



# **Clase 3**

## **Signals – Pipes**



## **Signals**

- Se envía de un proceso a otro proceso
- No posee datos asociados, solo el número de signal
- Hay un handler que se ejecuta al recibirla
- Existen handlers por defecto, que el programador puede reescribir.



## Signals

- El estándar POSIX define 32 signals

`kill -l`

```
ernesto@ernesto-X401A1:~$ kill -l
```

|                 |                 |                 |                 |                 |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| 1) SIGHUP       | 2) SIGINT       | 3) SIGQUIT      | 4) SIGILL       | 5) SIGTRAP      |
| 6) SIGABRT      | 7) SIGBUS       | 8) SIGFPE       | 9) SIGKILL      | 10) SIGUSR1     |
| 11) SIGSEGV     | 12) SIGUSR2     | 13) SIGPIPE     | 14) SIGALRM     | 15) SIGTERM     |
| 16) SIGSTKFLT   | 17) SIGCHLD     | 18) SIGCONT     | 19) SIGSTOP     | 20) SIGTSTP     |
| 21) SIGTTIN     | 22) SIGTTOU     | 23) SIGURG      | 24) SIGXCPU     | 25) SIGXFSZ     |
| 26) SIGVTALRM   | 27) SIGPROF     | 28) SIGWINCH    | 29) SIGIO       | 30) SIGPWR      |
| 31) SIGSYS      | 34) SIGRTMIN    | 35) SIGRTMIN+1  | 36) SIGRTMIN+2  | 37) SIGRTMIN+3  |
| 38) SIGRTMIN+4  | 39) SIGRTMIN+5  | 40) SIGRTMIN+6  | 41) SIGRTMIN+7  | 42) SIGRTMIN+8  |
| 43) SIGRTMIN+9  | 44) SIGRTMIN+10 | 45) SIGRTMIN+11 | 46) SIGRTMIN+12 | 47) SIGRTMIN+13 |
| 48) SIGRTMIN+14 | 49) SIGRTMIN+15 | 50) SIGRTMAX-14 | 51) SIGRTMAX-13 | 52) SIGRTMAX-12 |
| 53) SIGRTMAX-11 | 54) SIGRTMAX-10 | 55) SIGRTMAX-9  | 56) SIGRTMAX-8  | 57) SIGRTMAX-7  |
| 58) SIGRTMAX-6  | 59) SIGRTMAX-5  | 60) SIGRTMAX-4  | 61) SIGRTMAX-3  | 62) SIGRTMAX-2  |
| 63) SIGRTMAX-1  | 64) SIGRTMAX    |                 |                 |                 |



## Signals

- SIGRT son señales pueden ser usadas por el programador.
- No tienen un uso predefinido.
- Sus números no son fijos, dependen de la implementación.
- Por eso se usan las macros SIGRTMIN y SIGRTMAX, que indican el rango válido.
- Características:
  - Se encolan y se entregan en el orden que se generaron.
  - Se recibe más información de quien envía.



## Signals

- Cómo enviar una signal a un proceso:
  - Por comando:
    - `kill -sn pid`
- Función de la API POSIX:
  - `int kill(pid_t pid, int sig) ;`



| Nombre  | Número | Descripción   |
|---------|--------|---|
| SIGINT  | 2      | Se envía al proceso cuando presionamos ctrl+C desde la terminal   |
| SIGKILL | 9      | Terminación forzada. No puede reescribirse el handler.  |
| SIGUSR1 | 10     | Disponible para uso del programador   |
| SIGUSR2 | 12     | Disponible para uso del programador   |
| SIGALRM | 14     | Alarma configurable   |
| SIGTERM | 15     | Terminar proceso (valor por defecto en comando kill). No actúa si el proceso está bloqueado o escribiendo un archivo. |
| SIGCHLD | 17     | El proceso hijo la envía al padre cuando el hijo finaliza.  |
| SIGCONT | 18     | Reanuda el proceso detenido con SIGSTP  |
| SIGSTP  | 20     | Detiene el proceso (ctrl+Z)   |



## Signals

- La signal **SIGINT** (al presionar ctrl+C) tiene un handler por defecto que hace terminar el proceso.
- Puede reescribirse el handler para hacer otra cosa.
- Algunos handlers no pueden reescribirse (**SIGKILL, SIGSTOP**)



## Signals

- La signal **SIGSTP** (al presionar ctrl+Z) detiene la ejecución del proceso.
- Puede reanudarse enviando la **SIGCONT**
- A diferencia de **SIGSTOP**, se puede reescribir el handler.





