Fundamentos de Software de Comunicaciones

Tema 2

Programación del Sistema Operativo

(2ª parte)

Comunicación entre procesos

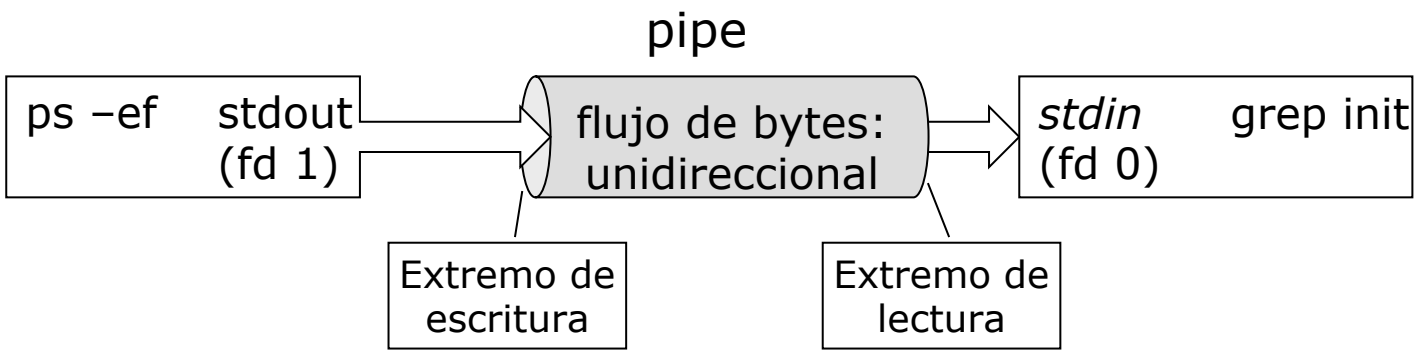
- Los procesos pueden comunicarse entre sí través de las herramientas que proporciona el S.O. y que se denominan IPCs (Inter-Process Communication)

- Tipos de IPC:
  - **Pipes UNIX Half-duplex**
  - **FIFOs (pipes con nombre)**
  - Colas de mensajes estilo System V
  - Semáforos estilo System V
  - Segmentos de memoria compartida estilo System V
  - Sockets (estilo Berkeley) TCP/IP y Unix

- Diferencias entre tuberías
  - Sin nombre (pipes)
    - Permiten comunicar procesos **con un ancestro común**
    - Requieren compartir descriptores de ficheros
  - Con nombre (FIFOS)
    - Existen en el sistema de ficheros (tienen nombre)
    - Permiten comunicar **cualesquiera dos procesos** (sin relación de parentesco entre ellos)

Pipes (tuberias de datos)

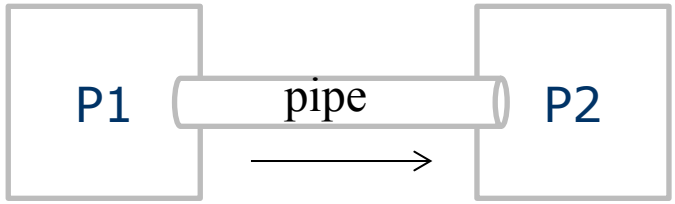
- Las tuberías de datos **existen en consola**
  - Simplemente conectan la salida estándar de un proceso con la entrada estándar de otro
  - Ejemplo**:** **ps -ef | grep init**



- El proceso que escribe (ps –ef) tiene su salida estándar (stdout, **fd = 1**) conectada al extremo de escritura de la tubería
- El proceso que lee (grep init) tiene su entrada estándar (stdin, **fd = 0**) conectada al extremo de lectura
- Ambos procesos no saben de la existencia de la tubería, si no que leen y escriben en sus descriptores habituales

Pipes (tuberias de datos)

- Pero una tubería también se utiliza como **canal de comunicación** especializado entre dos programas en C:
  - Se pueden escribir bytes usando **write()**
  - Se pueden leer bytes usando **read()**



- Material complementario
  - En jabega.uma.es
  - Libro: **The Linux Programming Interface**
  - Capítulo 44: https://learning.oreilly.com/library/view/the-linux-programming/9781593272203/xhtml/ch44.xhtml

Pipes: características fundamentales

- Es un buffer que está en la **memoria del núcleo**, y tiene una **capacidad máxima**, que depende de la implementación

- Las tubería son **flujo** (stream) de bytes
  - La misma abstracción que un fichero
  - Los datos pasan la tubería secuencialmente: se leen en el orden que se escriben
  - **No existe el estructura** alguna de los datos: son bytes uno detrás de otro (como los ficheros)
  - Cualquier noción de estructura ha de ser implementada en nuestro código

- Las tuberías son de **un solo sentido**
  - Trabajan en modo **half-duplex**
  - Tienen **dos extremos**: uno para **escribir**, y otro para **leer**
  - Se requieren **dos tuberías** para hacer una comunicación en los dos sentidos (**full-duplex**)

Pipes: características fundamentales

- Es un canal de comunicaciones que **no introduce errores** en la transmisión de los datos
  - El núcleo del sistema operativo asegura que permanecerán inmutables hasta que un proceso las lea.
  - Una vez leídas (en el mismo orden en el que se escribieron), sí desaparecen.

- **Tampoco introduce retardos de comunicación**, en el sentido clásico
  - Estamos en la misma máquina
  - Sí habrá un retardo inevitable por las copias de datos entre zonas de memoria

FIFOS: tuberías con nombre

- Las tuberías con nombre **existen en el sistema de archivos** como un archivo de dispositivo especial (ver **ls –l**)
  - OJO: la información no está en el disco, si no en la memoria del kérnel
- Permite la comunicación entre dos procesos cualesquiera
  - Un **escritor**, que abrirá el la fifo en **modo escritura**
  - Un **lector**, que abrirá la fifo en **modo lectura**
- La comunicación se realiza **enviando** y **recibiendo** bytes de la misma forma que en **ficheros regulares** → read/write
- Cuando se han realizado todas las operaciones de E/S por los procesos, la tubería con nombre permanece en el sistema de archivos

Creación de FIFOs en consola

- Teclear: **mkfifo tuberia_fsc**
- Para verla:

  >% ls -l nombre_fifo

```
prw-rw-r-- 1 alumno alumno 0 Nov 7 01:59 tuberia_fsc
```

- Ejemplo:
  - Partiendo de la tubería anterior, abrir dos terminales
    - Terminal 1: **cat tuberia_fsc**
    - Terminal 2: **cat > tuberia_fsc**
  - Los procesos no se desbloquean hasta que no se abren ambos extremos
  - Escribir en el **Terminal 2** y finalizar con **Ctrl+D**
  - ¿Qué pasa?

