Clase 3
Signals – Pipes

Autor: Esp. Ing. Ernesto Gigliotti. UTN-FRA

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 RT

- 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
SIGTSTP	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 **SIGTSTP** (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.

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

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;
```

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

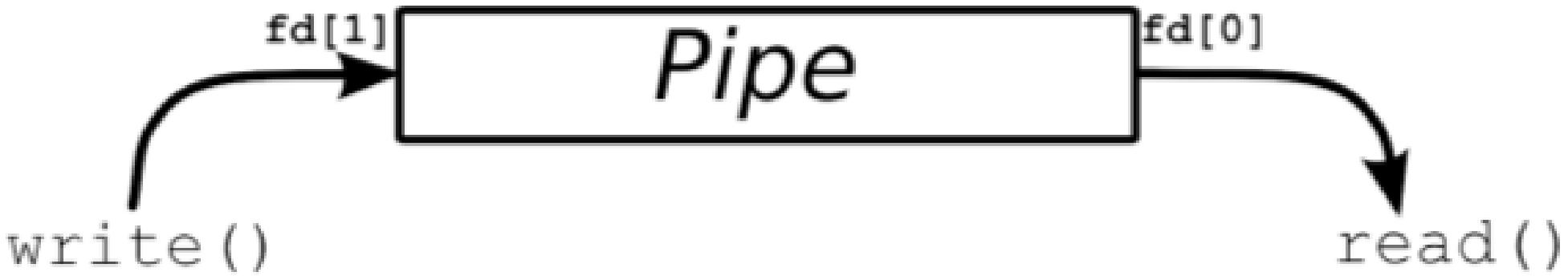
- Los procesos hijos “heredan” los File Descriptors del proceso padre. Por eso vemos los printf del proceso hijo en la consola que ejecuto el proceso padre.

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.

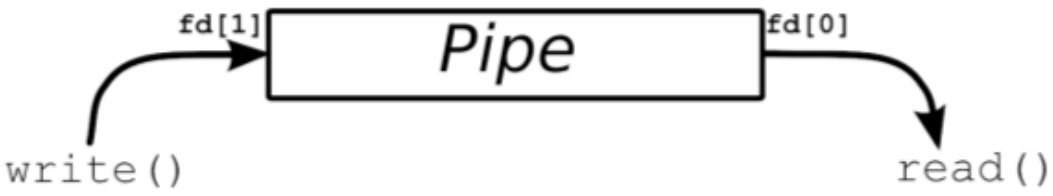
Pipes

```c
int main(void)
{
   int pfds[2];
   char buf[30];

