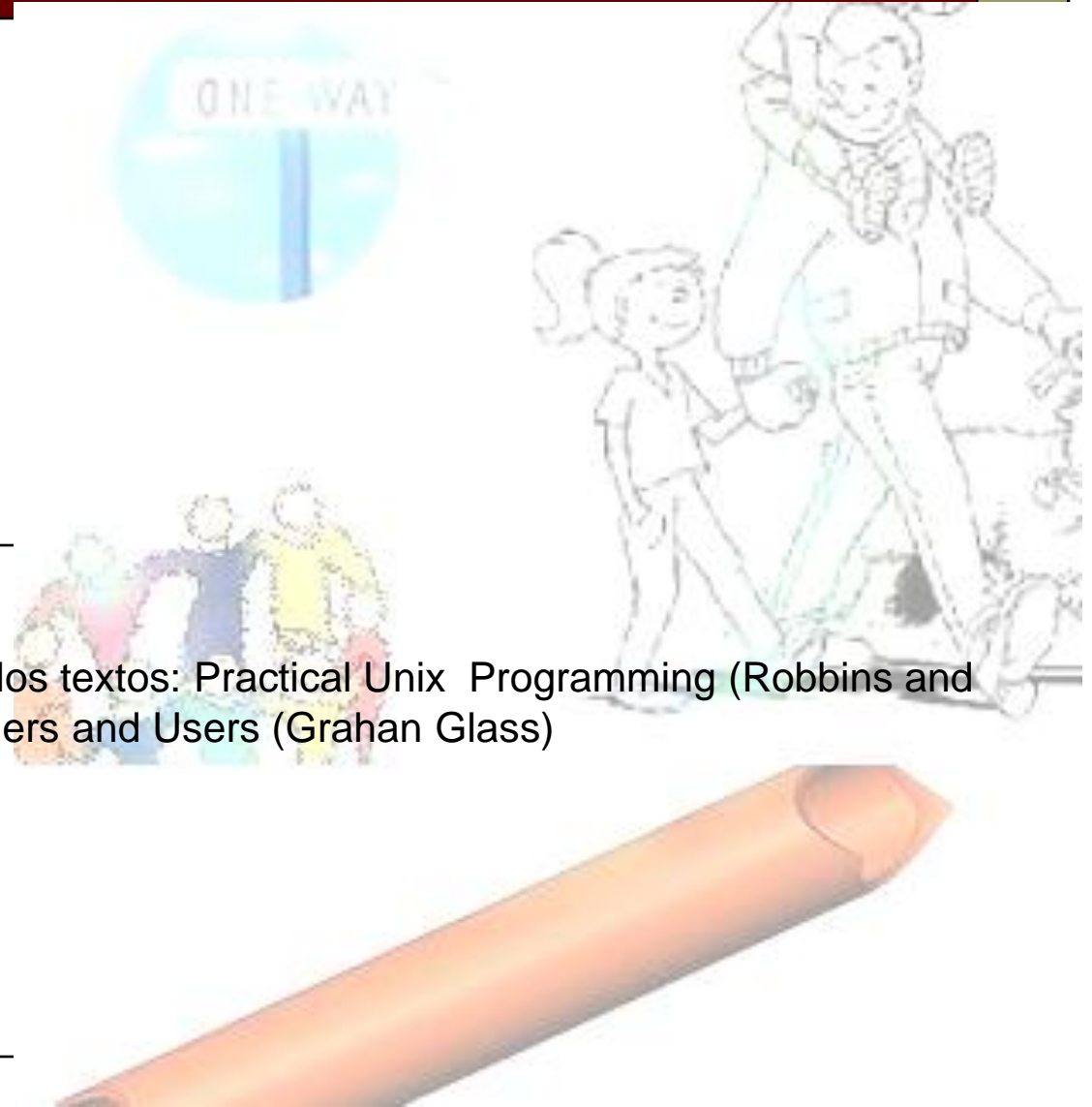


Pipes

M. Curiel

Ejemplos fueron tomados de los textos: Practical Unix Programming (Robbins and Robins) y Unix for Programmers and Users (Graham Glass)



Contenido

- ❑ Tablas del kernel para el manejo de archivos
- ❑ Llamada al sistema *dup*, redirección de la entrada y salida estándar.
- ❑ Pipes **no Nominales**
- ❑ Pipes **Nominales**

Tablas del SOP: Descriptores de Archivo

| | |
|---|--------------------------|
| 0 | Standard Input |
| 1 | Standard Output |
| 2 | Standard Error |
| 3 | Primer Descriptor Libre |
| 4 | Segundo Descriptor Libre |
| | |

```
int fd;  
mode_t fd_mode = S_IRUSR | S_IWUSR | S_IRGRP  
if ((fd = open("/home/perez/archivo.c", O_RDWR | O_CREAT, fd_mode) ) == -1)  
    perror("No se pudo abrir el archivo ("/home/perez/archivo.c:");
```

Tablas del Sistema Operativo

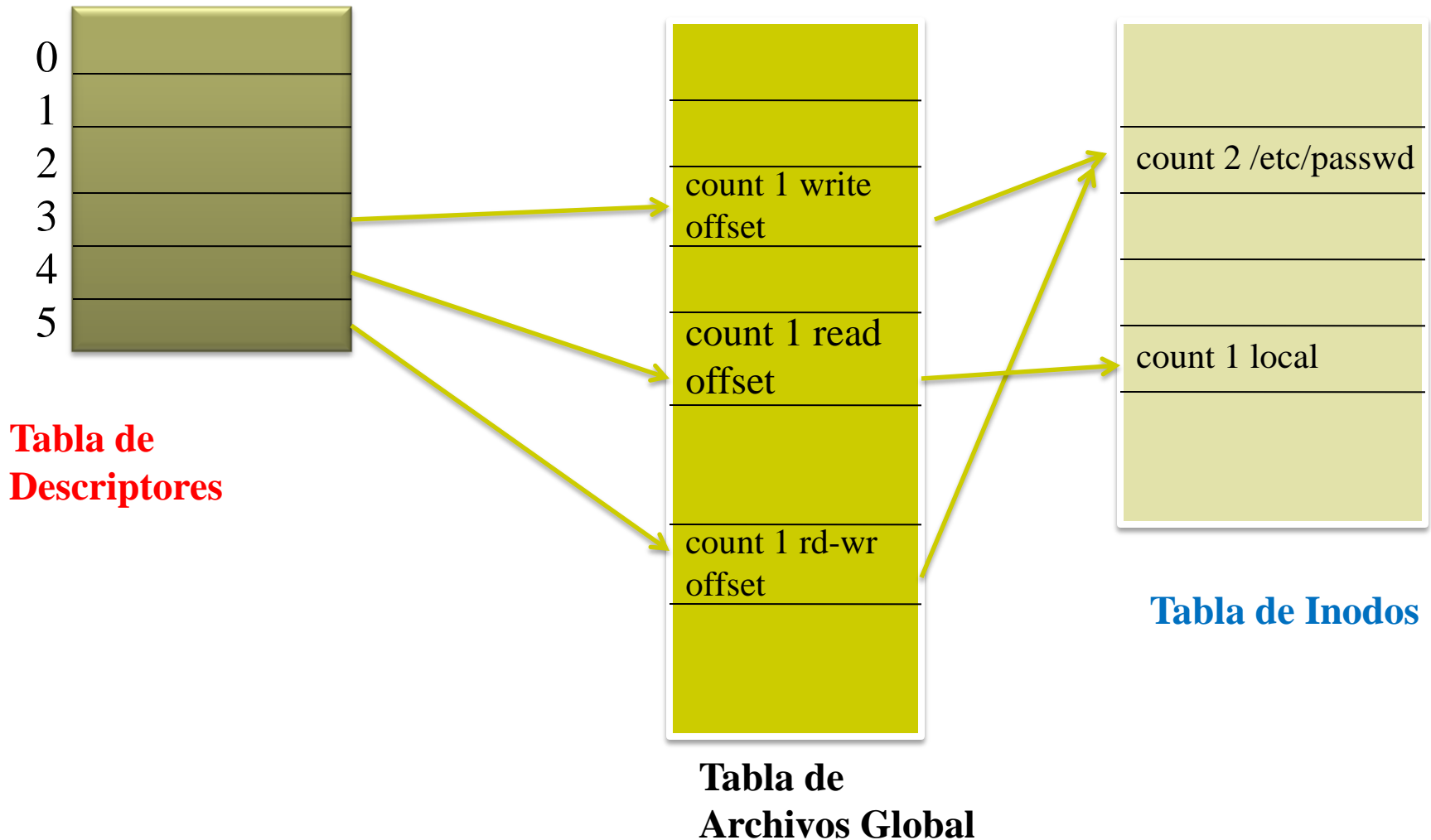
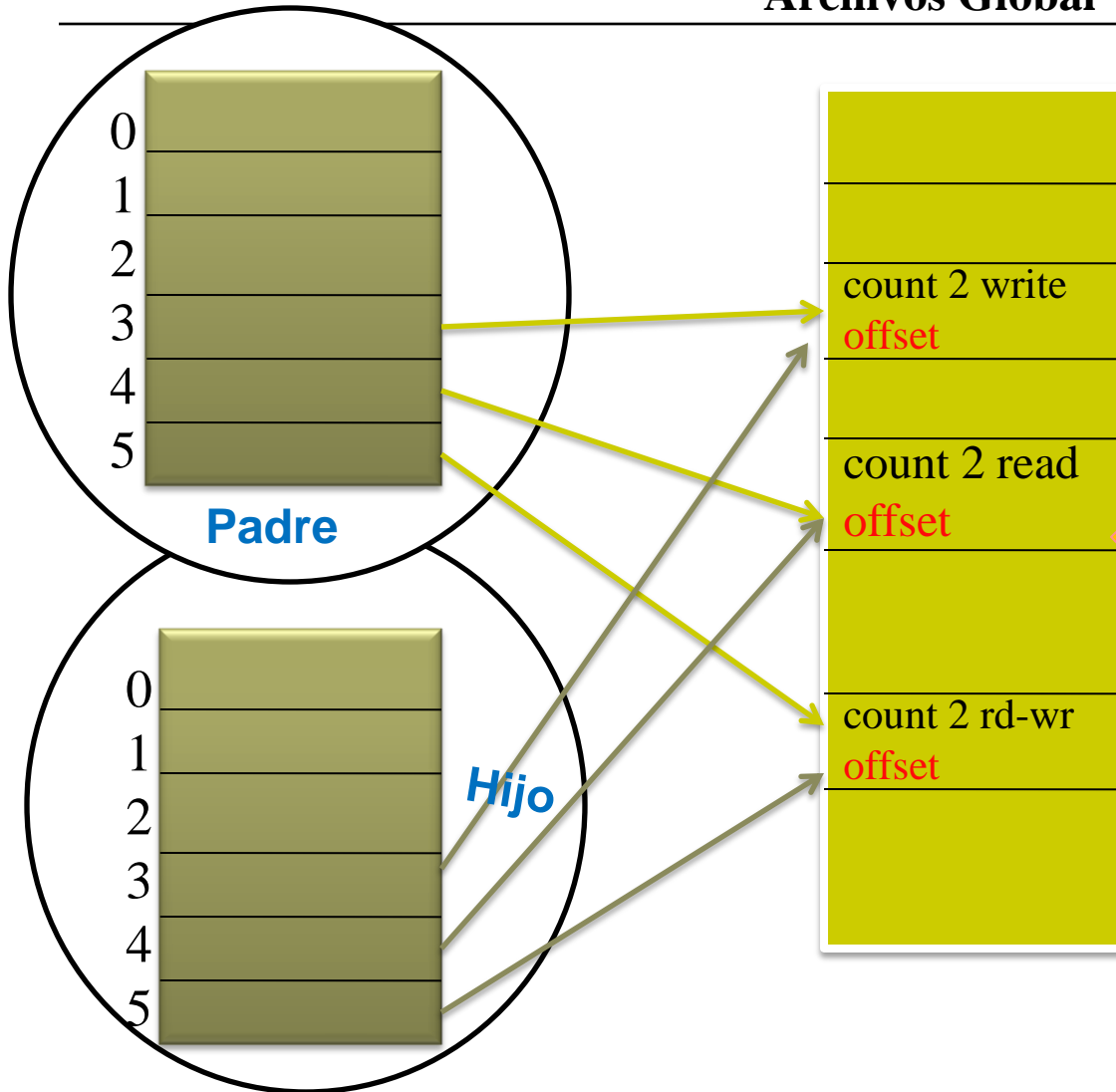


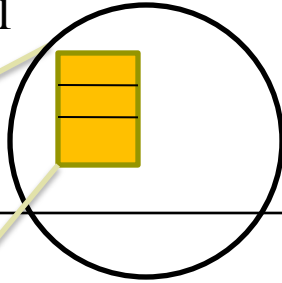
Tabla de Descriptores

Tabla de Archivos Global



Despues de un fork, se duplica la tabla de descriptores, Padres e hijos comparten el mismo offset.