## Signals

- La signal **SIGKILL** detiene el proceso sin poder evitarlo
- Se envía con: `kill -9 pid`
- La signal **SIGTERM** es la signal por default del comando kill: `kill pid`



# Signals

## Práctica 1



## Signals

- **jobs:** Lista los procesos en background.
- **bg:** Envía sigcont y desconecta stdin de la terminal.
- **fg:** Envía sigcont y conecta stdin con la terminal



## Signals

- Función kill

```
#include <sys/types.h>  
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```



## Signals

- Función sigaction
- Utilizada para reemplazar el handler de una signal

```
#include <signal.h>
```

```
int sigaction(  
    int sig,  
    const struct sigaction* act,  
    struct sigaction* oact);
```



## Signals

- Argumentos de `sigaction`
- **sig** : signal number
- **act** : struct con la función handler y otros datos que definen su comportamiento
- **oact** : struct del handler instalado previamente



## Signals

- Campos de la struct
- **sa\_handler** : Función handler (o SIG\_IGN)
- **sa\_mask** : Indica si otras signals pueden interrumpir a este handler o no.(ver *sigaddset()*)
- **sa\_flags** : Modifican el comportamiento del handler.



```
void sigint_handler(int sig) {  
    write(1, "Ahhh! SIGINT!\n", 14);  
}
```

```
int main(void) {  
    char s[200];  
    struct sigaction sa;
```

```
    sa.sa_handler = sigint_handler;  
    sa.sa_flags = 0; //SA_RESTART;  
    sigemptyset(&sa.sa_mask);
```

```
    if (sigaction(SIGINT, &sa, NULL) == -1) {  
        perror("sigaction");  
        exit(1);  
    }
```

```
    printf("Enter a string:\n");  
    if (fgets(s, sizeof s, stdin) == NULL)  
        perror("fgets");  
    else  
        printf("You entered: %s\n", s);  
    return 0;
```

```
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <errno.h>  
#include <signal.h>
```

**Presionamos Ctrl+C!**





## Signals

- Qué se puede hacer en el handler
- Algunas funciones no son seguras de ejecutarse
  - Por eso usamos **write()** en vez de **printf()**
- Tampoco podemos modificar variables globales, solo de un tipo especial

```
volatile sig_atomic_t got_usr1;
```

```
void sigusr1_handler(int sig)
{
    got_usr1 = 1;
}
```



# Signals

## Práctica 2



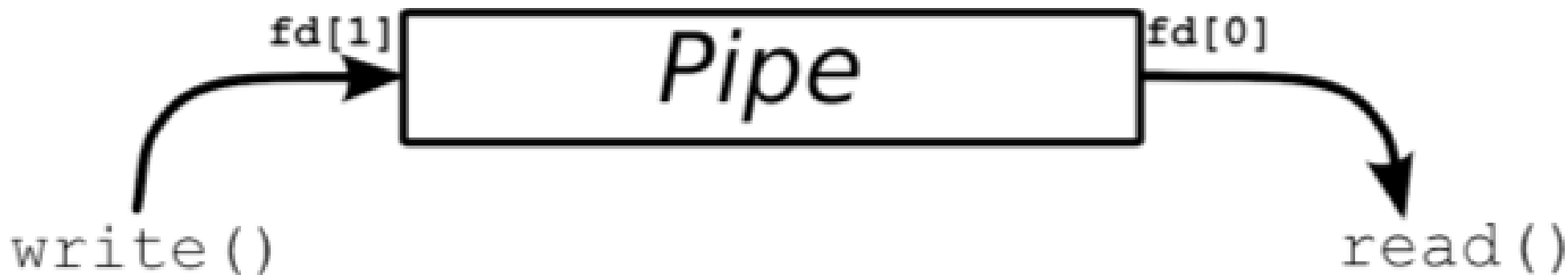
## Pipes

- **File Descriptor:**
  - Representa un archivo mediante un número int.
  - Es la posición en la tabla de archivos asociados a un proc.
  - Equivalente a FILE\* pero en bajo nivel.
  - Se utilizan con las funciones `open()`, `write()`, `read()`, `close()`.
- Existen 3 FD asociados a un proceso
  - Stdin: FD 0
  - Stdout: FD 1
  - Stderr: FD 2
- Los procesos hijos “heredan” los File Descriptors del proceso padre. Por eso vemos los `printf` del proceso hijo en la consola que ejecuto el proceso padre.



## Pipes

- La función **pipe()** crea dos FD asociados al proceso.
- Devuelve un array de dos FD.
- Un FD es de escritura [1]
- Un FD es de lectura [0]
- Los datos que se escriben en el FD de escritura pueden leerse en el FD de lectura



**How a pipe is organized.**



## Pipes