Uso de FIFOs: escritor

- Código que abre la fifo, escribe por ella, termina
- Compila y ejecuta (faltan #include)
- ¿Funciona?
  - **cat /tmp/tuberia_fsc**

```c
#define T 512
int main()
{
    int fd = open("/tmp/tuberia_fsc", O_WRONLY);
    if (fd < 0) {
        perror("open");
        exit(1);
    }
    printf("voy a escribir!\n");

    char datos[T] = "prueba total";

    size_t bytes_a_escribir = strlen(datos);

    int escritos = write(fd, datos, bytes_a_escribir);

    if (escritos < bytes_a_escribir) {
        perror("error en write");
        exit(1);
    }

    if (close(fd) < 0) {
        perror("close");
        exit(1);
    }
    return 0;
}
```

FIFOS: tuberías con nombre

- La apertura de una FIFO tiene una semántica especial
  - Abrir una FIFO para **lectura** (open() con flag **O_RDONLY**) **bloquea** hasta que otro proceso abre la FIFO para **escritura** (open() con **O_WRONLY**)
  - Ocurre lo mismo al contrario: abrir para escritura bloque hasta que no se abre el otro extremo para lectura
  - Es decir: ambos procesos se **sincronizan**

- Si una FIFO ya tiene un extremo abierto (e.g., **cat > /tmp/fifo**), la llamada a **open()** no bloquea

- Creación de una FIFO en el **código fuente**

```
int mkfifo(const char *pathname, mode_t mode);
```

  - Previamente se llama a la función **umask(0);** para poder asignarle los permisos necesarios

Uso de FIFOs: lector

- Recordatorio:
  - La tubería almacena bytes
  - Se leen en el array
- También es posible usar el lector con la consola
  - **cat >/tmp/tuberia_fsc**

```c
#define T 512
int main() {
    int fd = open("/tmp/tuberia_fsc", O_RDONLY);
    if (fd < 0) {
        perror("error en open");
        exit(1);
    }
    char datos[T];
    int leidos = read(fd, datos, T - 1);
    if (leidos < 0) {
        perror("read");
        exit(1);
    }
    int escritos = write(1, datos, leidos);
    if (escritos < leidos) {
        perror("error en write");
        exit(1);
    }

    if (close(fd) < 0 ) {
        perror("close");
        exit(1);
    }

    return 0;
}
```

Sincronización y semántica de **read**() para canales de comunicaciones

- Usaremos la función **read**() en el extremo de lectura

```
leidos = read(fd, buffer, T);
        //fd es un canal: tubería/socket
```

- Comportamiento de **read()**
  - Si hay datos disponibles, lee los que ha podido leer (no tiene porqué ser **T**)
  - La función read() se **bloquea** al leer de un canal **vacío**
  - La función read() devuelve **0** al leer de un canal **cerrado**
    - Es como si se llegase al fin de fichero (EOF)
    - Devuelve 0 tantas veces como se llame a la función
    - **OJO: no falla**!

- La lectura se puede realizar en bloques de cualquier tamaño, independientemente de cómo se escribieron por el extremo de escritura

Sincronización y semántica de **write**() para canales de comunicaciones

- Se usa la función write() en el extremo de escritura

```
int v = 5;
write(fd, &v, sizeof(int));
```

- Comportamiento de **write()**
  - Al tener capacidad limitada, la función write() **bloquea** al proceso cuando intenta escribir en una tubería **llena**
  - Si write() no puede escribir todos los datos solicitados en una tubería, escribe sólo escribirá aquellos que quepan en el espacio libre → escribir "Hola" en una tubería con 3 bytes disponibles
  - Intentar escribir en canal **cerrado:**
    1. El kernel envíe la señal **SIGPIPE**, que por defecto mata el proceso
    2. Si **SIGPIPE** se ignora/maneja, entonces write() devuelve -1 y fija **errno** a **EPIPE**

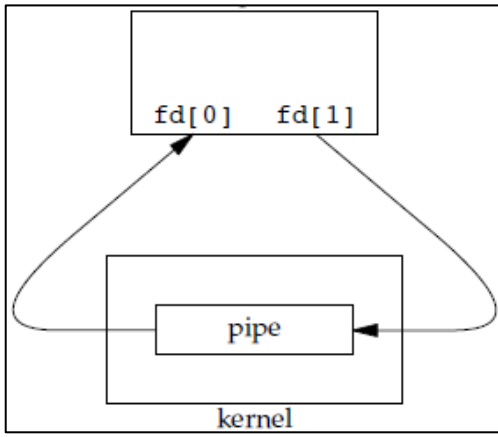
read() vs read_n()
write() vs write_n()

- ¿Cuándo usar **write_n**()?
  - SIEMPRE

- ¿Cuándo usar **read_n**()?
  - Cuando se conoce el **número de bytes a leer**
  - Variables enteras, float, chars... y **arrays** cuando **se sabe de antemano su longitud**

Creación de pipes en C

- Una tubería es un **array de dos enteros** que almacenará dos **descriptores de ficheros** → `int p[2];`
  - En la posición cero (`p[0]`) está el extremo para leer datos
  - En la posición uno (`p[1]`) está el extremo para escribir datos



- Para crear la tubería se emplea la función:

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

  - Crea dos descriptores de fichero y almacena su valor en los dos enteros que contiene el array **`pipefd`**
  - El primer descriptor de fichero se abre como **`O_RDONLY:`** sólo para lecturas
  - El segundo se abre como **`O_WRONLY`**: sólo para escritura

```
int p[2];
pipe(p);
//p[0] está abierto para lectura
//p[1] está abierto para escritura
```

  - Devuelve **0** si tuvo éxito, **-1** en caso de error, con **errno**: **EMFILE** (tabla de ficheros del sistema llena) o **EFAULT** (el vector fd no es valido)

- Una vez creado un pipe, se podrán hacer lecturas y escrituras como si se tratase de cualquier fichero
  - `write(tuberia[1],"hola",4);`
  - `read(tuberia[0], buffer, MAX_BUFFER_SIZE);`

Cierre de tuberías

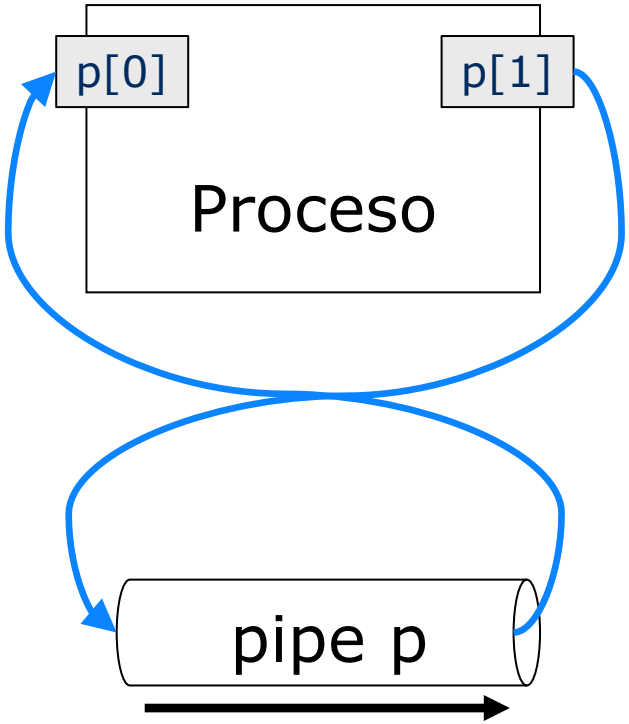
- Acción de suma importancia para su correcto funcionamiento
- Los recursos de la tubería en el núcleo no se liberan hasta que todos los descriptores asociados están cerrados
- El proceso que lee de la pipe
  - Debe cerrar el extremo de escritura para que, cuando el proceso que escribe termine, pueda ver el EOF
  - Si no lo cierra, no podrá ver EOF ya que el núcleo sabe que aún hay un extremo de escritura abierto
- El proceso que escribe en la pipe
  - Debe cerrar el extremo de lectura para indicarle al otro proceso el estado de la tubería (si trata de escribir, recibe SIGPIPE o EPIPE)
  - Si no lo cierra, y al no leer de él, el proceso que escribe en el otro extremo podría quedar bloqueado indefinidamente si la tubería se llena