P1

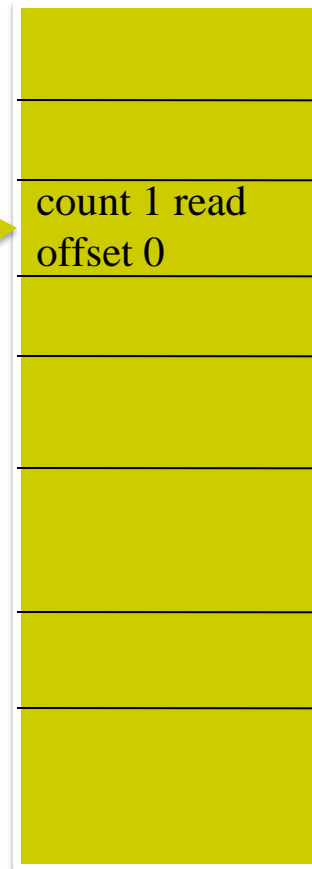


P1 ejecuta:

```
fd1= open("/etc/passwd", O_RDONLY);
```



**Tabla de
descriptores
de P1**



**Tabla de
Archivos Global**

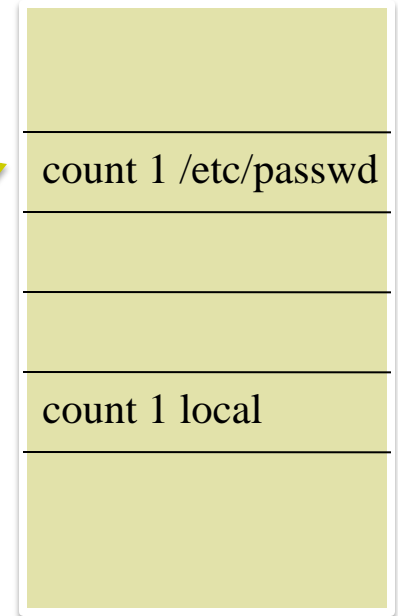


Tabla de Inodos

P1 ejecuta:

```
fd1= open("/etc/passwd", O_RDONLY);  
fd2 =open("local", O_WRONLY);
```

| | |
|---|--|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

**Tabla de
Descriptores de P1**

| |
|---------------------------|
| |
| |
| count 1 read offset 0 |
| |
| count 1 write offset 0 |
| |
| |
| |
| |

**Tabla de
Archivos Global**

| |
|---------------------|
| |
| |
| count 1 /etc/passwd |
| |
| |
| count 1 local |
| |

Tabla de Inodos

```
fd1= open("/etc/passwd", O_RDONLY);
fd2 =open("local", O_WRONLY);
fd3 =open("/etc/passwd", O_RDWR)
```

| | |
|---|--|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

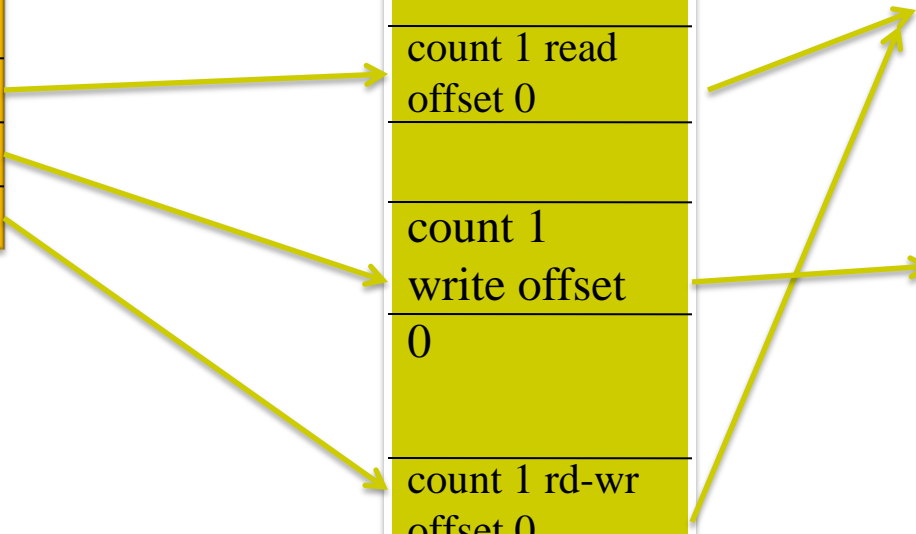
Tabla de Descriptores de P1

| |
|------------------------------|
| |
| |
| count 1 read offset 0 |
| |
| count 1 write offset 0 |
| |
| count 1 rd-wr offset 0 |
| |

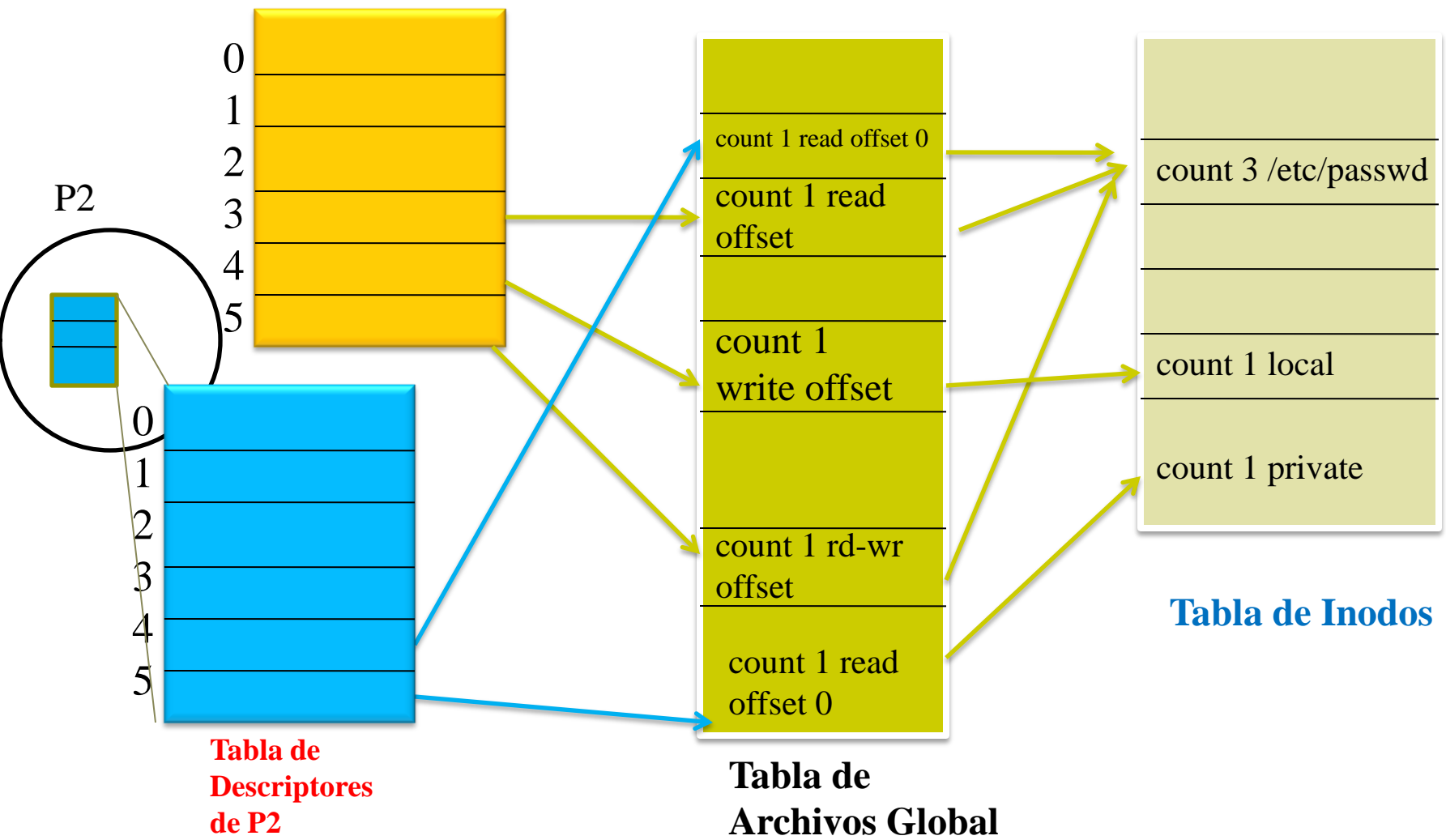
Tabla de Archivos Global

| |
|---------------------|
| |
| |
| count 2 /etc/passwd |
| |
| |
| count 1 local |
| |

Tabla de Inodos



```
P2 ejecuta:  
fd1= open(“/etc/passwd”, O_RDONLY);  
fd2 =open(“private”, O_RDONLY);
```



Dup

int dup(fd)

int dup2(int oldfd, int newfd)


- **dup(fd)** encuentra la primera entrada libre en la tabla de descriptores y la pone “a apuntar” al mismo lugar al que apunta **fd**, i.e a la misma entrada en la Tabla de Archivos Global (*File Table*)
- **dup2(int oldfd, int newfd)**: cierra **newfd** si está activo y coloca esta entrada a apuntar donde apunta **oldfd** (a la misma entrada en la Tabla de Archivos Global). Es equivalente a:

close(newfd)

dup(oldfd)

Ejemplo del uso del Dup2

```
write(STDOUT_FILENO, buff, 20)
fd = open("myfile", O_WRONLY|O_CREAT, 066)
dup2(fd, STDOUT_FILENO)
close(fd)
write(STDOUT_FILENO, buff, 20)
```

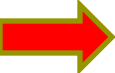


| | |
|---|--------|
| 0 | stdin |
| 1 | stdout |
| 2 | stderr |
| 3 | myfile |
| 4 | |
| 5 | |

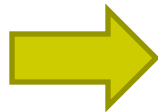
Tabla de descriptores del proceso
después del open

Ejemplo del uso del Dup2

```
write(STDOUT_FILENO, buff, 20)
fd = open("myfile", O_WRONLY|O_CREAT, 066)
dup2(fd, STDOUT_FILENO)
close(fd)
write(STDOUT_FILENO, buff, 20)
```



| | |
|---|--------|
| 0 | stdin |
| 1 | stdout |
| 2 | stderr |
| 3 | myfile |
| 4 | |
| 5 | |

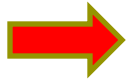


| | |
|---|--------|
| 0 | stdin |
| 1 | myfile |
| 2 | stderr |
| 3 | myfile |
| 4 | |
| 5 | |

Tabla de
descriptores del
proceso después
del dup2

Ejemplo del uso del Dup2

```
write(STDOUT_FILENO, buff, 20)
fd = open("myfile", O_WRONLY|O_CREAT, 066)
dup2(fd, STDOUT_FILENO)
close(fd)
write(STDOUT_FILENO, buff, 20)
```



| | |
|---|--------|
| 0 | stdin |
| 1 | stdout |
| 2 | stderr |
| 3 | myfile |
| 4 | |
| 5 | |



| | |
|---|--------|
| 0 | stdin |
| 1 | myfile |
| 2 | stderr |
| 3 | myfile |
| 4 | |
| 5 | |



| | |
|---|--------|
| 0 | stdin |
| 1 | myfile |
| 2 | stderr |
| 3 | |
| 4 | |
| 5 | |

Después del **dup2**
la salida estándar
se va al archivo myfile

Ejemplo: Redirección de Salida Estándar

\$ ls > ls.out

¿Cómo se redirecciona la salida de un comando a un archivo?