```
int main(void)
```

```
{
```

```
    int pfd[2];
```

```
    char buf[30];
```

```
    if (pipe(pfd) == -1) {
```

```
        perror("pipe");
```

```
        exit(1);
```

```
    }
```

```
    printf("writing to file descriptor #%d\n", pfd[1]);
```

```
    write(pfd[1], "test", 5);
```

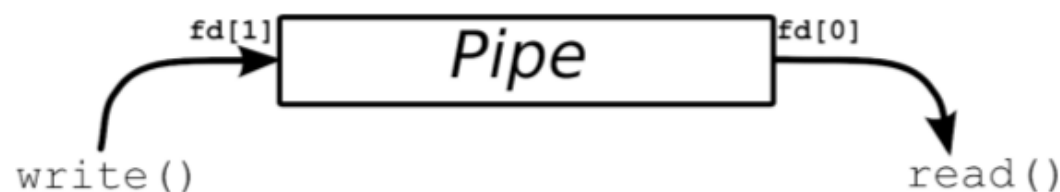
```
    printf("reading from file descriptor #%d\n", pfd[0]);
```

```
    read(pfd[0], buf, 5);
```

```
    printf("read \"%s\"\n", buf);
```

```
    return 0;
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
```



How a pipe is organized.



## **Pipes**

- **El programa anterior no tenía ningún sentido.**
- **Ventaja: Procesos padres e hijos comparten los FD**
- **Combinamos pipe() con fork() para comunicar procesos hijos**



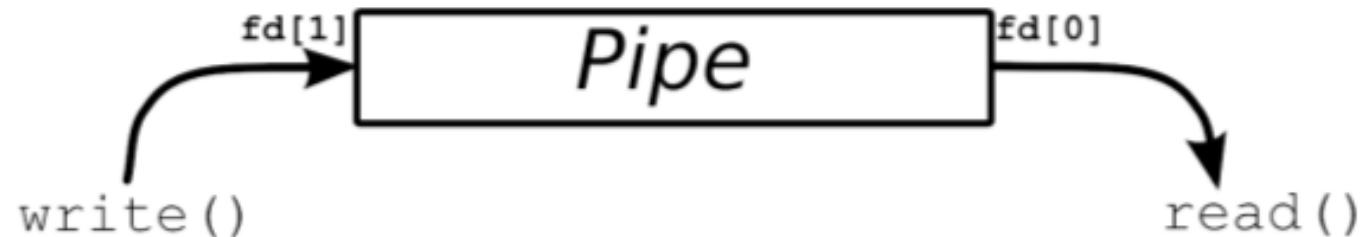
```
int main(void)
{
```