Padres, hijos y pipes

- Las tuberías sin nombre han de usarse entre procesos con relación de parentesco
- Un proceso hijo hereda todos los descriptores de ficheros abiertos de su padre y, por tanto, comparten los extremos de lectura y escritura
- Para asegurar la unidireccionalidad de la tubería, es necesario que tanto padre como hijo cierren los respectivos descriptores de ficheros que no van a usar
  - Si la comunicación va a ser de padre a hijo
    - El padre cierra el descriptor de lectura
    - El hijo cierra el descriptor de escritura
  - Si la comunicación va a ser de hijo a padre
    - El padre cierra el descriptor de escritura
    - El hijo cierra el descriptor de lectura

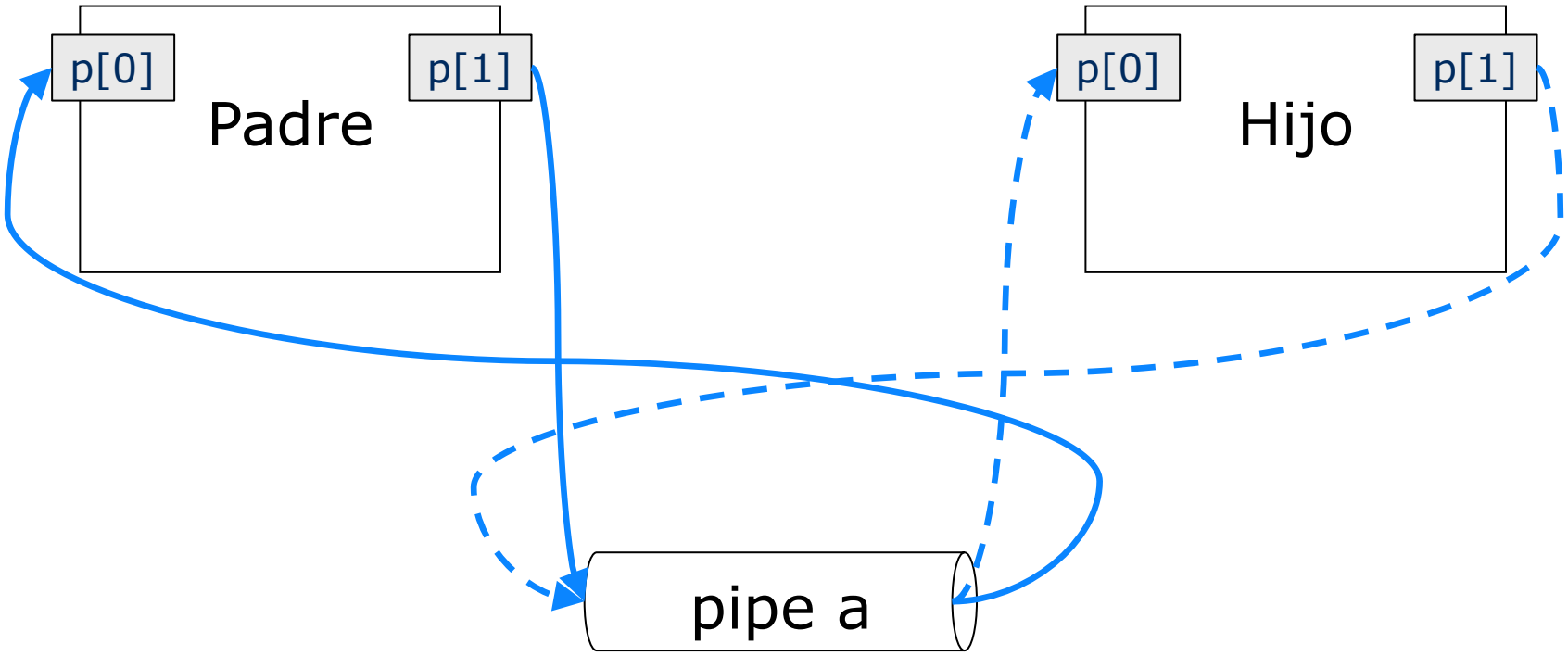
Comunicación unidireccional con pipes

- Después de crear las **pipe**, y antes del **fork**()



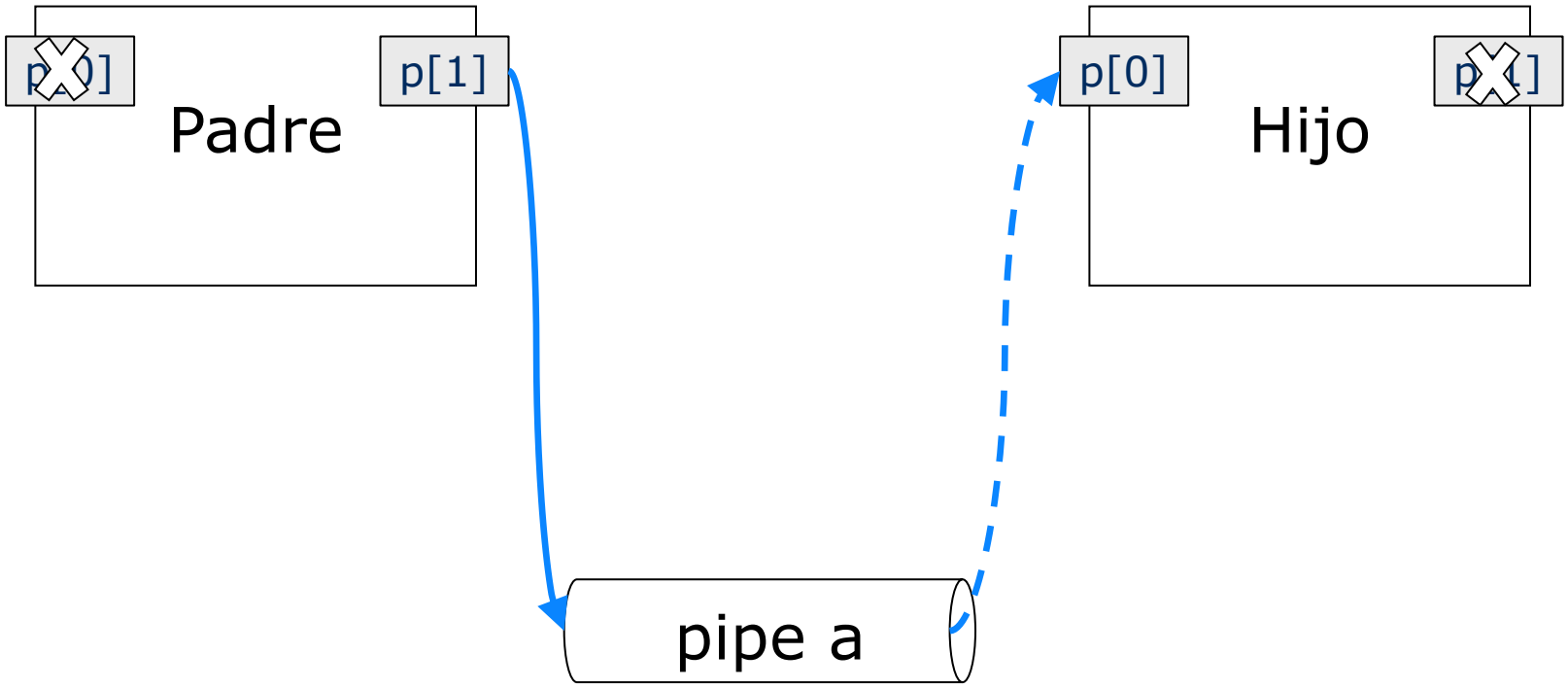
Comunicación unidireccional con pipes

- Después clonar con **fork**()
  - En línea discontinua, los descriptores copiados tras la clonación