Ejemplo: Redirección de Salida Estándar

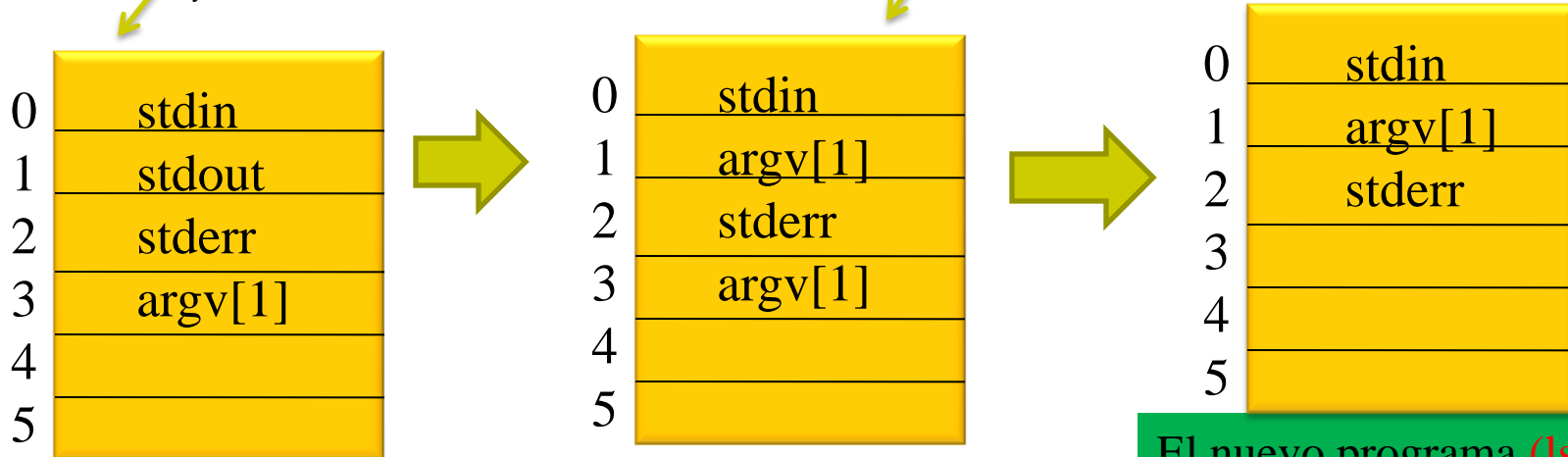
\$ Redirect ls.out ls -l

El program **Redirect**, redirecciona la salida del comando que recibe como segundo argumento, al archivo que recibe como primer argumento,.

```
main(int argc, char **argv) {
    int fd;
    fd = open(argv[1], O_CREAT | O_TRUNC | O_WRONLY, 0600);
    dup2(fd, STDOUT_FILENO);
    close(fd);
    execvp(argv[2], &argv[2]);
    perror("main");
}
```

Ejemplo del uso del Dup2

```
main(int argc, char **argv) {  
    int fd;  
    fd = open(argv[1], O_CREAT | O_TRUNC | O_WRONLY, 0600);  
    dup2(fd, STDOUT_FILENO);  
    close(fd);  
    execvp(argv[2], &argv[2]);  
    perror("main");  
}
```



El nuevo programa (`ls`) tiene esta misma tabla de descriptors.

Pipes

```
$ ls *.c | wc -l
```

¿Cómo se implementa?

El comando `ls` lista los archivos con extensión `.c`, y su salida va hacia el comando `wc` que cuando se invoca con la opción `-l` muestra el total de líneas en la entrada.

Así, la orden completa calcula el total de archivos con extensión `.c` que hay en el directorio actual.

Pipes

```
$ ls *.c | wc -l
```

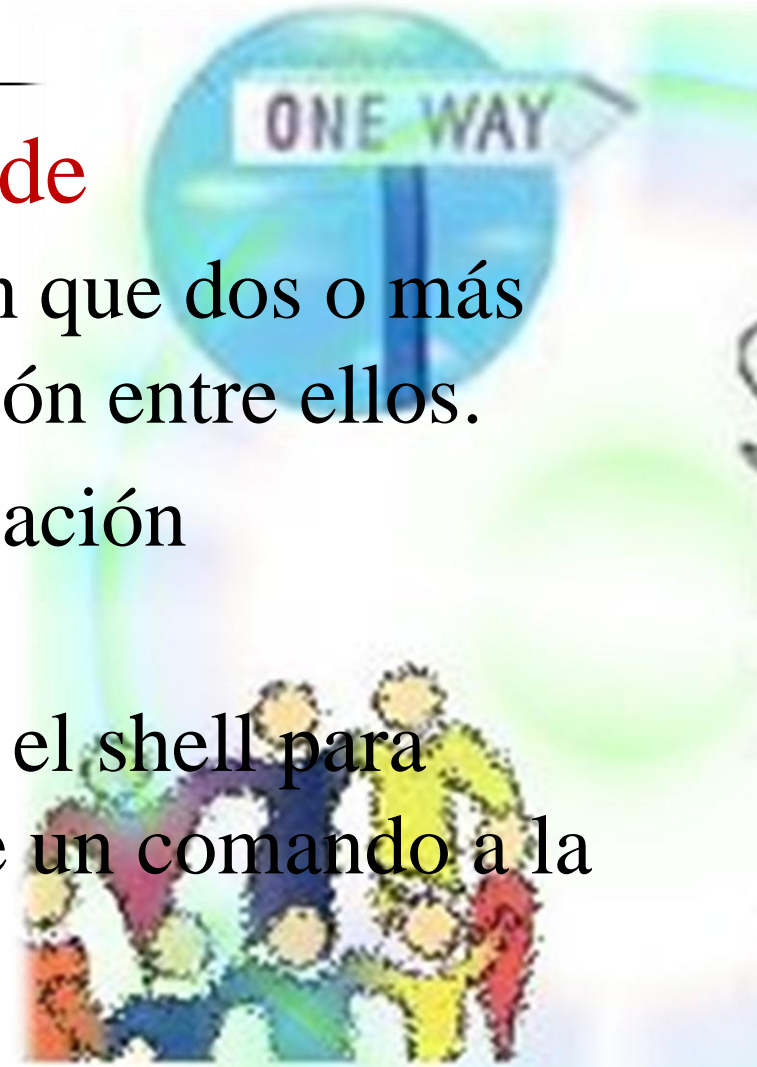
¿Cómo se implementa?

El comando `ls` lista los archivos con extensión `.c`, y su salida va hacia el comando `wc` que cuando se invoca con la opción `-l` muestra el total de líneas en la entrada.

Así, la orden completa calcula el total de archivos con extensión `.c` que hay en el directorio actual.

Definición

- ❑ Los **pipes** son **mecanismos de comunicación** que permiten que dos o más procesos se envíen información entre ellos.
- ❑ Son mecanismos de comunicación **unidireccionales**.
- ❑ Son usados comúnmente por el shell para conectar la salida estándar de un comando a la entrada estándar de otro.
- ❑ `$ ls *.c | wc -l`



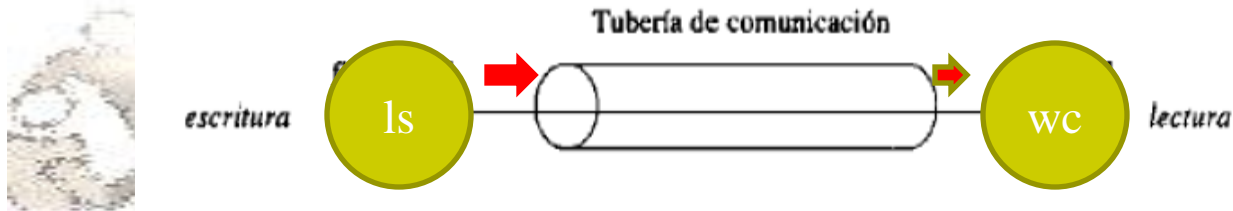
Pipes no Nominales

- Hay dos tipos de pipes: **nominales** (*named pipes*) y **no nominales** (*unnamed pipes*).
- Los pipes **no nominales** se crean usualmente **para la comunicación entre procesos padres e hijos**; uno de los procesos lee y el otro escribe.

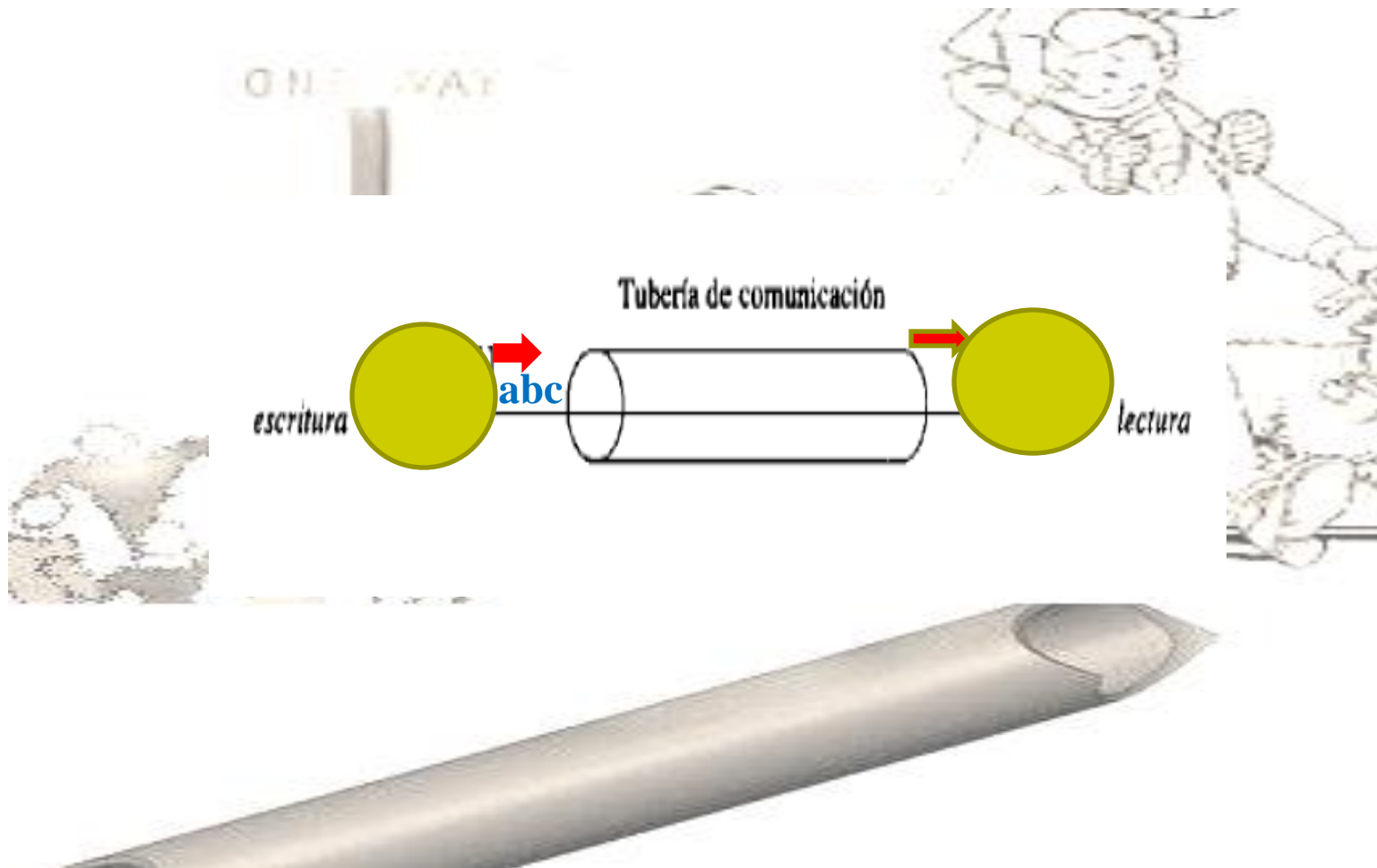


Pipes

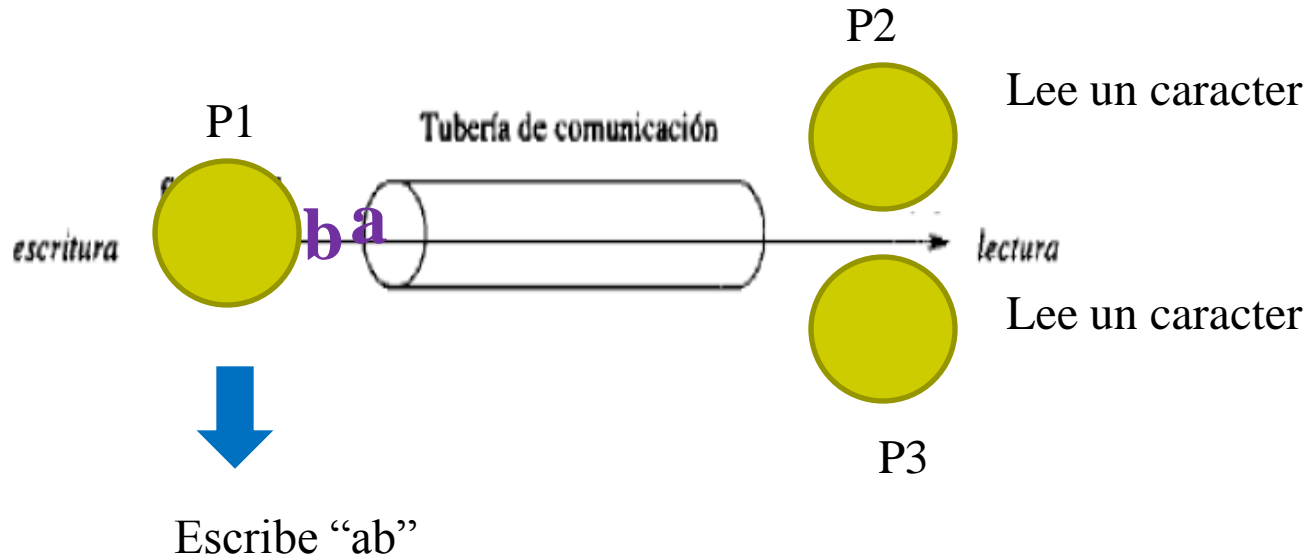
- El pipe tiene un extremo para lectura y otro para escritura.
- Tanto el lector como el escritor de un pipe ejecutan en forma concurrente.