   if (pipe(pfds) == -1) {
      perror("pipe");
      exit(1);
   }

   printf("writing to file descriptor #%d\n", pfds[1]);
   write(pfds[1], "test", 5);
   printf("reading from file descriptor #%d\n", pfds[0]);
   read(pfds[0], buf, 5);
   printf("read \"%s\"\n", buf);
   return 0;
```

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

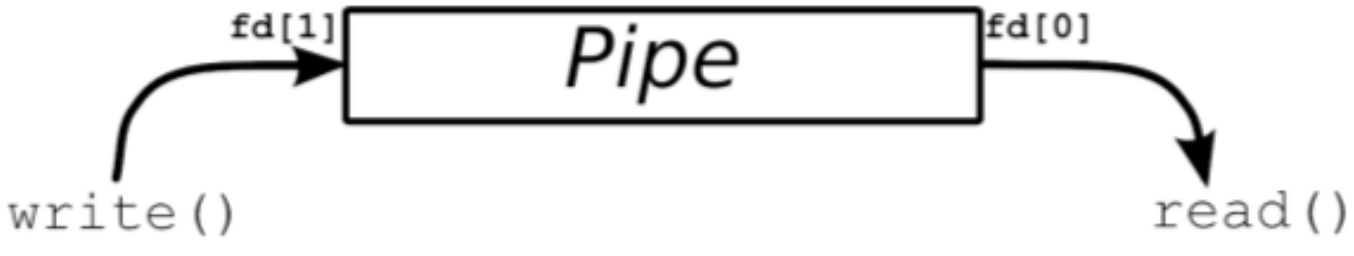


How a pipe is organized.

Pipes

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

Pipes



How a pipe is organized.

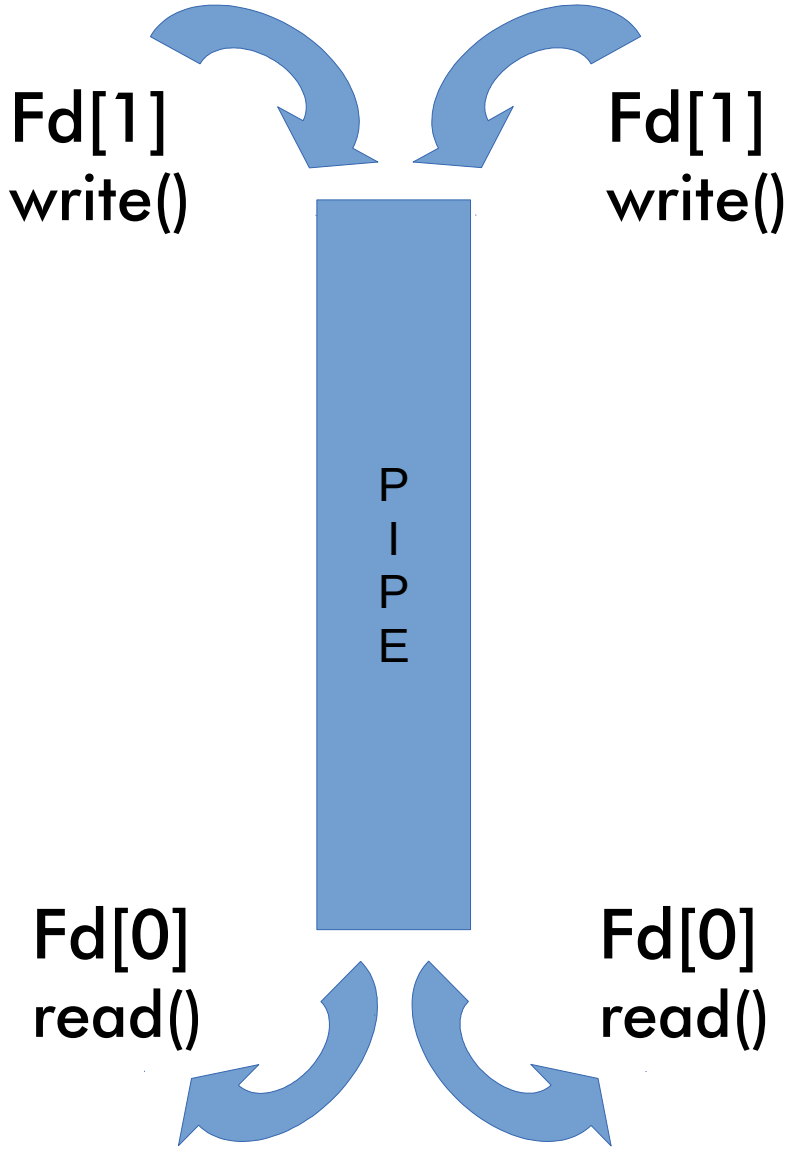
```c
int main(void)
{
   int pfds[2];
   char buf[30];

   pipe(pfds);