Comunicación unidireccional con pipes

- Después clonar con **fork**()
  - En línea discontinua, los descriptores copiados tras la clonación
  - Padre cierra **p[0]**
  - El hijo cierra **p[1]**



Ejemplo 1

1. Se asume que la tubería está creada, siendo **Proceso1** el que **escribe** y **Proceso2** el que **lee**
2. Mientras que **Proceso1** no escriba, **Proceso2** está **bloqueado** (lín. 30)

Proceso1

Código ejecutable

```
11  // Se asume que existen
12  // int p[2] ;
13
14  // Y que la pipe está creada
15  // pipe(p);
16
17  char b[T] = "abc";
18  write(p[1], b, 3);
19  close(p[1]);
```

Montículo (heap)

################	
…	
b[3]	'\0'
b[2]	'c'
b[1]	'b'
b[0]	'a'
p[1]	

Pila (stack)

p[1] pipe p[0]

Proceso2

Código ejecutable

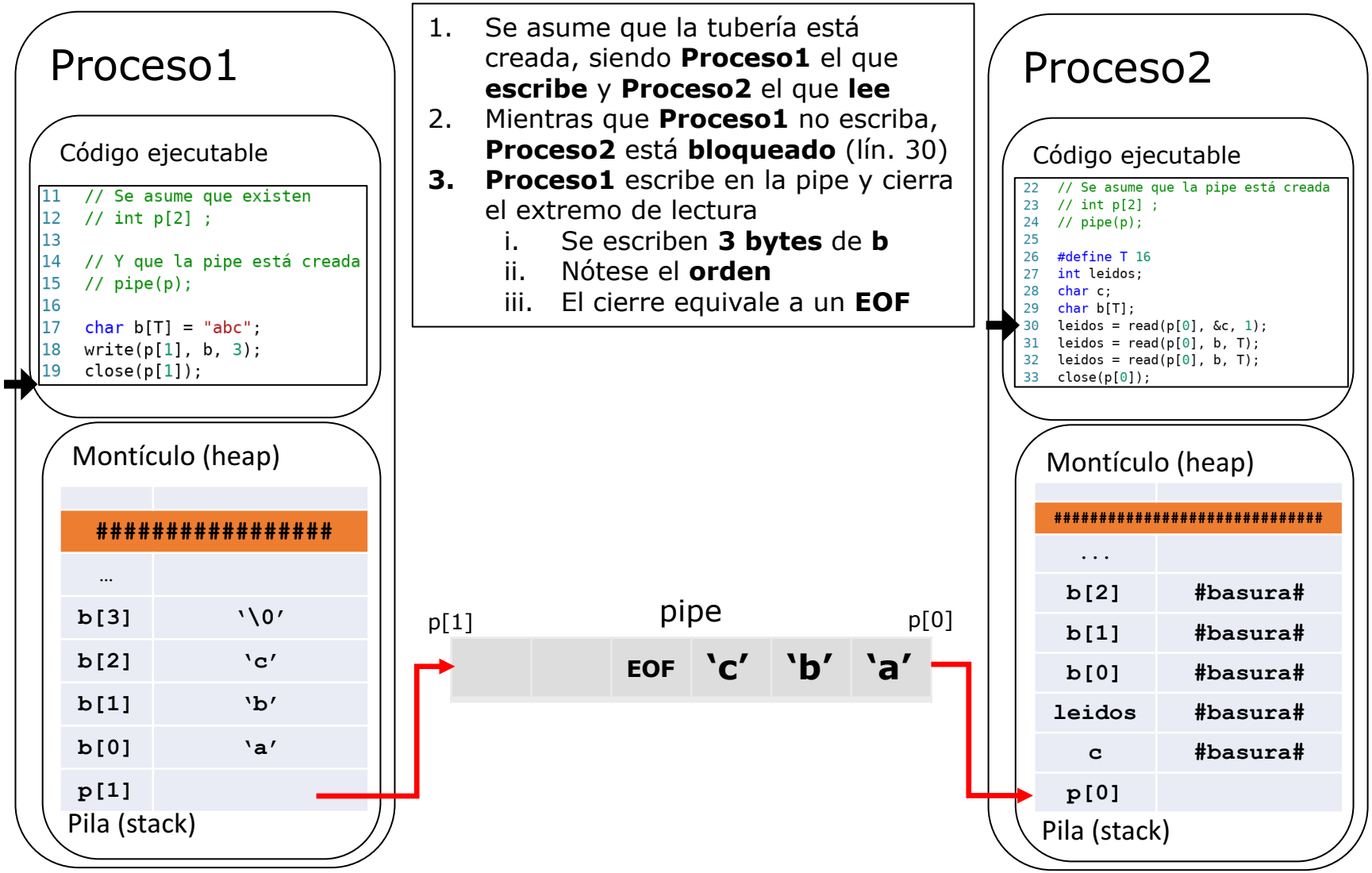
```
22  // Se asume que la pipe está creada
23  // int p[2] ;
24  // pipe(p);
25
26  #define T 16
27  int leidos;
28  char c;
29  char b[T];
30  leidos = read(p[0], &c, 1);
31  leidos = read(p[0], b, T);
32  leidos = read(p[0], b, T);
33  close(p[0]);
```

Montículo (heap)

##############################	
…	
b[2]	#basura#
b[1]	#basura#
b[0]	#basura#
leidos	#basura#
c	#basura#
p[0]	

Pila (stack)

Ejemplo 1


Proceso1
Código ejecutable
11 // Se asume que existen
12 // int p[2] ;
13
14 // Y que la pipe está creada
15 // pipe(p);
16
17 char b[T] = "abc";
18 write(p[1], b, 3);
19 close(p[1]);
Montículo (heap)
##################
…
b[3] '\0'
b[2] 'c'
b[1] 'b'
b[0] 'a'
p[1]
Pila (stack)
1. Se asume que la tubería está creada, siendo Proceso1 el que escribe y Proceso2 el que lee
2. Mientras que Proceso1 no escriba, Proceso2 está bloqueado (lín. 30)
3. Proceso1 escribe en la pipe y cierra el extremo de lectura
i. Se escriben 3 bytes de b
ii. Nótese el orden
iii. El cierre equivale a un EOF
p[1]
pipe
p[0]
EOF 'c' 'b' 'a'
Proceso2
Código ejecutable
22 // Se asume que la pipe está creada
23 // int p[2] ;
24 // pipe(p);
25
26 #define T 16
27 int leidos;
28 char c;
29 char b[T];
30 leidos = read(p[0], &c, 1);
31 leidos = read(p[0], b, T);
32 leidos = read(p[0], b, T);
33 close(p[0]);
Montículo (heap)
##############################
...
b[2] #basura#
b[1] #basura#
b[0] #basura#
leidos #basura#
c #basura#
p[0]
Pila (stack)

Ejemplo 1

Proceso1

Código ejecutable

```
11  // Se asume que existen
12  // int p[2] ;
13
14  // Y que la pipe está creada
15  // pipe(p);
16
17  char b[T] = "abc";
18  write(p[1], b, 3);
19  close(p[1]);
```

Montículo (heap)

################	
…	
b[3]	'\0'
b[2]	'c'
b[1]	'b'
b[0]	'a'
p[1]	

Pila (stack)

