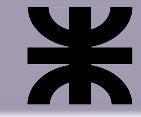


Tuberías

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 ejecutándose en un mismo SO:

- Tuberías (*pipes*, tuberías anónimas): comunicación de **datos** entre procesos relacionados.
- FIFO (tuberías con nombre): comunicación de **datos** entre procesos relacionados o no relacionados.
- Cola de mensajes: comunicación de **mensajes** (delimitados) entre procesos relacionados o no relacionados.

Para procesos ejecutándose en distintos SO (computadores en red):

- Sockets

Persistencia

¿Qué persistencia tiene la información transmitida por un IPC?

Mientras se ejecute el proceso:

- Tuberías.
- FIFO.
- Socket.

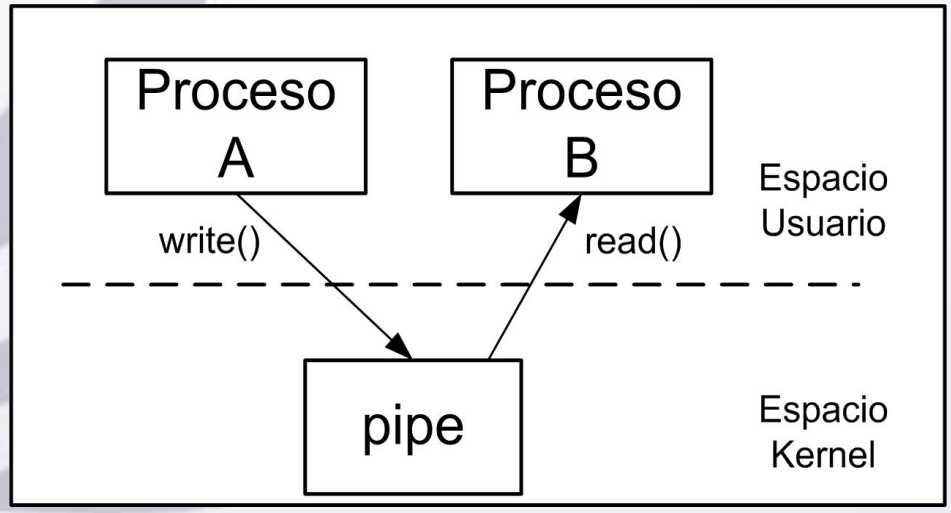
Mientras se ejecute el kernel:

- Cola de mensajes

Tuberías sin nombre - PIPE

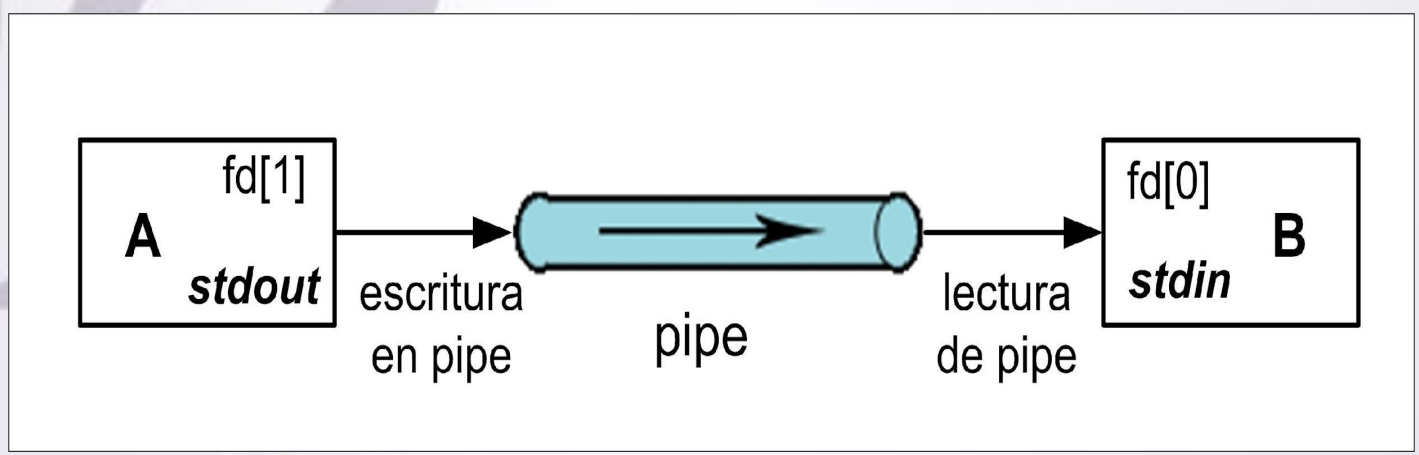
- Usualmente se usa para pasar datos entre procesos relacionados (padre, hijo).
- Buffer tipo FIFO mantenido en memoria del Kernel. Capacidad limitada.
- Es unidireccional.
- Tienen persistencia de proceso.