Pipes



Asociados a cada extremo del pipe puede haber uno o más lectores o escritores.



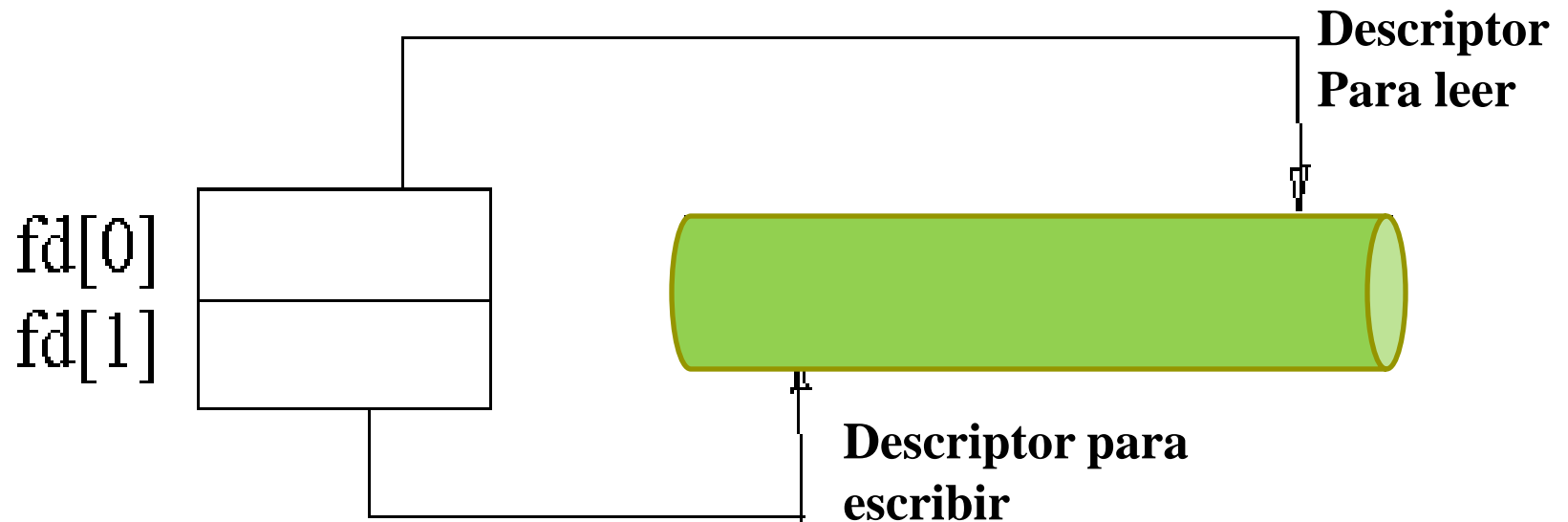
Pipes no Nominales

- Un pipe no nominal es un enlace de comunicación unidireccional que puede crearse usando la llamada al sistema *pipe*. Cada extremo del pipe tiene un descriptor de archivo asociado.
- *pipe()* crea un pipe no nominal y retorna dos descriptors de archivo. El descriptor asociado con el extremo del pipe por el que se va a leer se almacena en `fd[0]` y el descriptor asociado al extremo del pipe por donde se va a escribir se almacena en `fd[1]`.
-

Pipes no nominales: Creación

```
int fd[2];
```

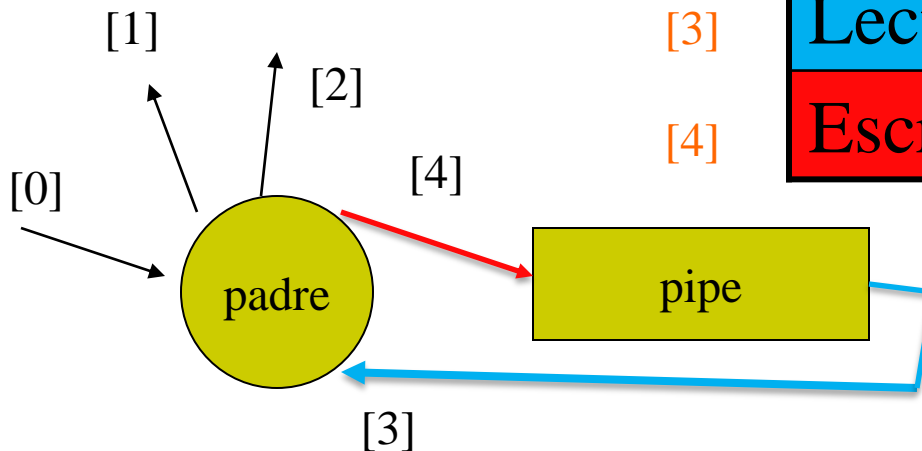
```
→ pipe(fd);
```



Pipes no Nominales: Ejemplo 1

```
main (int argc, char *argv[]) {
    int fd[2], countbytes;
    char message[100], frase = "abc";
    pipe(fd);
    if (fork()==0) { /* codigo del hijo */
        close(fd[0]); /* Se cierran el descriptor que no se usa */
        write(fd[1], frase, strlen(frase) + 1);
        close(fd[1]);
    } else { /* codigo del padre */
        close(fd[1]); /* Se cierran los descriptors que no se usan */
        countbytes = read(fd[0], message, 100);
        printf("Mensaje leído %s", message);
        close(fd[0]);
    }
}
```

Tabla de Descriptores del Proceso



[0]

Entrada estándar

[1]

Salida estándar

[2]

Error estándar

[3]

Lectura del pipe

[4]

Escritura al Pipe

Ejemplo 1

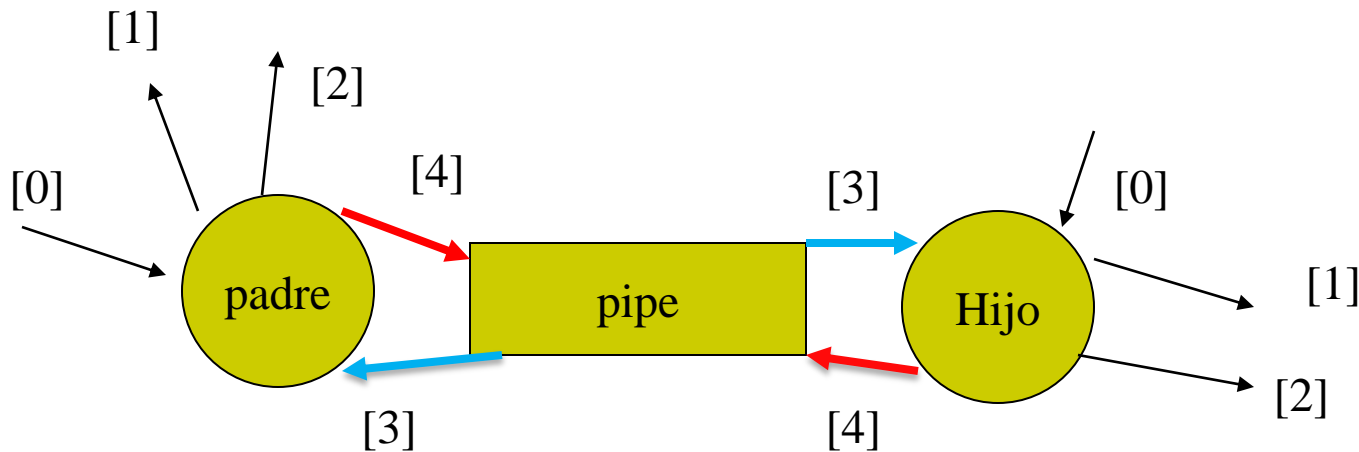
```
main (int argc, char *argv[]) {  
    int fd[2], countbytes;  
    char message[100], frase =“Hola soy el hijo”;  
    pipe(fd);  
    if (fork()==0) { /* codigo del hijo */  
        } else { /* codigo del padre */  
    }  
}
```



TD del Padre

TD del Hijo

| | | |
|-----|-------------------|-------------------|
| [0] | Entrada estándar | Entrada estándar |
| [1] | Salida estándar | Salida estándar |
| [2] | Error estándar | Error estándar |
| [3] | Lectura del pipe | Lectura del pipe |
| [4] | Escritura al Pipe | Escritura al pipe |



Tablas de
descriptores
(TD)
después del
fork()

Pipes No Nominales

- Se puede escribir al pipe y leer de él usando las llamadas al sistema *write()* y *read()*, respectivamente.
- Cuando un proceso termina de usar el pipe, debe cerrarlo (*close()*).
- Los extremos del pipe que no se utilizan deben cerrarse también.

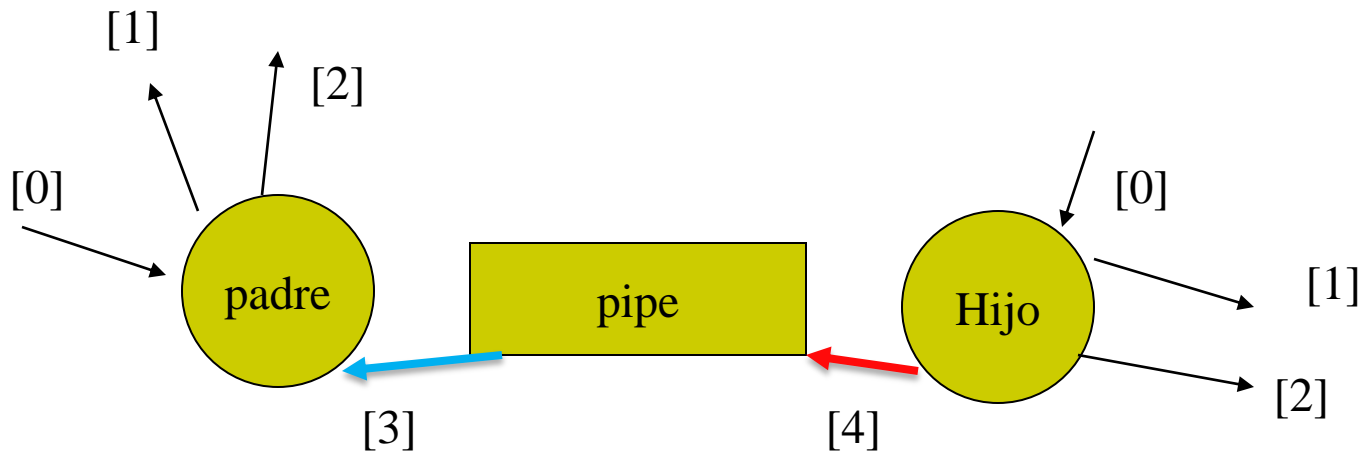
Ejemplo 1

```
main (int argc, char *argv[]) {
    int fd[2], countbytes;
    char message[100], frase = "abc";
    pipe(fd);
    if (fork()==0) { /* codigo del hijo */
        close(fd[0]); /* Se cierran el descriptor que no se usa */
        write(fd[1], frase, strlen(frase) + 1);
        close(fd[1]);
    } else { /* codigo del padre */
        close(fd[1]); /* Se cierran los descriptors que no se usan */
        countbytes = read(fd[0], message, 100);
        printf("Mensaje leído %s", message);
        close(fd[0]);
    }
}
```