1. Se asume que la tubería está creada, siendo **Proceso1** el que **escribe** y **Proceso2** el que **lee**
2. Mientras que **Proceso1** no escriba, **Proceso2** está **bloqueado** (lín. 29)
3. **Proceso1** escribe en la pipe y cierra el extremo de lectura
4. **Proceso2** lee el primer byte de la pipe, y guarda su contenido en **c**
   i. Se actualiza el valor de **leidos**
   ii. El byte con el carácter **'a'** desaparece de la pipe
   iii. Nótese que se puede leer cualquier número de bytes, independientemente de cómo se hayan escrito

Proceso2

Código ejecutable

```
22  // Se asume que la pipe está creada
23  // int p[2] ;
24  // pipe(p);
25
26  #define T 16
27  int leidos;
28  char c;
29  char b[T];
30  leidos = read(p[0], &c, 1);
31  leidos = read(p[0], b, T);
32  leidos = read(p[0], b, T);
33  close(p[0]);
```

Montículo (heap)

##############################	
…	
b[2]	#basura#
b[1]	#basura#
b[0]	#basura#
leidos	1
c	'a'
p[0]	

Pila (stack)

Ejemplo 1

Proceso1

Código ejecutable

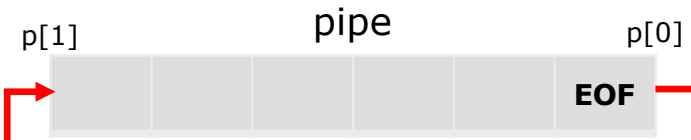
```
11  // Se asume que existen
12  // int p[2] ;
13
14  // Y que la pipe está creada
15  // pipe(p);
16
17  char b[T] = "abc";
18  write(p[1], b, 3);
19  close(p[1]);
```

Montículo (heap)

################	
…	
b[3]	'\0'
b[2]	'c'
b[1]	'b'
b[0]	'a'
p[1]	

Pila (stack)

1. Se asume que la tubería está creada, siendo **Proceso1** el que **escribe** y **Proceso2** el que **lee**
2. Mientras que **Proceso1** no escriba, **Proceso2** está **bloqueado** (lín. 29)
3. **Proceso1** escribe en la pipe y cierra el extremo de lectura
4. **Proceso2** lee el primer byte de la pipe, y guarda su contenido en **c**
5. Asumiendo **T = 16**, en la siguiente lectura (lín.32), **read()** sólo copia en **b** los dos caracteres restantes
   i. **leidos** se actualiza con el valor correspondiente



Proceso2

Código ejecutable

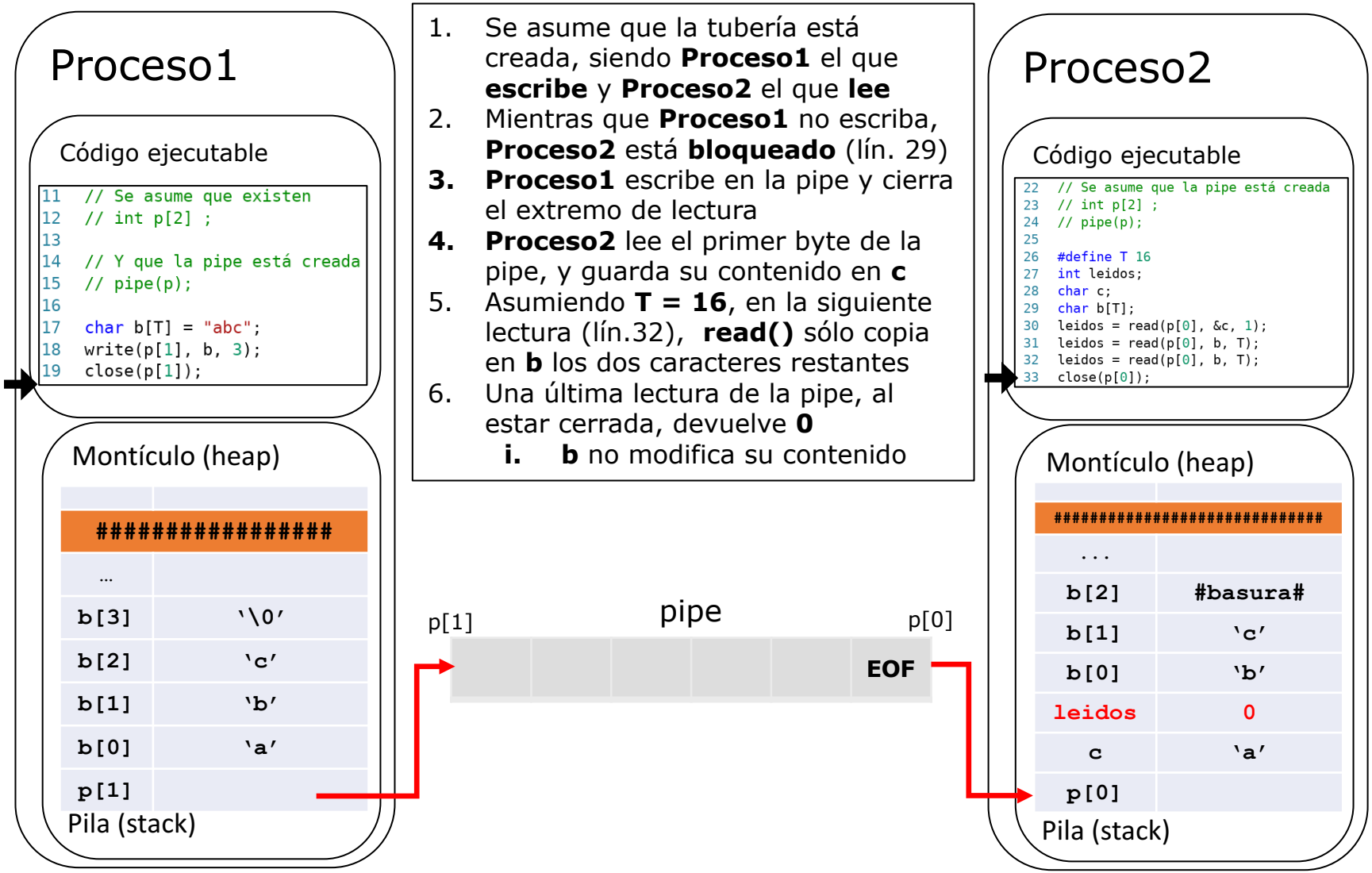
```
22  // Se asume que la pipe está creada
23  // int p[2] ;
24  // pipe(p);
25
26  #define T 16
27  int leidos;
28  char c;
29  char b[T];
30  leidos = read(p[0], &c, 1);
31  leidos = read(p[0], b, T);
32  leidos = read(p[0], b, T);
33  close(p[0]);
```

Montículo (heap)

##############################	
…	
b[2]	#basura#
b[1]	'c'
b[0]	'b'
leidos	2
c	'a'
p[0]	

Pila (stack)