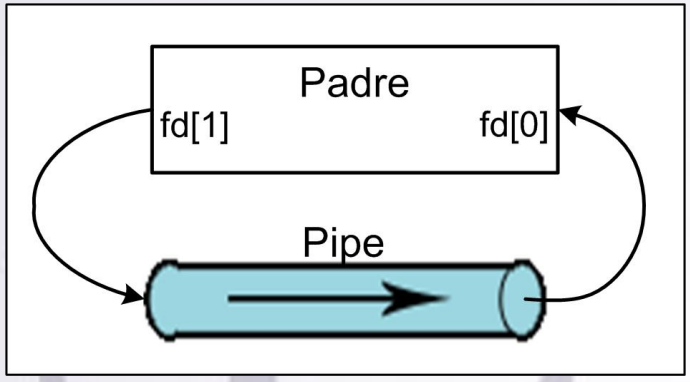
Tubería (pipeline, pipe)



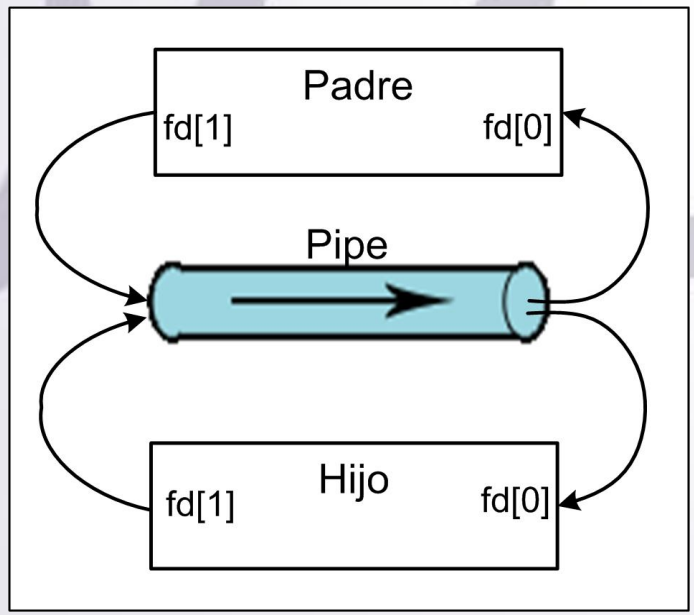
Es un método de conexión que une la salida estándar de un proceso a la entrada estándar de otro. Para esto se utilizan “descriptores de archivos” reservados, los cuales en forma general son:

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

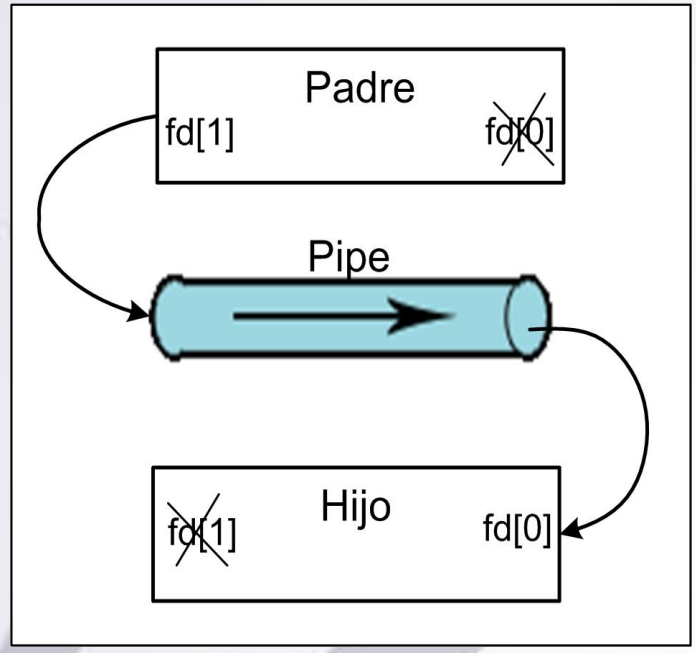




1) Al crear una tubería el proceso se comunica con sí mismo.



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(p[0])`, el lado de lectura de la tubería, y el hijo haga un `close(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 `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

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.

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

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` (End Of File) 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 una tubería, la tubería **se destruye.**

Situaciones conflictivas:

1. Un proceso que lee una tubería vacía se bloquea.

2. Un proceso que escribe en una tubería llena se bloquea.

3. Si dos procesos quieren leer desde una misma tubería no se puede determinar quien leyó primero.

4. Cuando un proceso intenta escribir en una tubería con descriptores de lectura cerrados, el kernel envía la señal `SIGPIPE` al proceso que intenta la escritura y termina el proceso.

```
$ man 7 signal

Signal  Standard  Action  Comment
SIGPIPE P1990     Term    Broken pipe: write to pipe with no
                           readers; see pipe(7)
```

Tuberías en línea de comando (BASH 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 “|” .

En la línea de comandos (BASH):

```
$ 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 defecto (`stdout`) del primer comando (`ls`: listar) es reenviada a la entrada por defecto (`stdin`) del segundo comando (`sort`: ordenar alfabéticamente).

Bibliografía

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

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 de **`buffer`** (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).

Cast 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, double, uint_32, uint_64`).

Por tanto, cuando se invoca a estas funciones se recomienda hacer un “casteo” (cast) del puntero a `buffer` (`*buffer`) y especificar en tiempo de compilación 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 *)` es un casteo e indica que `buffer_1` es un puntero a un arreglo de `char`.