TD del Padre

TD del Hijo

| | | |
|-----|------------------|-------------------|
| [0] | Entrada estándar | Entrada estándar |
| [1] | Salida estándar | Salida estándar |
| [2] | Error estándar | Error estándar |
| [3] | Lectura del pipe | |
| [4] | | Escritura al pipe |

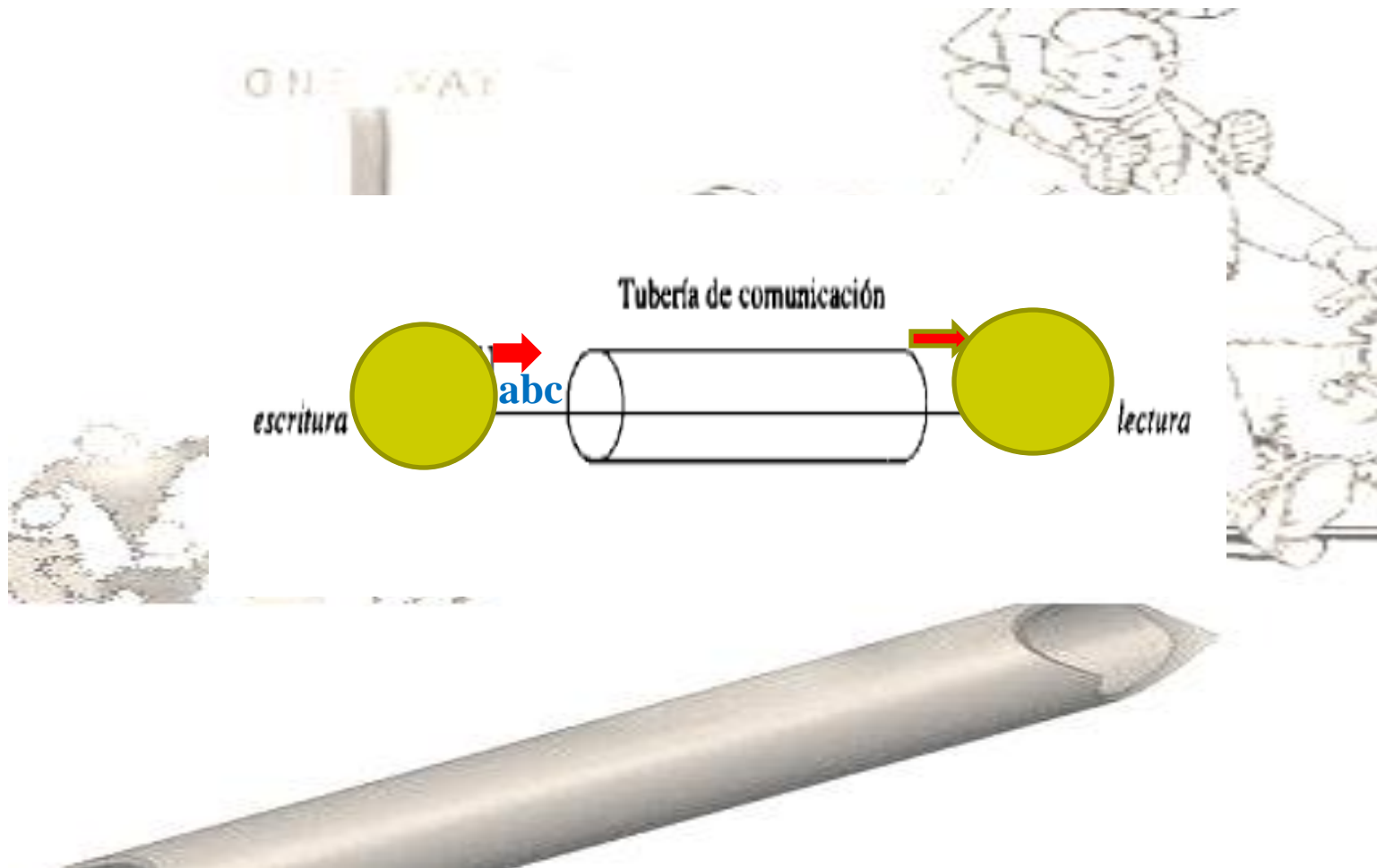


Se cierran
descriptores
no utilizados

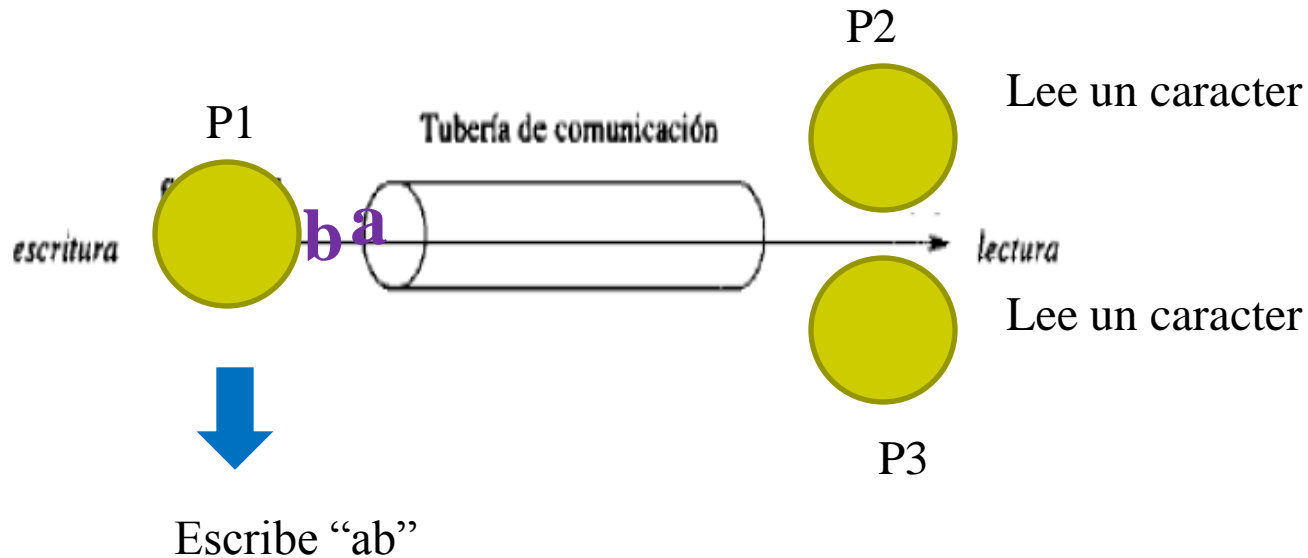
Ejemplo 1

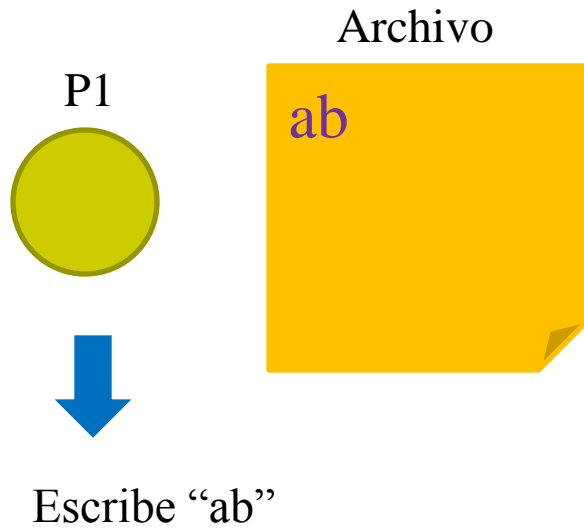
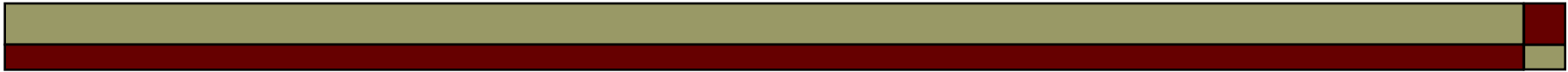
```
main (int argc, char *argv[]) {
    int fd[2], countbytes;
    char message[100], frase = "abc";
    pipe(fd);
    if (fork()==0) { /* codigo del hijo */
        close(fd[0]); /* Se cierran el descriptor que no se usa */
        write(fd[1], frase, strlen(frase) + 1);
        close(fd[1]);
    } else { /* codigo del padre */
        close(fd[1]); /* Se cierran los descriptors que no se usan */
        countbytes = read(fd[0], message, 100);
        printf("Mensaje leido %s", message);
        close(fd[0]);
    }
}
```

Pipes

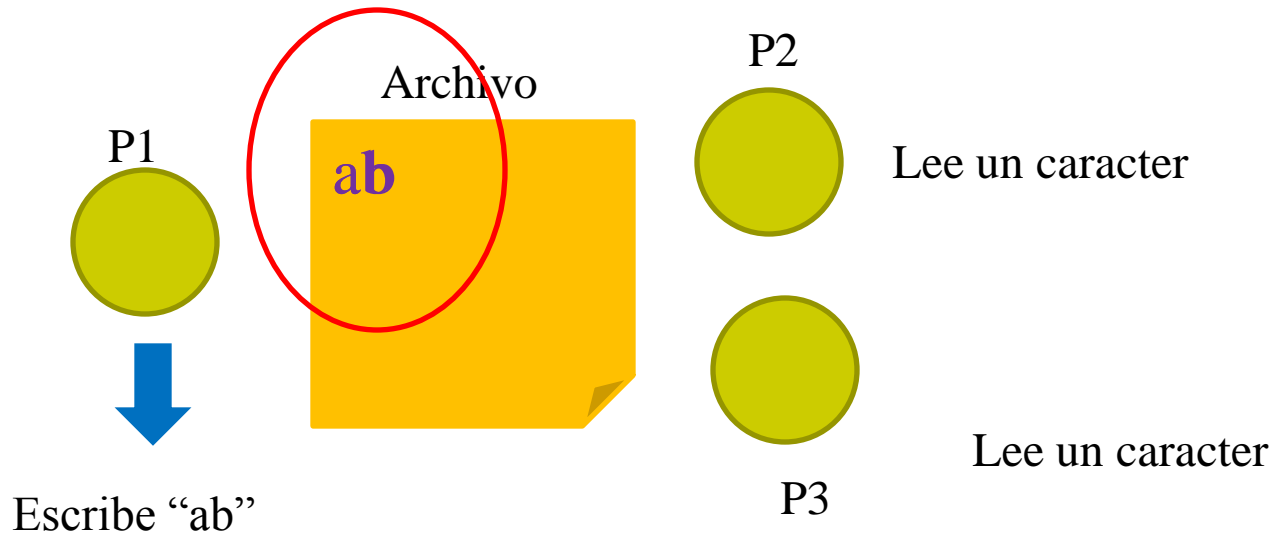


Los datos que se leen del pipe se “extraen” del mismo, es decir no quedan disponibles para otro proceso.





Lo que se lee de un archivo regular si permanece en el archivo,
disponible para otros procesos.



P2 abre el archivo, crea un proceso hijo (P3)
Y luego ambos leen un carácter del archivo.
¿Qué pasa si P2 y P3 son dos procesos independientes y
cada uno abre el archivo para leer un caracter?

Pipes: Reglas

- Las siguientes reglas aplican a los **procesos que leen del pipe:**
 - Si un proceso lee de un pipe cuyo extremo de escritura ha sido cerrado, la llamada al sistema *read()* retorna 0, indicando fin de la entrada.
 - Si un proceso lee de un pipe que esta vacío, y cuyo descriptor de escritura se mantiene todavía abierto, el proceso se suspende hasta que se introduce algún contenido en el pipe.

Pipes:Reglas

- ❑ Si un proceso trata de leer más bytes de los que están almacenados en un pipe, se retorna el contenido actual del pipe, y *read()* retorna la cantidad de bytes leídos.

Pipes:Reglas

- Las siguientes reglas aplican a los **procesos que escriben al pipe**:
 - Si un proceso escribe a un pipe cuyo extremo de lectura ha sido cerrado, el escritor falla y el sistema operativo le envía una **señal (SIGPIPE)**. La acción por defecto de esta señal es **terminar con el proceso que la recibe**.
 - Si un proceso escribe **menos bytes al pipe de lo que este último puede almacenar**, se garantiza que el *write()* es **atómico**; esto significa que se garantiza que el escritor completará su llamada al sistema sin haber sido desalojado del CPU (preempted) por otro proceso escritor.