```
    int pfd[2];
    char buf[30];
```

```
    pipe(pfd);
```

```
    if (!fork()) {
        printf(" CHILD: writing to the pipe\n");
        write(pfd[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else {
        printf("PARENT: reading from pipe\n");
        read(pfd[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }
    return 0;
}
```

## Pipes

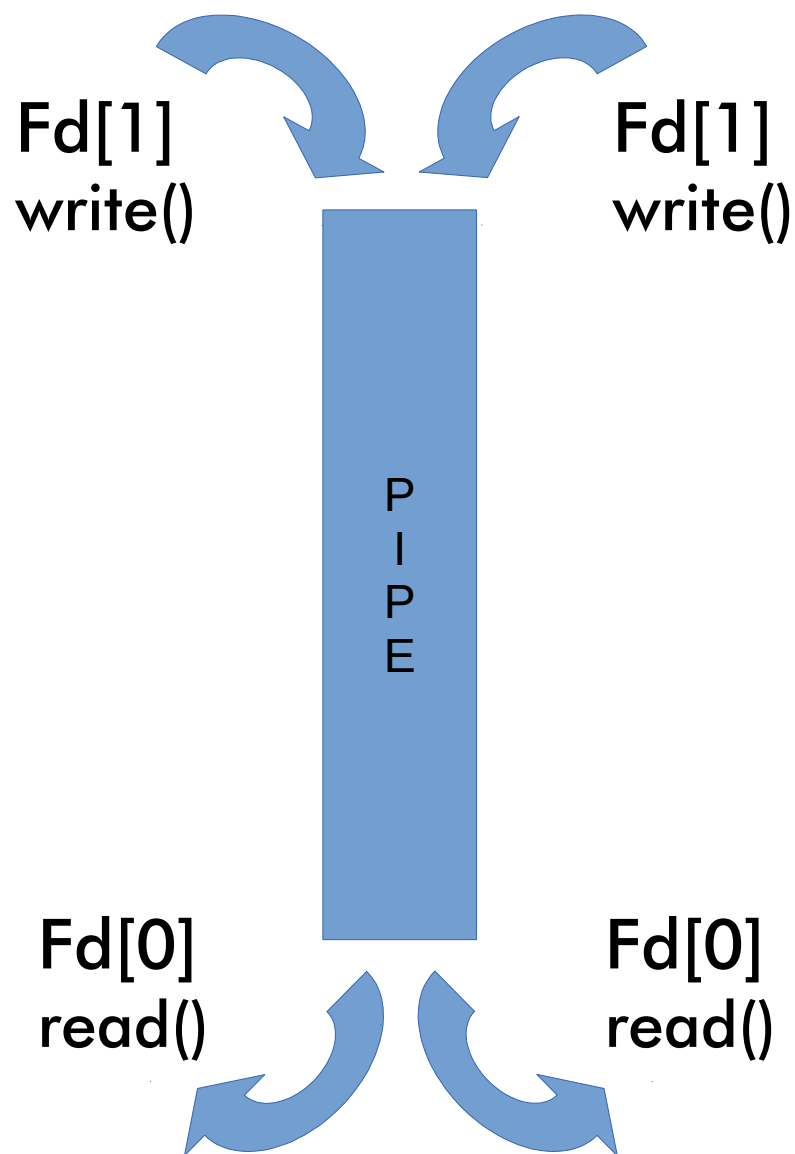


How a pipe is organized.



**Padre**

**Hijo**







**Padre**

**Hijo**

`close(pfds[1])`

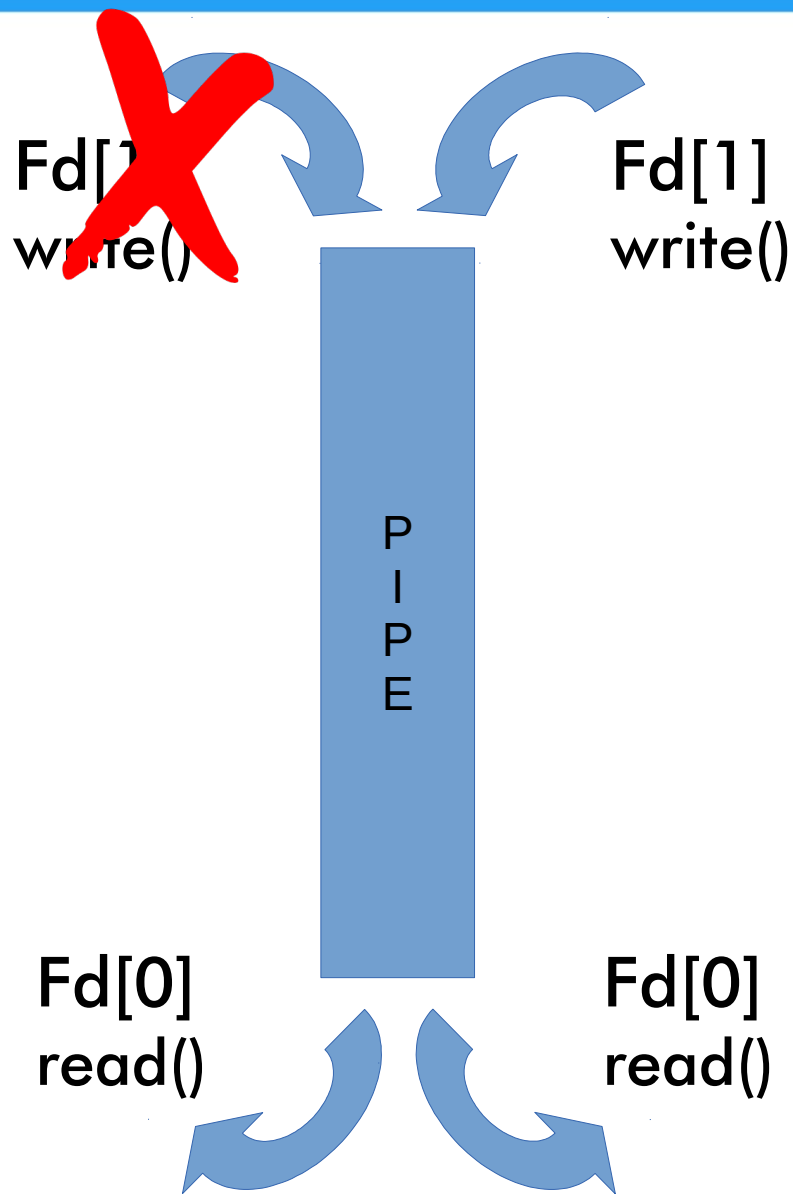
~~`Fd[1]  
write()`~~

`Fd[1]  
write()`

P  
I  
P  
E

`Fd[0]  
read()`

`Fd[0]  
read()`