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

Padre

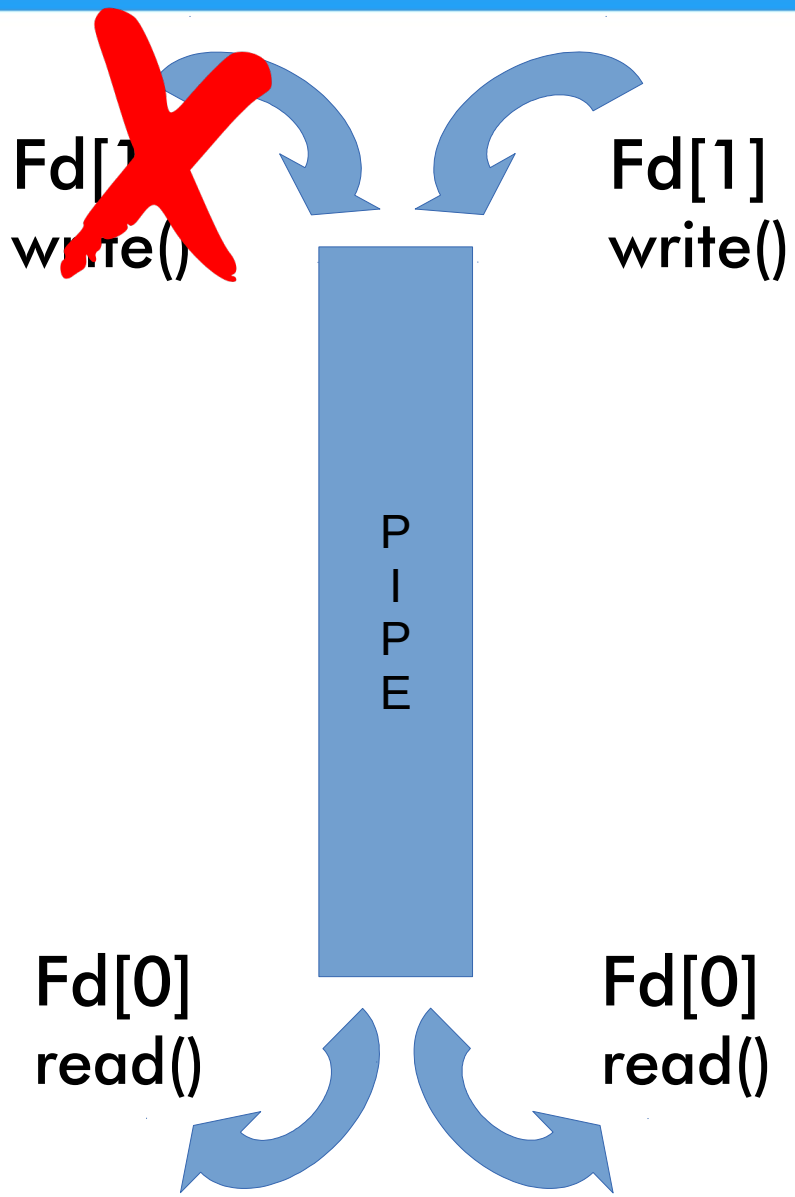
Hijo



Padre

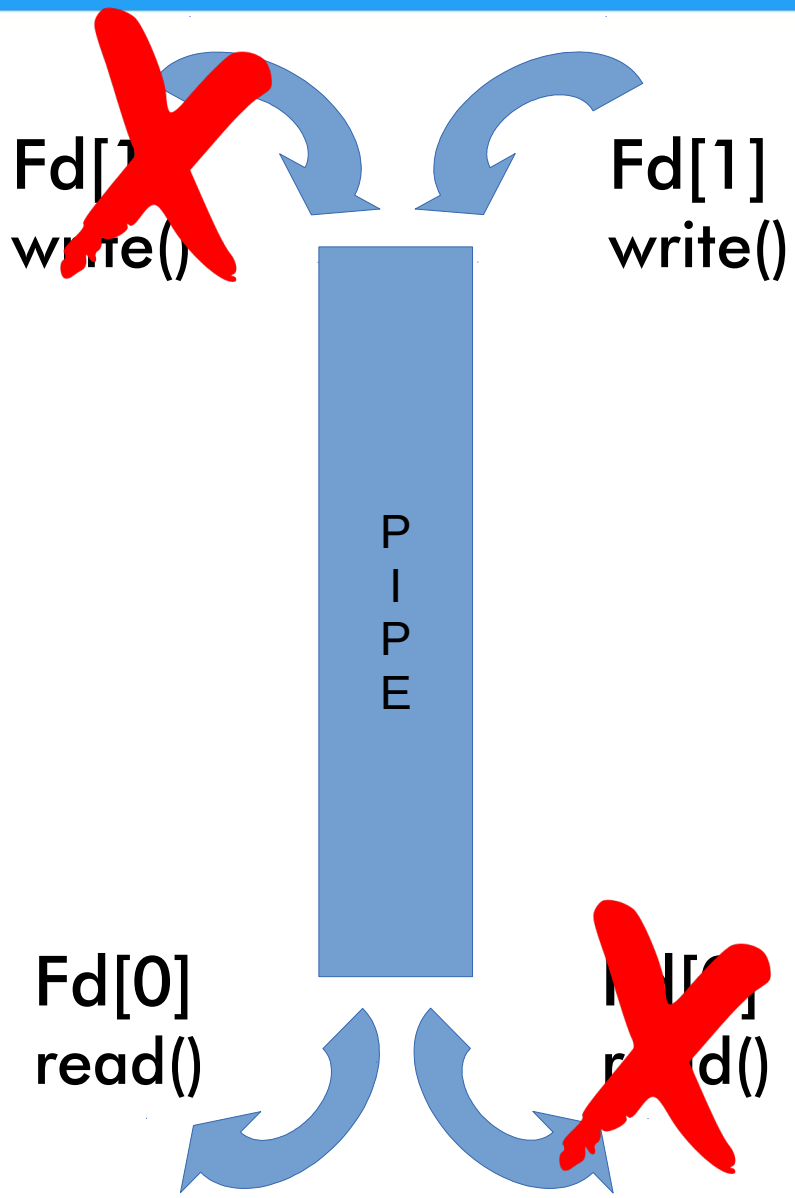