Pipes:Reglas

- ❑ Si un proceso trata de escribir a un pipe más bytes, de los que éste puede almacenar, no se garantiza la atomicidad de la escritura.
- ❑ Debido a que el acceso a los pipes no nominales es a través de descriptores de archivo, sólo el proceso que crea el pipe y sus descendientes pueden usarlo.
- ❑ La llamada al sistema *lseek* no tiene ningún significado cuando se aplica a un pipe.

Pipes: Reglas

- Si el kernel no tiene suficiente espacio para crear un pipe, la llamada al sistema retorna -1, de otra forma retorna 0.
- Si el pipe esta full, el escritor de turno automáticamente se suspende

Ejemplo 2

`ls -l | sort -n +4`

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

```
int main(void) {
    int fd[2];
    pid_t childpid;
```

`pipe(fd);`



```
if ((childpid = fork()) == 0) { /* ls es el hijo */
    dup2(fd[1], STDOUT_FILENO);
    close(fd[0]);
    close(fd[1]);
    execl("/usr/bin/ls", "ls", "-l", NULL);
    perror("Exec failed");
}
else {
    dup2(fd[0], STDIN_FILENO);
    close(fd[0]);
    close(fd[1]);
    execl("usr/bin/sort", "sort", "-n", "+4", NULL);
    perror("...");
}
```

Padre

Hijo

[0] Entrada estándar

Entrada estándar

[1] Salida estándar

Salida estándar

[2] Error estándar

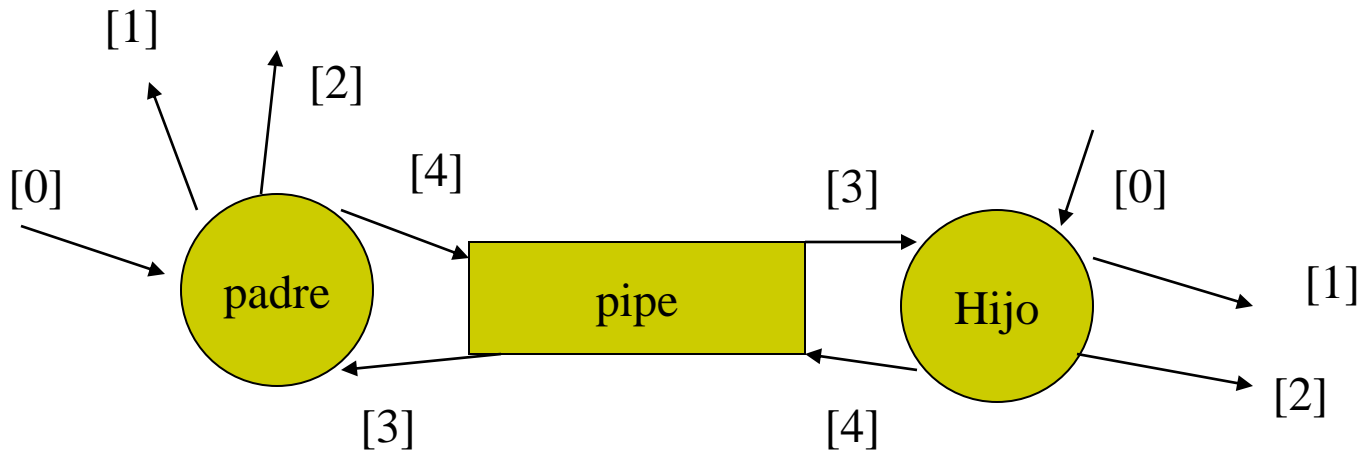
Error estándar

[3] Lectura del pipe

Lectura del pipe

[4] Escritura al Pipe

Escritura al pipe



Después del
fork

Ejemplos

```
ls -l | sort -n +4
```

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

```
int main(void) {
    int fd[2];
    pid_t childpid;
```

```
    pipe(fd);
    if ((childpid = fork()) == 0) { /* ls es el hijo */
        dup2(fd[1], STDOUT_FILENO);
        close(fd[0]);
        close(fd[1]);
        execl("/usr/bin/ls", "ls", "-l", NULL);
        perror("Exec failed");
    }
    else {
        dup2(fd[0], STDIN_FILENO);
        close(fd[0]);
        close(fd[1]);
        execl("usr/bin/sort", "sort", "-n", "+4", NULL);
        perror("...");
    }
}
```

Padre

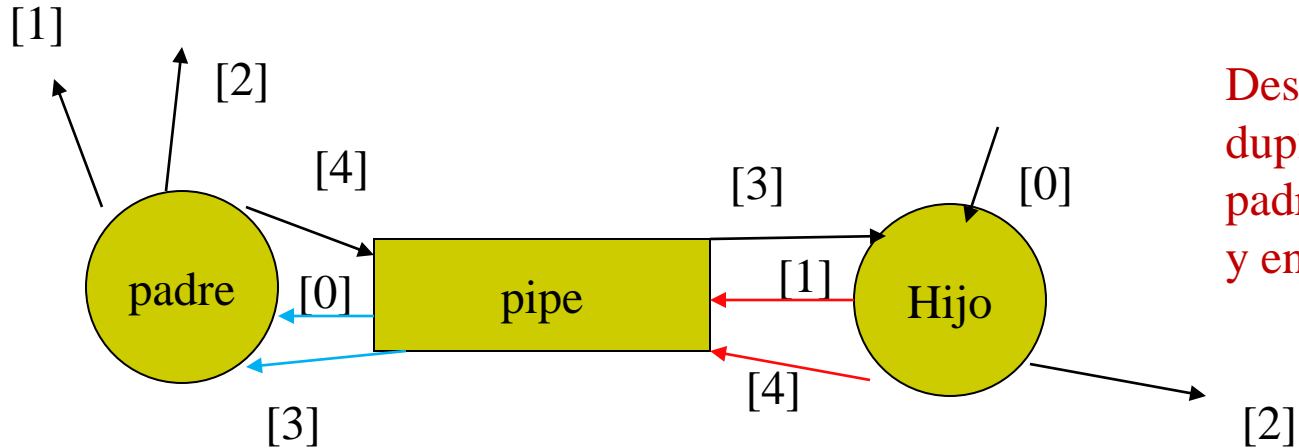
Hijo

| | |
|-----|--------------------|
| [0] | Lectura del pipe |
| [1] | Salida estándar |
| [2] | Error estándar |
| [3] | Lectura del pipe |
| [4] | Escritura del Pipe |

| |
|--------------------|
| Entrada estándar |
| Escritura del pipe |
| Error estándar |
| Lectura del pipe |
| Escritura del pipe |

`dup2(fd[0], STDIN_FILENO);`

`dup2(fd[1], STDOUT_FILENO);`



Ejemplos

```
ls -l | sort -n +4
```

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

```
int main(void) {
    int fd[2];
    pid_t childpid;
```

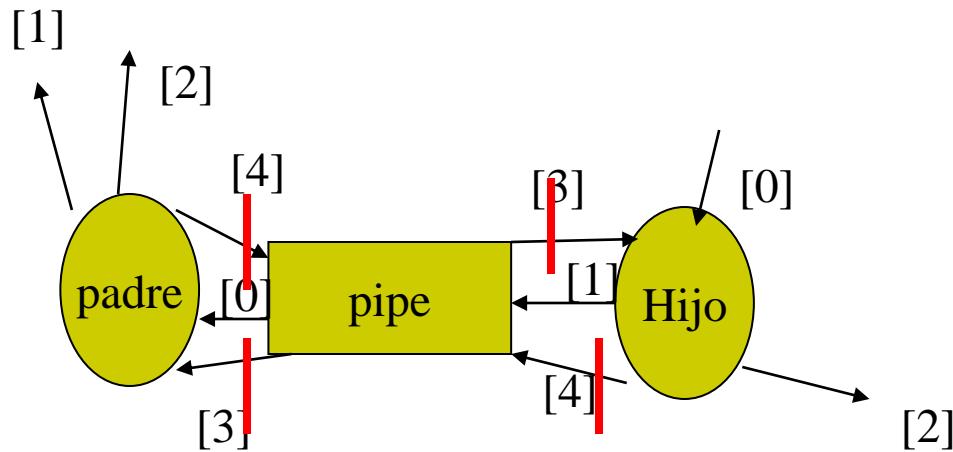
```
    pipe(fd);
    if ((childpid = fork()) == 0) { /* ls es el hijo */
        dup2(fd[1], STDOUT_FILENO);
        close(fd[0]);
        close(fd[1]);
        execl("/usr/bin/ls", "ls", "-l", NULL);
        perror("Exec failed");
    }
    else {
        dup2(fd[0], STDIN_FILENO);
        close(fd[0]);
        close(fd[1]);
        execl("usr/bin/sort", "sort", "-n", "+4", NULL);
        perror("...");
    }
}
```

TD en el Padre

TD en el Hijo

| | |
|-----|------------------|
| [0] | Lectura del Pipe |
| [1] | Salida estándar |
| [2] | Error estándar |
| [3] | |
| [4] | |

| |
|-------------------|
| Entrada estándar |
| Escritura al Pipe |
| Error estándar |
| |
| |

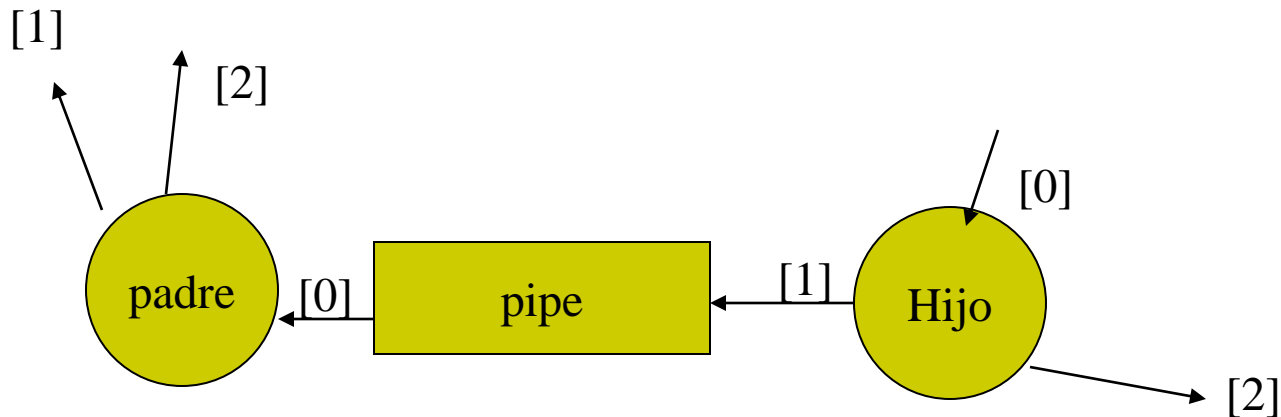


TD en el Padre

TD en el Hijo

| | |
|-----|------------------|
| [0] | Lectura del Pipe |
| [1] | Salida estándar |
| [2] | Error estándar |
| [3] | |
| [4] | |

| |
|-------------------|
| Entrada estándar |
| Escritura al Pipe |
| Error estándar |
| |
| |



Ejemplos


```
ls -l | sort -n +4
```

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

```
int main(void) {
    int fd[2];
    pid_t childpid;
```

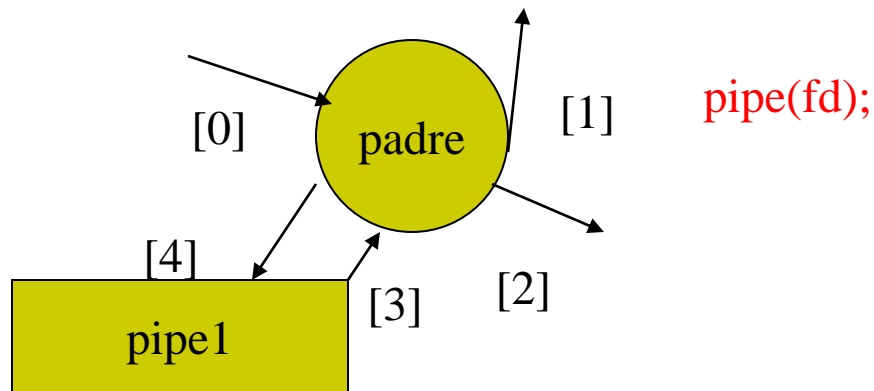
```
    pipe(fd);
    if ((childpid = fork()) == 0) { /* ls es el hijo */
        dup2(fd[1], STDOUT_FILENO);
        close(fd[0]);
        close(fd[1]);
        execl("/usr/bin/ls", "ls", "-l", NULL);
        perror("Exec failed");
    }
    else {
        dup2(fd[0], STDIN_FILENO);
        close(fd[0]);
        close(fd[1]);
        execl("usr/bin/sort", "sort", "-n", "+4", NULL);
        perror("...");
    }
}
```