**Padre**

`close(pfds[1])`

~~Fd[1]  
write()~~

Fd[0]  
read()

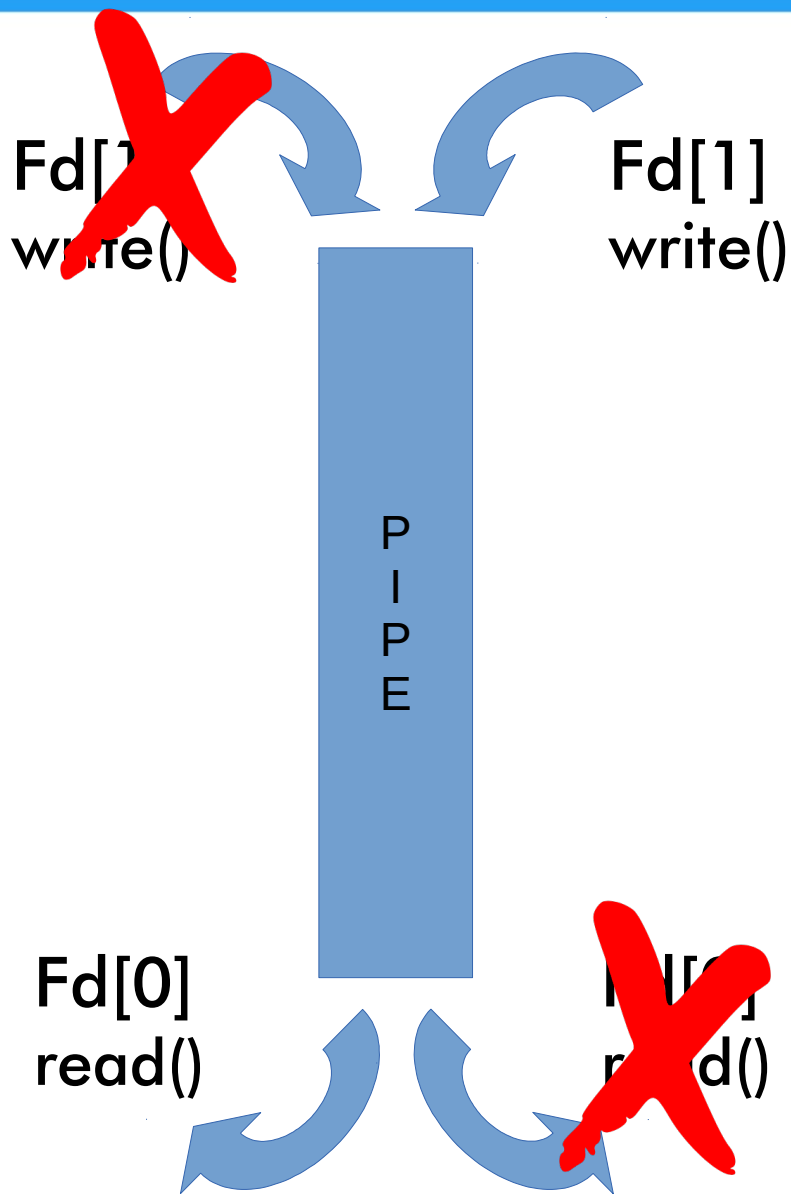
**Hijo**

`close(pfds[0])`

Fd[1]  
write()

~~Fd[0]  
read()~~

P  
I  
P  
E





## Padre

`close(pfds[1])`

~~Fd[1]  
write()~~

Fd[0]  
read()

## Hijo

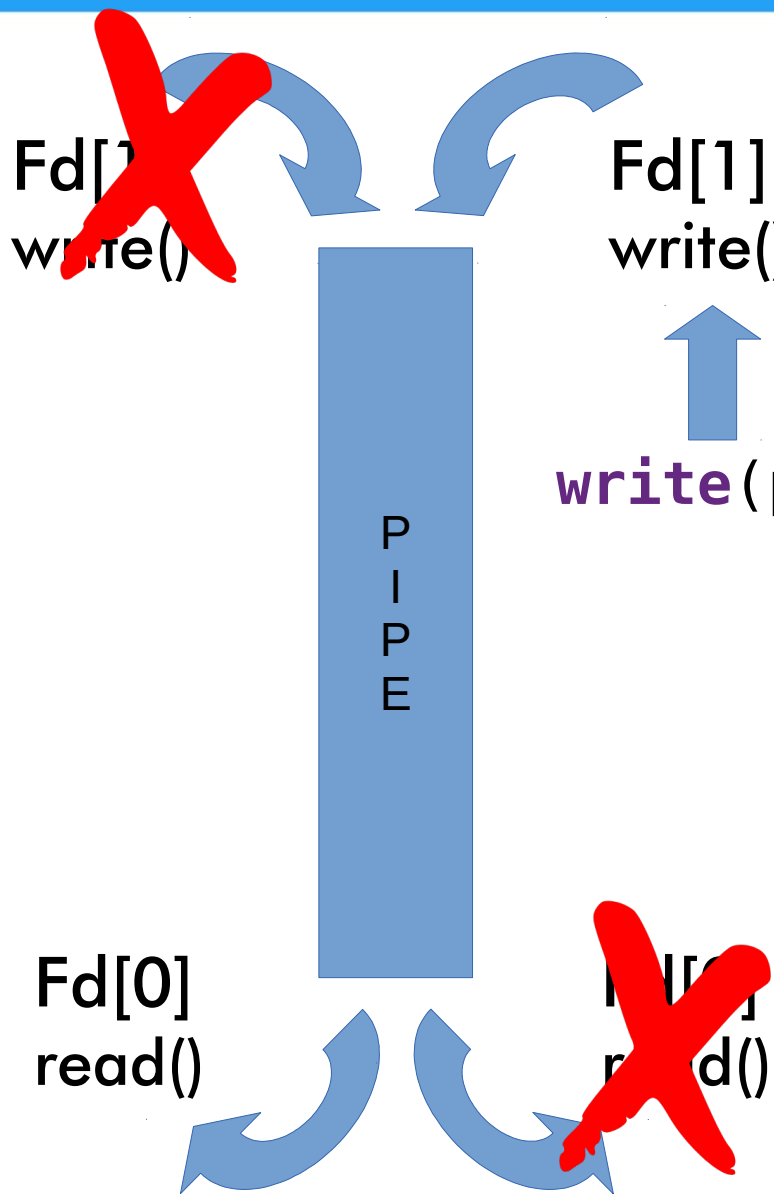
`close(pfds[0])`

Fd[1]  