Ejemplo 1


Proceso1
Código ejecutable
11 // Se asume que existen
12 // int p[2] ;
13
14 // Y que la pipe está creada
15 // pipe(p);
16
17 char b[T] = "abc";
18 write(p[1], b, 3);
19 close(p[1]);
Montículo (heap)
##################
…
b[3] '\0'
b[2] 'c'
b[1] 'b'
b[0] 'a'
p[1]
Pila (stack)
1. Se asume que la tubería está creada, siendo Proceso1 el que escribe y Proceso2 el que lee
2. Mientras que Proceso1 no escriba, Proceso2 está bloqueado (lín. 29)
3. Proceso1 escribe en la pipe y cierra el extremo de lectura
4. Proceso2 lee el primer byte de la pipe, y guarda su contenido en c
5. Asumiendo T = 16, en la siguiente lectura (lín.32), read() sólo copia en b los dos caracteres restantes
6. Una última lectura de la pipe, al estar cerrada, devuelve 0
i. b no modifica su contenido
p[1]
pipe
p[0]
EOF
Proceso2
Código ejecutable
22 // Se asume que la pipe está creada
23 // int p[2] ;
24 // pipe(p);
25
26 #define T 16
27 int leidos;
28 char c;
29 char b[T];
30 leidos = read(p[0], &c, 1);
31 leidos = read(p[0], b, T);
32 leidos = read(p[0], b, T);
33 close(p[0]);
Montículo (heap)
##############################
...
b[2] #basura#
b[1] 'c'
b[0] 'b'
leidos 0
c 'a'
p[0]
Pila (stack)

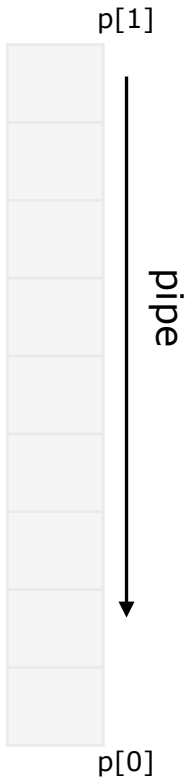
Ejemplo 2: read_n/write_n

Emisor

```
15  // Se asume que la pipe está creada
16  // int p[2] ;
17  // pipe(p);
18
19  #define T 16
20  char b[T];
21  int leidos, longitud;
22
23  // Se asume que el usuario introduce "hola"
24  leidos   = read(0, b, T);
25  // Se quita el \n
26  longitud = leidos - 1;
27  // Se envía la longitud de manenera protegida
28  // ya que se sabe la cantidad de bytes a enviar
29  if (write_n(p[1], &longitud, sizeof(int)) != sizeof(int)) {
30      perror("write longitud");
31      exit(-1);
32  }
33  // Se envía la cadena de manera protegida ya que
34  // se sabe la cantidad de datos a enviar
35  if (write_n(p[1], b, longitud) != longitud) {
36      perror("write b");
37      exit(-1);
38  }
39
40  close(p[1]);
```

Receptor

```
43  // Se asume que la pipe está creada
44  // int p[2] ;
45  // pipe(p);
46
47  #define T 16
48  int longitud;
49  char b[T];
50  // Se lee la longitud de forma protegida, porque
51  // se sabe la cantidad de bytes que se han de recibir
52  if (read_n(p[0], &longitud, sizeof(int)) != sizeof(int)) {
53      perror("read_n longitud");
54      exit(-1);
55  }
56  // Se lee la cadena de forma protegida, porque ahora
57  // también se sabe la cantidad de bytes que la componen
58  if (read_n(p[0], b, longitud) != longitud) {
59      perror("read_n b");
60      exit(-1);
61  }
62
63  close(p[0]);
```



1. Se asume que la tubería está creada, siendo **Emisor** el que **escribe** y **Receptor** el que **lee**
2. Mientras que **Emisor** no escriba, **Receptor** está **bloqueado** (línea 52)

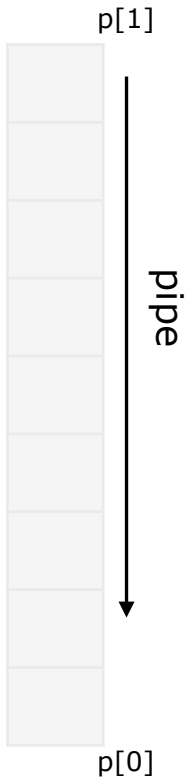
Ejemplo 2: read_n/write_n

Emisor

```
15  // Se asume que la pipe está creada
16  // int p[2] ;
17  // pipe(p);
18
19  #define T 16
20  char b[T];
21  int leidos, longitud;
22
23  // Se asume que el usuario introduce "hola"
24  leidos   = read(0, b, T);
25  // Se quita el \n
26  longitud = leidos - 1;
27  // Se envía la longitud de manenera protegida
28  // ya que se sabe la cantidad de bytes a enviar
29  if (write_n(p[1], &longitud, sizeof(int)) != sizeof(int)) {
30      perror("write longitud");
31      exit(-1);
32  }
33  // Se envía la cadena de manera protegida ya que
34  // se sabe la cantidad de datos a enviar
35  if (write_n(p[1], b, longitud) != longitud) {
36      perror("write b");
37      exit(-1);
38  }
39
40  close(p[1]);
```

Receptor

```
43  // Se asume que la pipe está creada
44  // int p[2] ;
45  // pipe(p);
46
47  #define T 16
48  int longitud;
49  char b[T];
50  // Se lee la longitud de forma protegida, porque
51  // se sabe la cantidad de bytes que se han de recibir
52  if (read_n(p[0], &longitud, sizeof(int)) != sizeof(int)) {
53      perror("read_n longitud");
54      exit(-1);
55  }
56  // Se lee la cadena de forma protegida, porque ahora
57  // también se sabe la cantidad de bytes que la componen
58  if (read_n(p[0], b, longitud) != longitud) {
59      perror("read_n b");
60      exit(-1);
61  }
62
63  close(p[0]);
```



1. Se asume que la tubería está creada, siendo **Emisor** el que **escribe** y **Receptor** el que **lee**
2. Mientras que **Emisor** no escriba, **Receptor** está **bloqueado** (línea 52)
3. El **Emisor** lee de teclado con **read()** y se asume que el usuario introduce "**abc**"
   1. No se puede usar **read_n()** porque no se sabe la cantidad de bytes de antemano
   2. leidos = 4 ("**abc\n**")

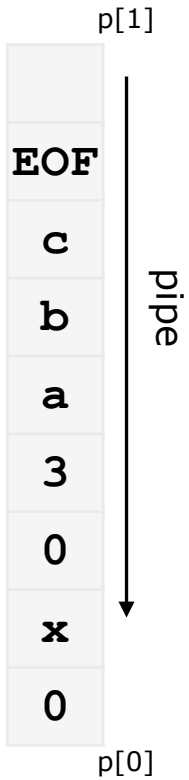
Ejemplo 2: read_n/write_n

Emisor

```
15  // Se asume que la pipe está creada
16  // int p[2] ;
17  // pipe(p);
18
19  #define T 16
20  char b[T];
21  int leidos, longitud;
22
23  // Se asume que el usuario introduce "hola"
24  leidos   = read(0, b, T);
25  // Se quita el \n
26  longitud = leidos - 1;
27  // Se envía la longitud de manenera protegida
28  // ya que se sabe la cantidad de bytes a enviar
29  if (write_n(p[1], &longitud, sizeof(int)) != sizeof(int)) {
30      perror("write longitud");
31      exit(-1);
32  }
33  // Se envía la cadena de manera protegida ya que
34  // se sabe la cantidad de datos a enviar
35  if (write_n(p[1], b, longitud) != longitud) {
36      perror("write b");
37      exit(-1);
38  }
39
40  close(p[1]);
```

Receptor

```
43  // Se asume que la pipe está creada
44  // int p[2] ;
45  // pipe(p);
46
47  #define T 16
48  int longitud;
49  char b[T];
50  // Se lee la longitud de forma protegida, porque
51  // se sabe la cantidad de bytes que se han de recibir
52  if (read_n(p[0], &longitud, sizeof(int)) != sizeof(int)) {
53      perror("read_n longitud");
54      exit(-1);
55  }
56  // Se lee la cadena de forma protegida, porque ahora
57  // también se sabe la cantidad de bytes que la componen
58  if (read_n(p[0], b, longitud) != longitud) {
59      perror("read_n b");
60      exit(-1);
61  }
62
63  close(p[0]);
```