Hijo

```
close(pfds[1])
```



Padre

```
close(pfds[1])
```

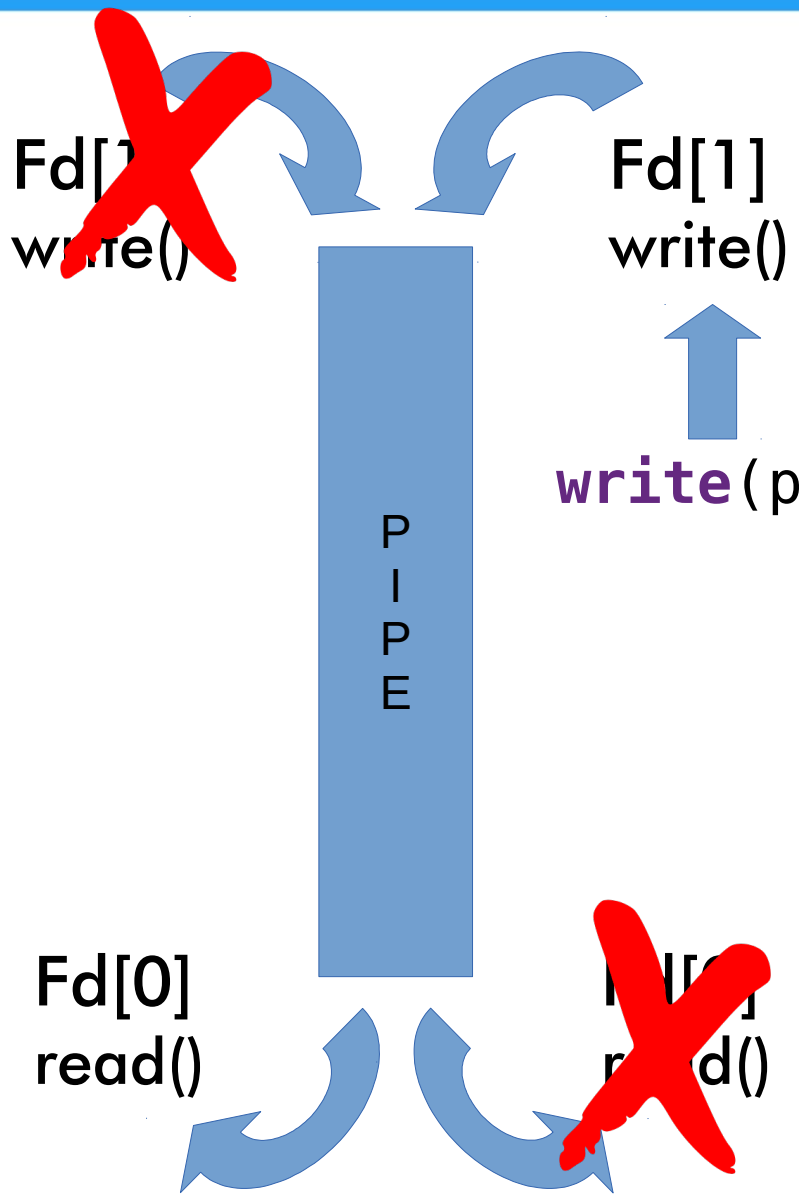


Hijo

```
close(pfds[0])
```

Padre

```
close(pfds[1])