write()

`write(pfds[1], "test", 5);`

~~Fd[0]  
read()~~

P  
I  
P  
E





## Padre

`close(pfds[1])`

`read(pfds[0], buf, 5);`

~~Fd[1]  
write()~~

Fd[0]  
read()

## Hijo

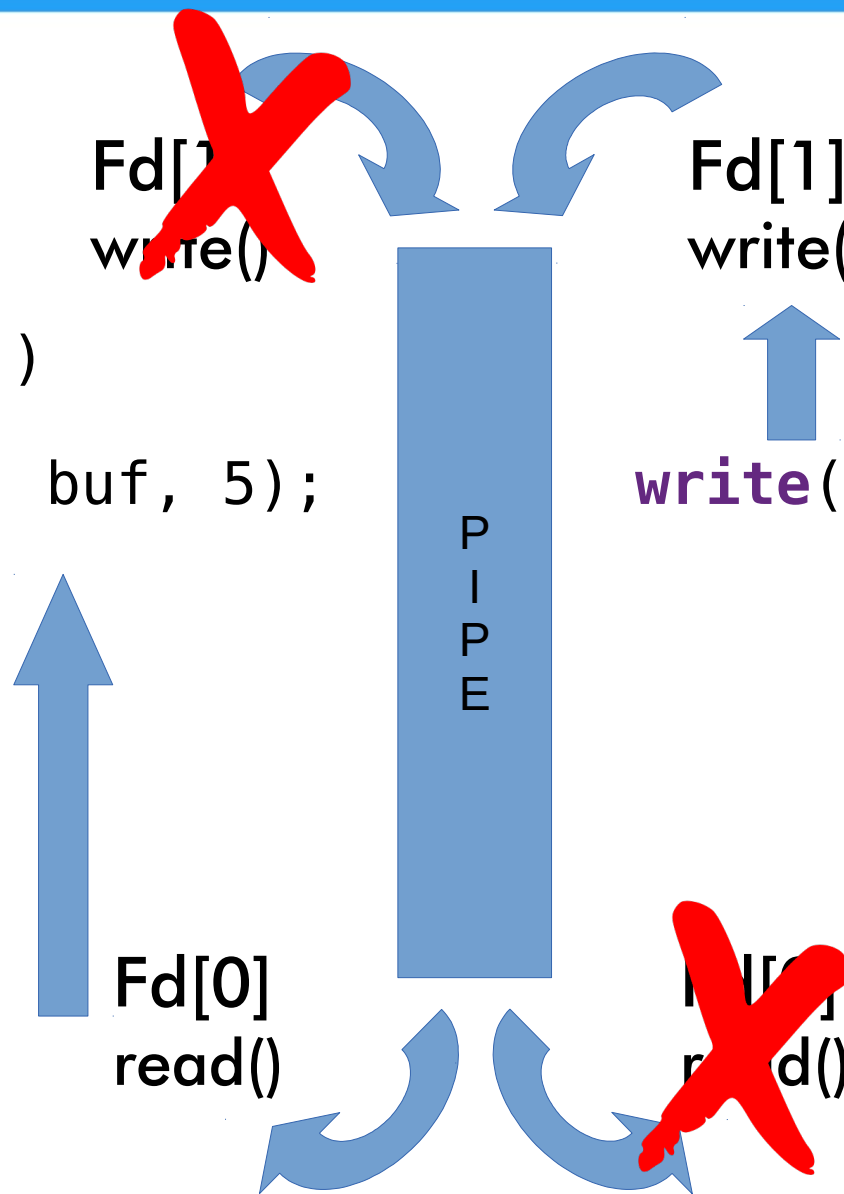
Fd[1]  
write()

`close(pfds[0])`

`write(pfds[1], "test", 5);`

~~Fd[0]  
read()~~

P  
I  
P  
E