1. Se asume que la tubería está creada, siendo **Emisor** el que **escribe** y **Receptor** el que **lee**
2. Mientras que **Emisor** no escriba, **Receptor** está **bloqueado** (línea 52)
3. El **Emisor** lee de teclado con **read()** y se asume que el usuario introduce "**abc**"
4. Envía el mensaje en formato longitud + cadena y cierra. Se asume que el **entero 3** usa **4 bytes**: **0x03**
    1. Se puede usar un **write_n()** porque se sabe la cantidad de datos a enviar en cada caso
    2. Se comprueba en el propio envío si se **escribe lo esperado**
    3. Por ejemplo, si **write_n()** no devuelve 4 o 3 bytes en las ambas escrituras es que ha habido algún **error** (líneas 29 y 35)
    4. Nótese que no hace falta el **\0** para nada

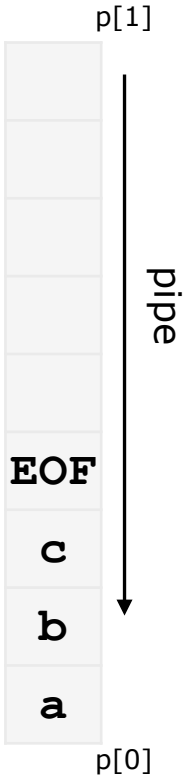
Ejemplo 2: read_n/write_n

Emisor

```
15  // Se asume que la pipe está creada
16  // int p[2] ;
17  // pipe(p);
18
19  #define T 16
20  char b[T];
21  int leidos, longitud;
22
23  // Se asume que el usuario introduce "hola"
24  leidos   = read(0, b, T);
25  // Se quita el \n
26  longitud = leidos - 1;
27  // Se envía la longitud de manenera protegida
28  // ya que se sabe la cantidad de bytes a enviar
29  if (write_n(p[1], &longitud, sizeof(int)) != sizeof(int)) {
30      perror("write longitud");
31      exit(-1);
32  }
33  // Se envía la cadena de manera protegida ya que
34  // se sabe la cantidad de datos a enviar
35  if (write_n(p[1], b, longitud) != longitud) {
36      perror("write b");
37      exit(-1);
38  }
39
40  close(p[1]);
```

Receptor

```
43  // Se asume que la pipe está creada
44  // int p[2] ;
45  // pipe(p);
46
47  #define T 16
48  int longitud;
49  char b[T];
50  // Se lee la longitud de forma protegida, porque
51  // se sabe la cantidad de bytes que se han de recibir
52  if (read_n(p[0], &longitud, sizeof(int)) != sizeof(int)) {
53      perror("read_n longitud");
54      exit(-1);
55  }
56  // Se lee la cadena de forma protegida, porque ahora
57  // también se sabe la cantidad de bytes que la componen
58  if (read_n(p[0], b, longitud) != longitud) {
59      perror("read_n b");
60      exit(-1);
61  }
62
63  close(p[0]);
```



1. Se asume que la tubería está creada, siendo **Emisor** el que **escribe** y **Receptor** el que **lee**
2. Mientras que **Emisor** no escriba, **Receptor** está **bloqueado** (línea 52)
3. El **Emisor** lee de teclado con **read()** y se asume que el usuario introduce "**abc**"
4. Envía el mensaje en formato longitud + cadena y cierra.
5. El proceso **Receptor** pasa a ejecutar, pero sabe la cantidad de bytes debe leer, por lo que puede hacer una lectura protegida del entero
    1. La información desaparece de la pipe
    2. A la vez se realiza la recepción y se comprueba si **se lee lo esperado**
    3. Si **read_n()** no devuelve los 4 bytes de un entero, entonces ha habido algún **error** y salimos (líneas 43 y 54)
    4. La variable **longitud** en el receptor tiene valor **3**, es decir, el tamaño de la cadena

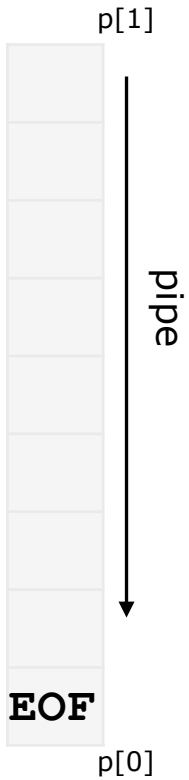
Ejemplo 2: read_n/write_n

Emisor

```
15  // Se asume que la pipe está creada
16  // int p[2] ;
17  // pipe(p);
18
19  #define T 16
20  char b[T];
21  int leidos, longitud;
22
23  // Se asume que el usuario introduce "hola"
24  leidos   = read(0, b, T);
25  // Se quita el \n
26  longitud = leidos - 1;
27  // Se envía la longitud de manenera protegida
28  // ya que se sabe la cantidad de bytes a enviar
29  if (write_n(p[1], &longitud, sizeof(int)) != sizeof(int)) {
30      perror("write longitud");
31      exit(-1);
32  }
33  // Se envía la cadena de manera protegida ya que
34  // se sabe la cantidad de datos a enviar
35  if (write_n(p[1], b, longitud) != longitud) {
36      perror("write b");
37      exit(-1);
38  }
39
40  close(p[1]);
```

Receptor

```
43  // Se asume que la pipe está creada
44  // int p[2] ;
45  // pipe(p);
46
47  #define T 16
48  int longitud;
49  char b[T];
50  // Se lee la longitud de forma protegida, porque
51  // se sabe la cantidad de bytes que se han de recibir
52  if (read_n(p[0], &longitud, sizeof(int)) != sizeof(int)) {
53      perror("read_n longitud");
54      exit(-1);
55  }
56  // Se lee la cadena de forma protegida, porque ahora
57  // también se sabe la cantidad de bytes que la componen
58  if (read_n(p[0], b, longitud) != longitud) {
59      perror("read_n b");
60      exit(-1);
61  }
62
63  close(p[0]);
```



1. Se asume que la tubería está creada, siendo **Emisor** el que **escribe** y **Receptor** el que **lee**
2. Mientras que **Emisor** no escriba, **Receptor** está **bloqueado** (línea 52)
3. El **Emisor** lee de teclado con **read()** y se asume que el usuario introduce "**abc**"
4. Envía el mensaje en formato longitud + cadena y cierra.
5. El proceso **Receptor** pasa a ejecutar, pero sabe o la cantidad de bytes debe leer, por lo que puede hacer una lectura protegida del entero
6. El **Receptor**, como ya sabe la longitud de la cadena, vuelve a hacer otra lectura protegida con la cantidad de datos exactos
    1. Los bytes de la cadena desaparecen de la pipe
    2. No hace falta detectar el cierre, puesto que ambos saben la cantidad de bytes a intercambiar