```



Hijo

```
close(pfds[0])

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

Padre

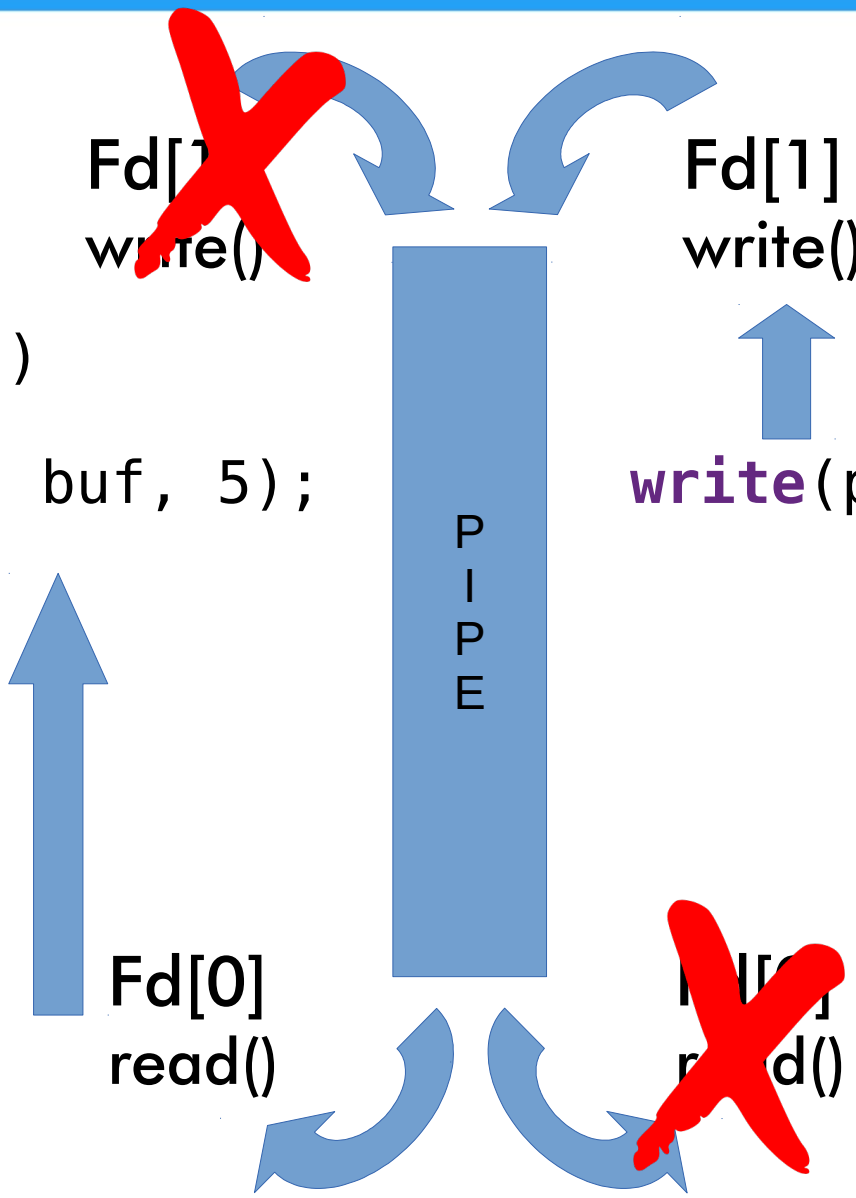
```
close(pfds[1])

