto mata el proceso.

Tuberías en línea de comando (shell)

Una tubería es una combinación de varios comandos que se ejecutan en cascada. El resultado del primero se envía a la entrada del siguiente. Esta tarea se realiza por medio del carácter barra vertical pipe “|” .

Este mecanismo es ampliamente usado, en la línea de comandos (shell)

```
$ ls | sort
```

Es un ejemplo de “pipeline”, donde se toma la salida de un comando ls como entrada de un comando sort.

La salida por stdout del primer comando (ls: listar) es reenviada por el stdin del segundo (sort: ordenar alfabéticamente).

Pases de variables por valor o por referencia

Las siguientes funciones,

```
int read  (int fd[0], void *buffer, size_t count);
int write (int fd[1], void *buffer, size_t count);
```

esperan un puntero a un arreglo (`*buffer`, pase por referencia, *passed by reference*) y no el arreglo completo (`buffer[N]`, pase por valor, *passed by value*). Otro prototipo de estas funciones podría ser,

```
int read  (int fd[0], void buffer[N], size_t count);
```

Esta definición supone dos problemas: **1)** Se debe conocer de antemano el tamaño del arreglo (N) y **2)** es ineficiente pasar el arreglo completo.

Cada vez que se invoca una función, sus argumentos de entrada y salida se pasan haciendo un *push* a la pila del proceso (*stack*) que se crea en el espacio de memoria del mismo. Es mucho más eficiente pasar un puntero a un arreglo (entero de 32/64 bits) que el arreglo entero (N variables de 32/64 bits).

Casteo a puntero

Además, las siguientes funciones esperan como entrada un puntero a void,

```
int read  (int fd[0], void *buffer, size_t count);
int write (int fd[1], void *buffer, size_t count);
```

Esto significa que en tiempo de compilación se establece que estas funciones pueden recibir un puntero a cualquier tipo de dato (`int, char, float, uint_32, uint_64`).

Por tanto, cuando se invoca a estas funciones se recomienda hacer un “casteo” del puntero a `buffer` (`*buffer`) y especificar el tipo de dato con el que `buffer` fue creado,

```
char buffer_1 [] = {"Hola\n"};

write (STDOUT_FILENO, (char *)  buffer_1, sizeof(buffer_1));
```

La expresión `(char *)` indica que `buffer_1` es un puntero a un arreglo de `char`.

Bibliografía

Kerrisk, Michael. *The linux programming Interface.* 2011. **Capítulos 43, 44.**