```
int main(void)
{
```

```
    int pfd[2];
    char buf[30];
```

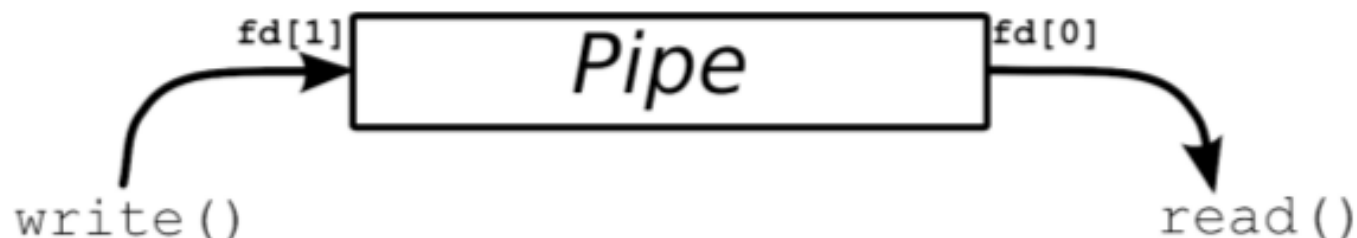
```
    pipe(pfd);
```

```
    if (!fork()) {
        printf(" CHILD: writing to the pipe\n");
```

```
        close(pfd[0]);
        write(pfd[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else {
```

```
        printf("PARENT: reading from pipe\n");
        close(pfd[1]);
        read(pfd[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }
```

## Pipes



How a pipe is organized.



## Padre

Fd[1]  
write()

**read**(pfds[0], buf, 5);

P  
I  
P  
E

Fd[0]  
read()

## Proceso Hijo finalizado

Si no cerramos el FD  
de escritura, la función  
read() se queda bloqueada  
Ya que es posible que  
se escriba en el pipe.



## Padre

Fd[1]  
write()

`close(pfds[1])`

`read(pfds[0], buf, 5);`

P  
I  
P  
E

Fd[0]  
read()

## Proceso Hijo finalizado

Si el FD de escritura estaba cerrado, al finalizar el hijo la función `read()` devuelve EOF (un cero).



## Padre

Debemos cerrar los FD que no se usan para que lleguen los EOF.

Fd[1]  
write()

Fd[0]  
read()

## Hijo

De lo contrario las funciones read y write nunca devolverán EOF.

Fd[1]  
write()

Fd[0]  
read()

P  
I  
P  
E





# **Pipes**

## **Práctica 1**



```
int main(void)
{
```

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

```
    if (!fork()) { // hijo
        //close(1); //cerramos stdout del hijo
        dup2(pfd[1],1); //reemplaza el stdout del hijo por el
                        //FD para write (y cierra el stdout)
        close(pfd[0]); // cerramos read
        execlp("ls", "ls", NULL); // reemplaza proceso hijo
                                // por "ls"
    }
    else { // padre
        //close(0); // cerramos stdin del padre
        dup2(pfd[0],0); // reemplaza el stdin del padre
                        // por el FD para read (y cierra el stdin)
        close(pfd[1]); // cerramos el FD para write
        execlp("wc", "wc", "-l", NULL); // reemplaza proceso
                                // padre por "wc -l"
    }
}
```

## Pipes

Ejecuta `ls | wc -l`  
con dos procesos



## **Bibliografía**

- Uresh Vahalia. (1996). UNIX Internals. The New Frontiers. New Jersey, Prentice Hall.
- Brian “Beej Jorgensen” Hall. (2015). Beej's Guide to Unix IPC.