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

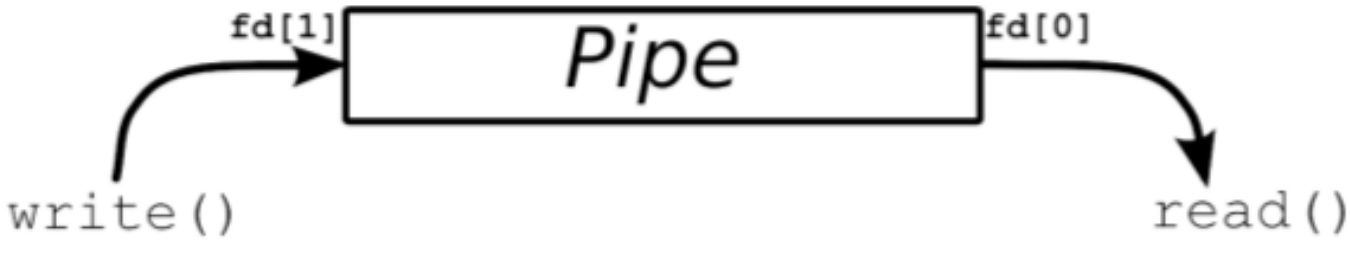
Hijo

```
close(pfds[0])

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



Pipes



How a pipe is organized.

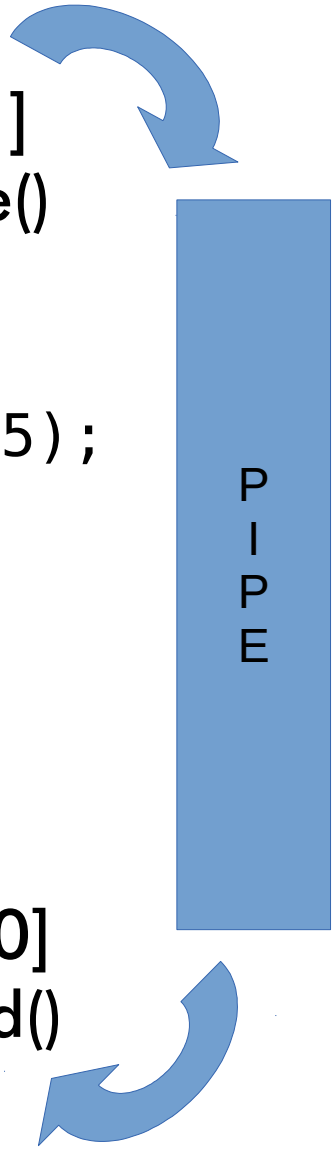
```c
int main(void)
{
   int pfds[2];
   char buf[30];

   pipe(pfds);