Ejemplo: Anillo de dos procesos

```
#include <unistd.h>
#define READ 0
#define WRITE 1
main (int argc, char *argv[]) {
    int fd[2];
     pipe(fd);
    dup2(fd[READ],0);
    dup2(fd[WRITE],1);
    close(fd[READ]);
    close(fd[WRITE]);
    pipe(fd);
    if (fork()==0) {
        dup2(fd[WRITE],1);
    }
    else
        dup2(fd[READ],0);
    close(fd[READ]);
    close(fd[WRITE]);
}
```

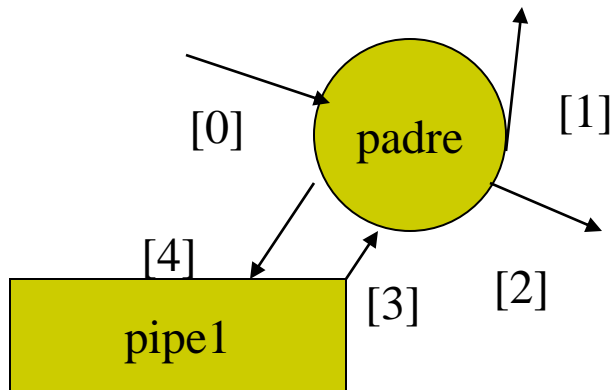
TD después de pipe

| | |
|-----|--------------------|
| [0] | Entrada estándar |
| [1] | Salida estándar |
| [2] | Error estándar |
| [3] | Lectura del pipe |
| [4] | Escritura del Pipe |



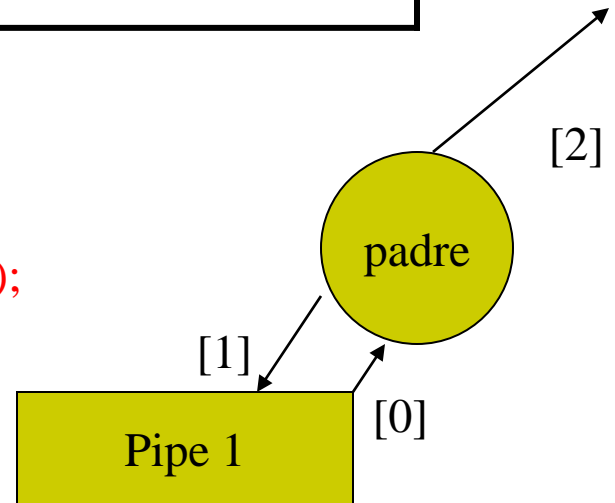
TD después del 2do close

| | | |
|-----|--------------------|---------------------|
| [0] | Entrada estándar | Lectura al pipe 1 |
| [1] | Salida estándar | Escritura al Pipe 1 |
| [2] | Error estándar | Error estándar |
| [3] | Lectura del pipe | |
| [4] | Escritura del Pipe | |

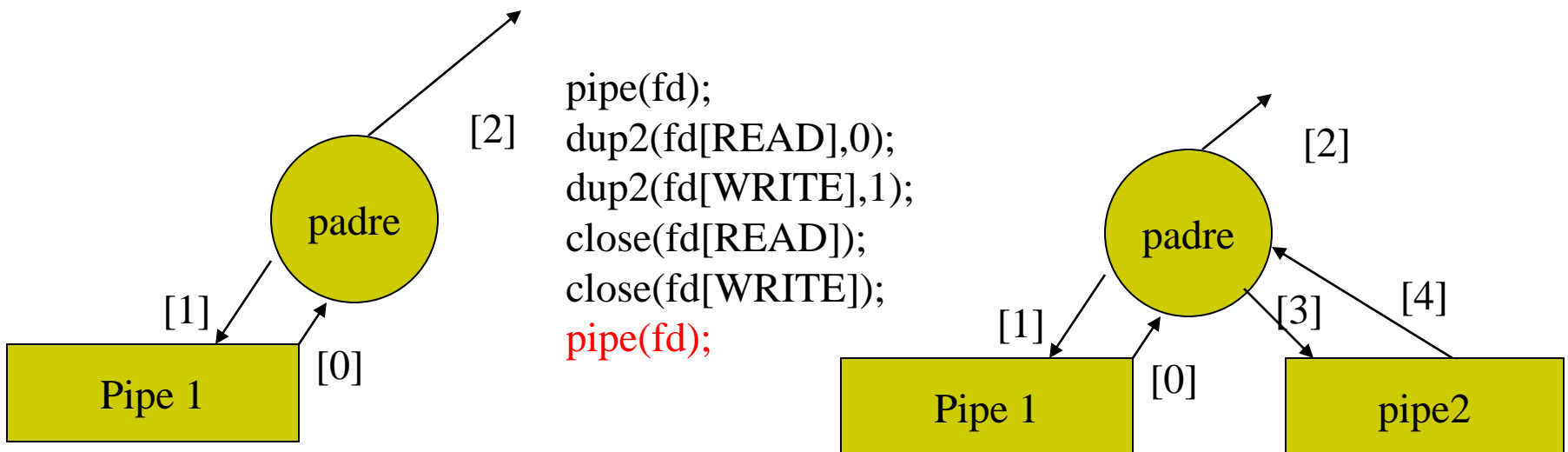
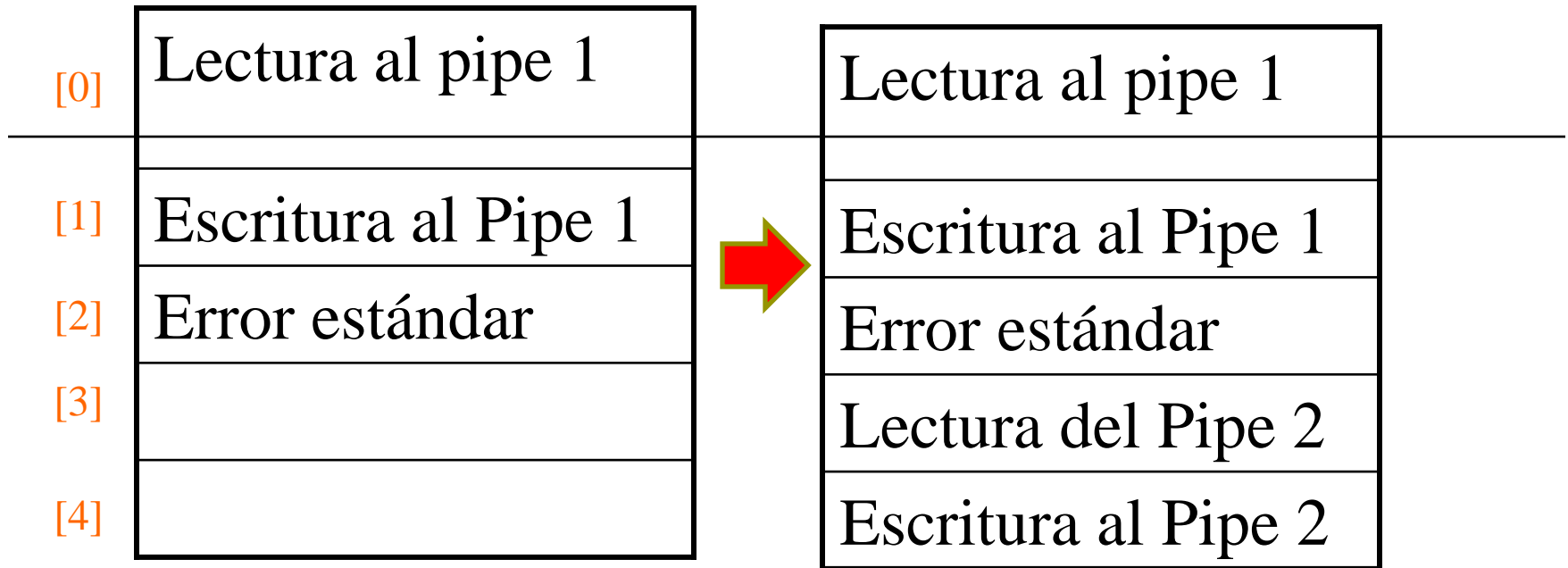


```

pipe(fd);
dup2(fd[READ],0);
dup2(fd[WRITE],1);
close(fd[READ]);
close(fd[WRITE]);
    
```



Después de crear un segundo pipe



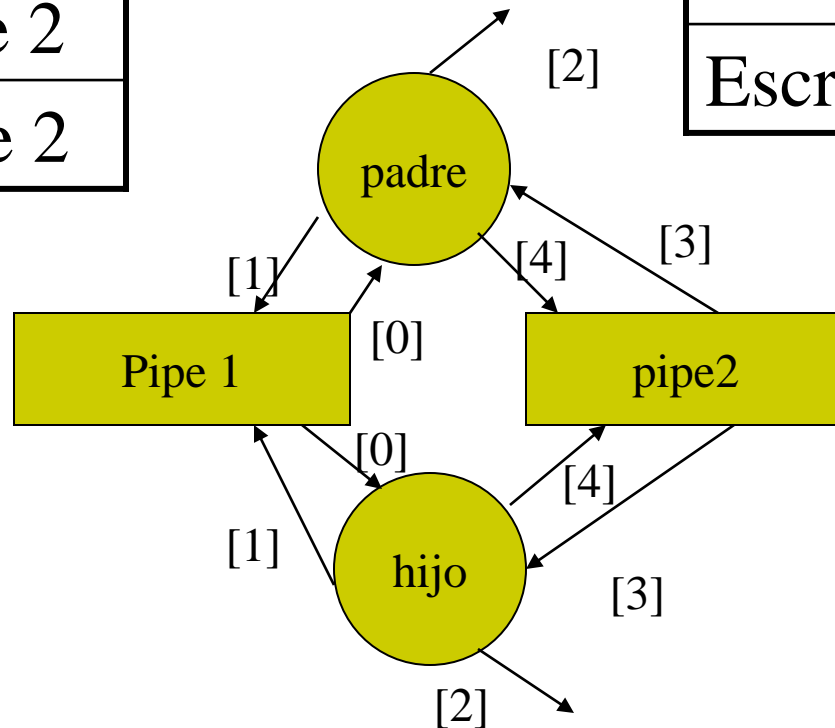
TD del Padre

| |
|---------------------|
| Lectura al pipe 1 |
| Escritura al Pipe 1 |
| Error estándar |
| Lectura del Pipe 2 |
| Escritura al Pipe 2 |

TD del Hijo

| |
|---------------------|
| Lectura al pipe 1 |
| Escritura al Pipe 1 |
| Error estándar |
| Lectura del Pipe 2 |
| Escritura al Pipe 2 |

```
pipe(fd);  
dup2(fd[READ],0);  
dup2(fd[WRITE],1);  
close(fd[READ]);  
close(fd[WRITE]);  
pipe[fd]  
if (fork()==0) {
```