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

Padre

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



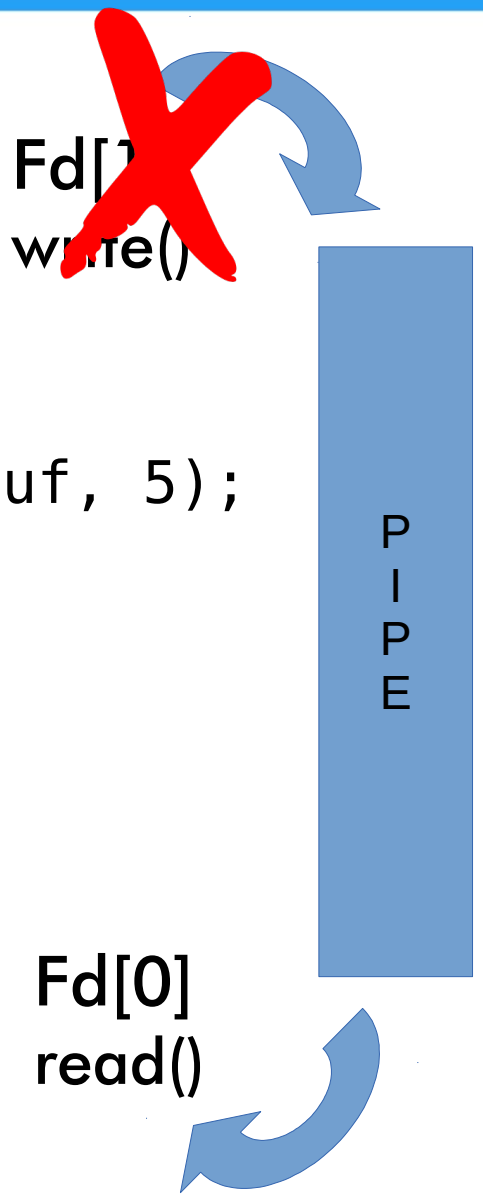
Proceso Hijo finalizado

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

Padre

```
close(pfds[1])

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

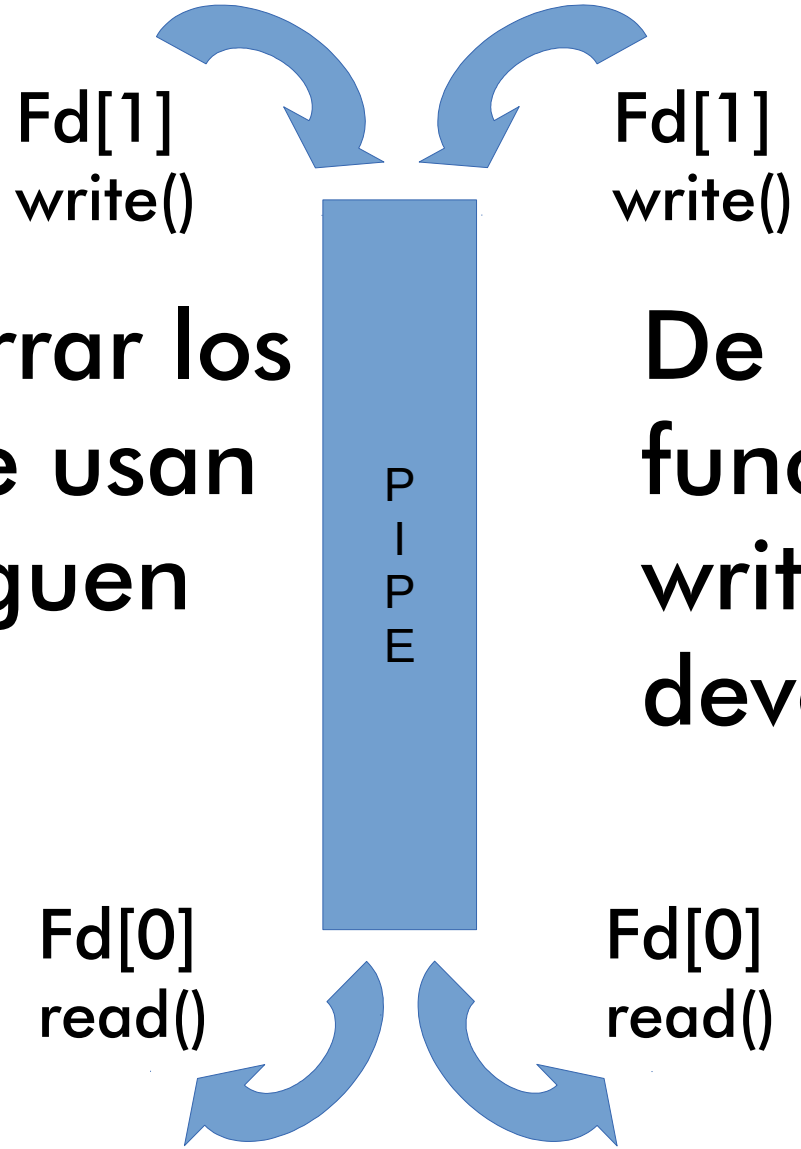


Proceso Hijo finalizado

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

Padre

Hijo



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

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

Pipes

Práctica 1

Pipes

Ejecuta> `ls | wc -l` con dos procesos

```
int main(void)
{
   int pfds[2];
   pipe(pfds);

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

Bibliografía

- Lewis Van Winkle. (2019). Hands-On Network Programming with C,Packt.
- Chris Simmonds. (2017). Mastering embedded linux programming 2nd edition, Packt.
- Uresh Vahalia. (1996). UNIX Internals. The New Frontiers. New Jersey, Prentice Hall.
- Brian “Beej Jorgensen” Hall. (2015). Beej's Guide to Unix IPC.