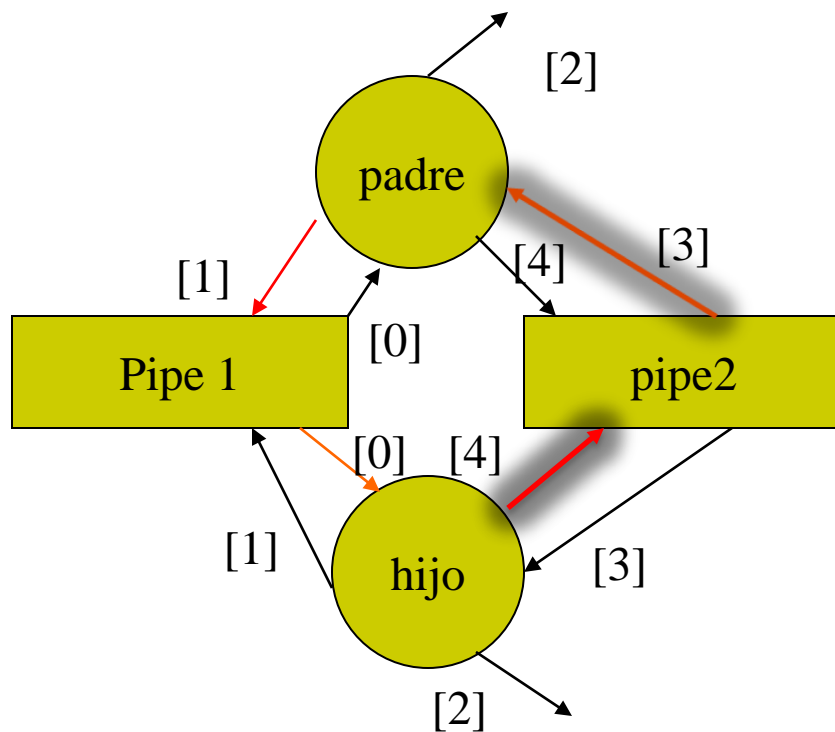


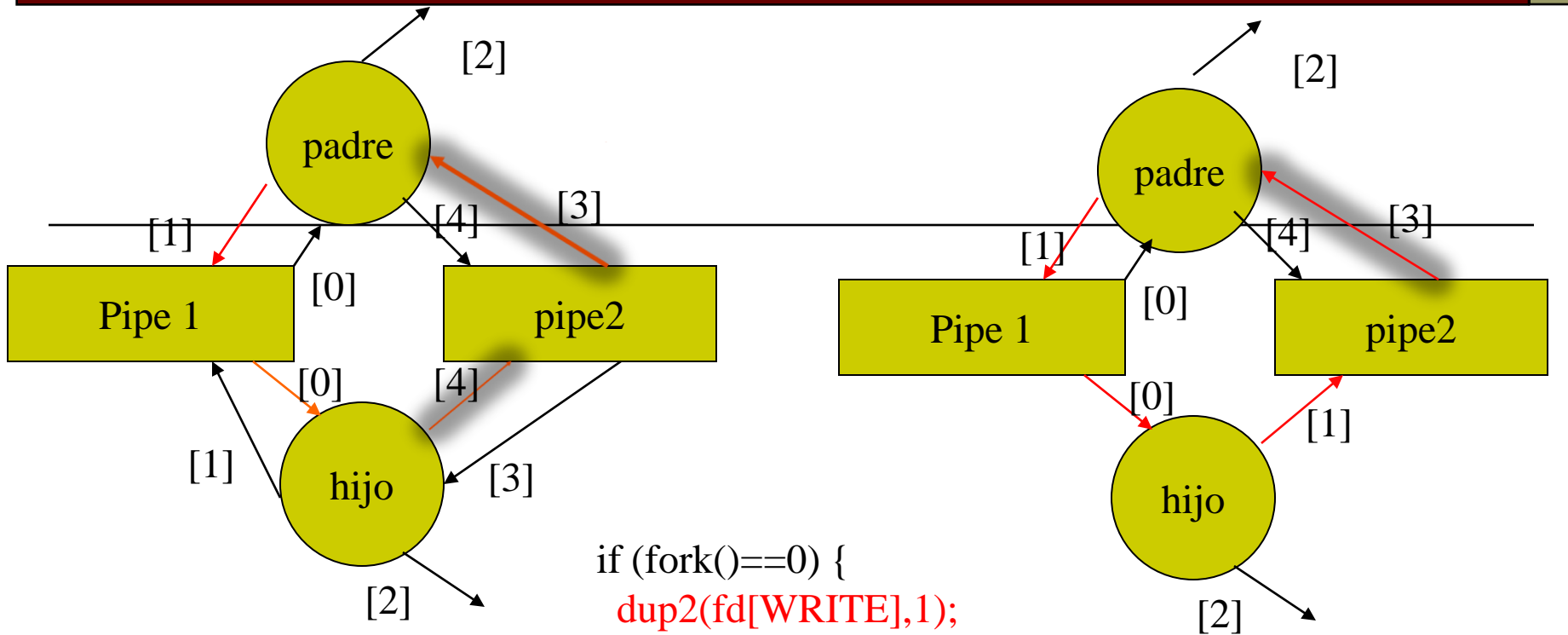
Padre

| |
|---------------------|
| Lectura al pipe 1 |
| Escritura al Pipe 1 |
| Error estándar |
| Lectura del Pipe 2 |
| Escritura al Pipe 2 |

Hijo

| |
|---------------------|
| Lectura al pipe 1 |
| Escritura al Pipe 1 |
| Error estándar |
| Lectura del Pipe 2 |
| Escritura al Pipe 2 |





```

if (fork()==0) {
    dup2(fd[WRITE],1);
} else
    dup2(fd[READ],0);
    close(fd[READ]);
    close(fd[WRITE]);

```

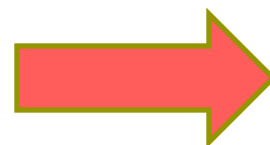


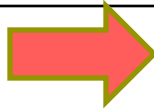
Tabla de Descriptores del **Hijo**

| |
|---------------------|
| Lectura al pipe 1 |
| Escritura al Pipe 2 |
| Error estándar |
| Lectura del Pipe 2 |
| Escritura al Pipe 2 |

| |
|---------------------|
| Lectura del pipe 1 |
| Escritura al Pipe 2 |
| Error estándar |
| |
| |

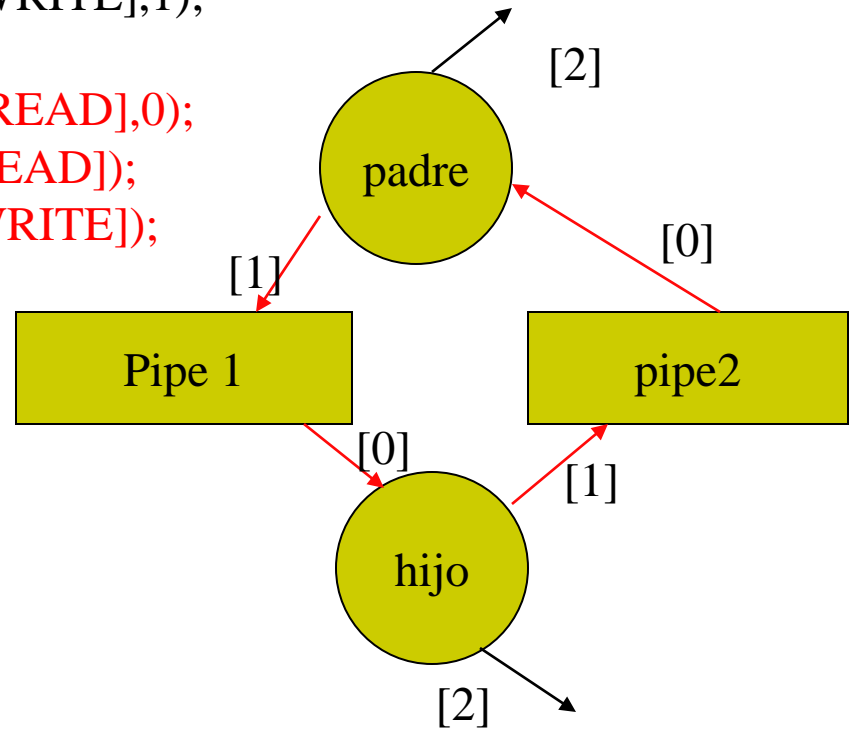
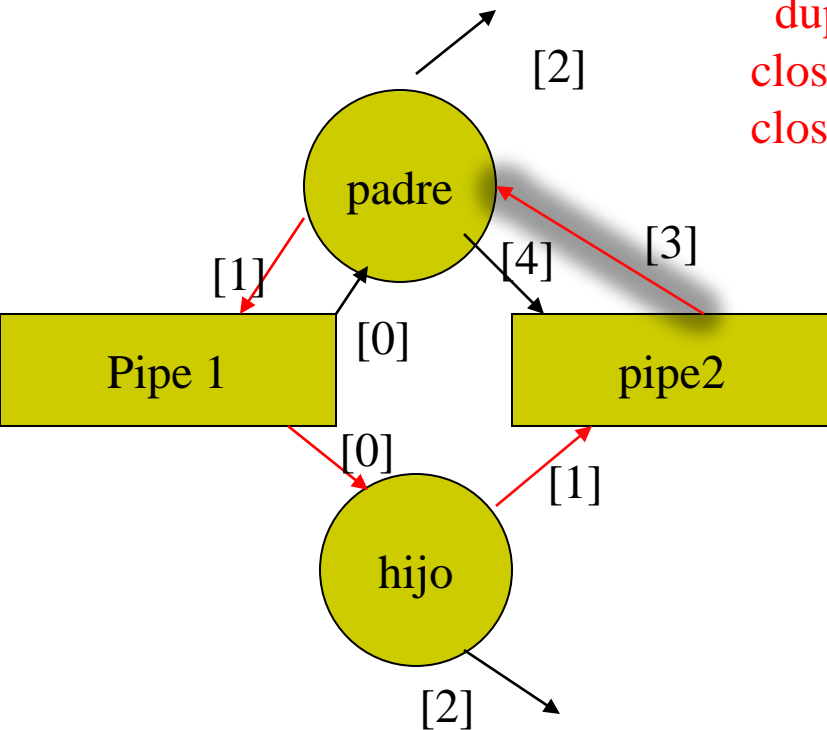
Tabla de Descriptores del Padre

| |
|---------------------|
| Lectura al pipe 1 |
| Escritura al Pipe 1 |
| Error estándar |
| Lectura del Pipe 2 |
| Escritura al Pipe 2 |



| |
|---------------------|
| Lectura al pipe 2 |
| Escritura al Pipe 1 |
| Error estándar |

```
if (fork()==0) {
    dup2(fd[WRITE],1);
} else
    dup2(fd[READ],0);
close(fd[READ]);
close(fd[WRITE]);
```





Ejercicio

- Realice un anillo de N procesos.

Pipes Nominales

- Los pipes nominales tienen menos restricciones que los pipes no nominales, y entre sus ventajas se encuentran:
 - que existe en el sistema de archivos. **Tienen un nombre**
 - Pueden ser usados por **procesos que no tienen ninguna relación.**
 - Existen hasta que son borrados explícitamente.

Pipes Nominales

- Todas las reglas mencionadas para los pipes no nominales aplican para los pipes nominales. Excepto que los pipes nominales tienen mayor capacidad.
- Los pipes nominales son archivos especiales del sistema de archivos y pueden crearse de las siguientes formas:
 - Usando el comando **mknod** desde el shell.
 - Usando la llamada al sistema **mknod()** o la llamada al sistema **mkfifo**.

Pipes Nominales

- La llamada al sistema `mknod` se utiliza para crear un archivo especial, un directorio o un pipe nominal.
`mknod(filename, modes, dev)`
- El tipo de archivo que se desea crear se le indica en el parámetro **`modes`**. El parámetro **`dev`** sólo se usa cuando se desean crear archivos para dispositivos tipo bloque o caracter. Para crear un pipe nominal usando la llamada al sistema `mknod`, se puede usar la macro **`S_IFIFO`** para el modo. Por ejemplo:

`mknod("mypipe", S_IFIFO, 0)`

Pipes Nominales

- Cuando un pipe nominal no se necesita más, se debe eliminar del sistema de archivos usando la llamada al sistema **unlink**.
- Al igual que los pipes no nominales, un pipe nominal sólo debe usarse para la comunicación en un solo sentido. El proceso que escribe, debe abrir el pipe sólo para escritura; y el proceso lector únicamente para lectura.

Ejemplo

Programa Lector

```
main (int argc, char *argv[]) {
    int fd;
    char message[100];
    unlink("aPipe");
    mknod("aPipe", S_IFIFO, 0);
    chmod("aPipe", 0660);
    fd = open("aPipe", O_RDONLY);
    while(readline(fd,message))
        printf("%s\n", message);
    close(fd);
}
```

```
readline(int fd, char *str) {
    /* Lee una linea terminada en NULL desde
    fd hasta str. Retorna 0 cuando
    ya no hay más caracteres en la entrada y 1
    en otro caso */
    int n;
    do {
        n = read(fd, str, 1);
    } while(n > 0 && *str++ != NULL);
    return(n > 0);
}
```

Ejemplo

Programa Escritor

```
main (int argc, char *argv[]) {
    int fd, messagelen,i;
    char message[100];
    sprintf(message, "Hello from PID %d", getpid());
    messagelen = strlen(message) + 1;
    do {
        fd = open("aPipe", O_WRONLY|O_NONBLOCK);
        if (fd == -1) sleep(1);
    } while(fd == -1);
    for (i = 1; i < 4; i++) {
        write(fd, message, messagelen);
        sleep(3);
    }
    close(fd);
}
```

mkfifo

```
mode_t fifo_mode = S_IRUSR | S_IWUSR;  
mkfifo(file, fifo_mode);
```

Inclur:

<sys/stat.h>

<sys/types.h>



Bibliografía