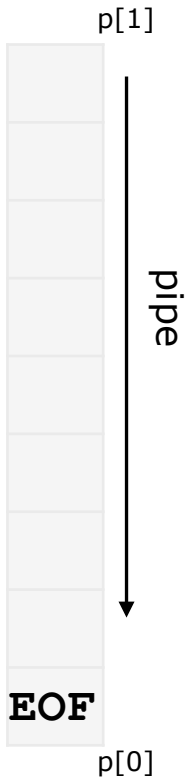
Ejemplo 2: read_n/write_n

Emisor

```
15  // Se asume que la pipe está creada
16  // int p[2] ;
17  // pipe(p);
18
19  #define T 16
20  char b[T];
21  int leidos, longitud;
22
23  // Se asume que el usuario introduce "hola"
24  leidos   = read(0, b, T);
25  // Se quita el \n
26  longitud = leidos - 1;
27  // Se envía la longitud de manenera protegida
28  // ya que se sabe la cantidad de bytes a enviar
29  if (write_n(p[1], &longitud, sizeof(int)) != sizeof(int)) {
30      perror("write longitud");
31      exit(-1);
32  }
33  // Se envía la cadena de manera protegida ya que
34  // se sabe la cantidad de datos a enviar
35  if (write_n(p[1], b, longitud) != longitud) {
36      perror("write b");
37      exit(-1);
38  }
39
40  close(p[1]);
```

Receptor

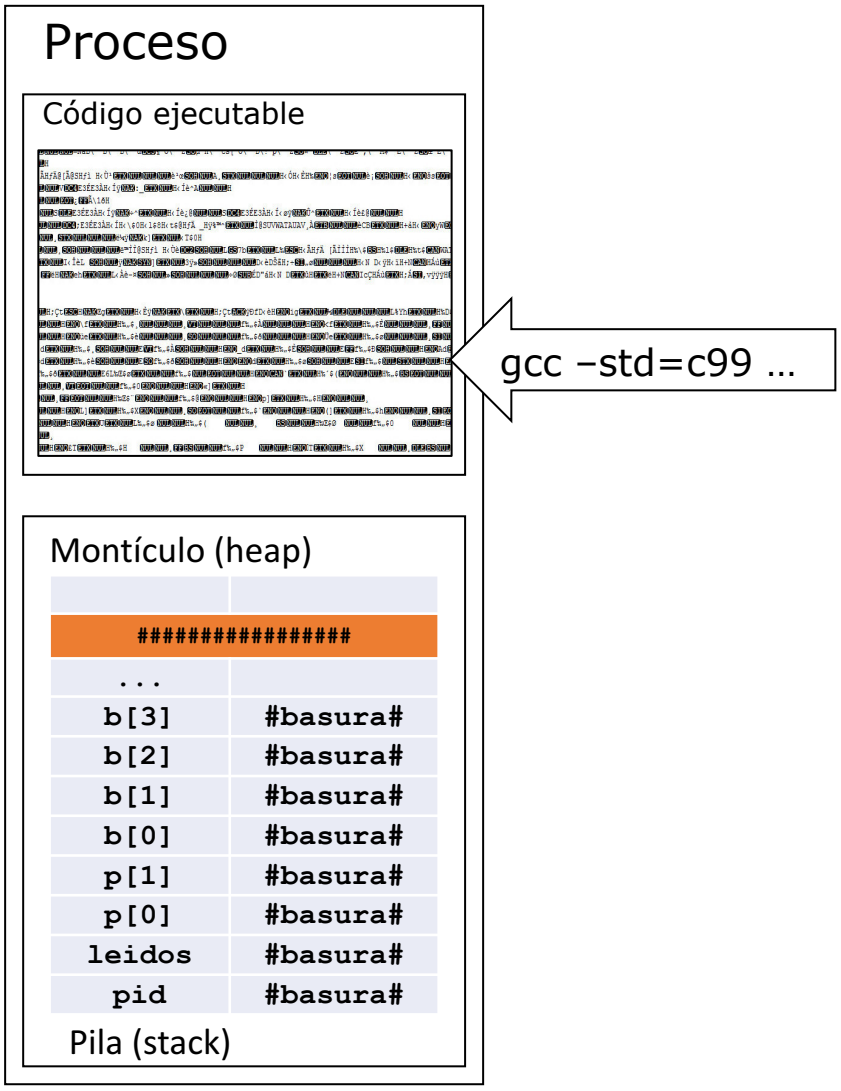
```
43  // Se asume que la pipe está creada
44  // int p[2] ;
45  // pipe(p);
46
47  #define T 16
48  int longitud;
49  char b[T];
50  // Se lee la longitud de forma protegida, porque
51  // se sabe la cantidad de bytes que se han de recibir
52  if (read_n(p[0], &longitud, sizeof(int)) != sizeof(int)) {
53      perror("read_n longitud");
54      exit(-1);
55  }
56  // Se lee la cadena de forma protegida, porque ahora
57  // también se sabe la cantidad de bytes que la componen
58  if (read_n(p[0], b, longitud) != longitud) {
59      perror("read_n b");
60      exit(-1);
61  }
62
63  close(p[0]);
```



1. Se asume que la tubería está creada, siendo **Emisor** el que **escribe** y **Receptor** el que **lee**
2. Mientras que **Emisor** no escriba, **Receptor** está **bloqueado** (línea 52)
3. El **Emisor** lee de teclado con **read()** y se asume que el usuario introduce "**abc**"
4. Envía el mensaje en formato longitud + cadena y cierra.
5. El proceso **Receptor** pasa a ejecutar, pero sabe o la cantidad de bytes debe leer, por lo que puede hacer una lectura protegida del entero
6. El **Receptor**, como ya sabe la longitud de la cadena, vuelve a hacer otra lectura protegida con la cantidad de datos exactos

1. Fijaos que se ha podido estructurar la información del canal para un intercambio protegido: se ha definido un **protocolo de comunicaciones** entre emisor y receptor

Ejemplo 3




```
 7  #define T 4
 8  int main(int argc, char * argv[]) {
 9      pid_t pid;
10      int leidos, p[2];
11      char b[T];
12
13      pipe(p);
14
15      if ((pid = fork()) == 0) {
16          close(p[1]);
17          while((leidos = read(p[0],b,T)) > 0) {
18              write(1, b, leidos);
19          }
20          close(p[0]);
21          exit(0);
22      } else {
23          close(p[0]);
24          write(p[1], "abcde\n", 5);
25          close(p[1]);
26          wait(0);
27      }
28  }
```

1. Se crea la tubería (línea 13)
2. Se clona el proceso con **fork**(): los descriptores se comparten
3. Se cierran los descriptores que no se utilizan
4. Cada proceso realiza sus tareas
   1. Padre: envía una cadena al hijo
   2. Hijo: lee la cadena en bloques de 4 bytes y las imprime por pantalla

Ejemplo 3

Proceso Padre

Código ejecutable

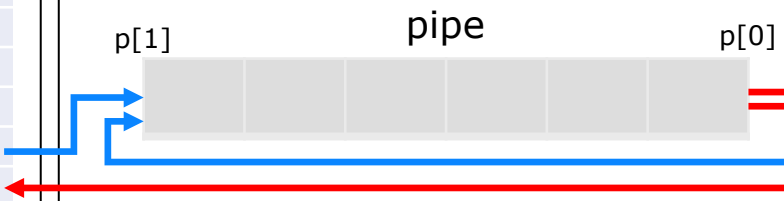
```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	#basura#
b[2]	#basura#
b[1]	#basura#
b[0]	#basura#
p[1]	
p[0]	
leidos	#basura#
pid	2119

Pila (stack)

1. Una vez creada la pipe, y clonado el proceso, empieza la ejecución concurrente
   i. Padre: línea 23
   ii. Hijo: línea 16
2. El proceso de clonación copia todas las variables, entre ellas los descriptores de la pipe, que ahora se comparten
   i. En **azul**, los extremos de escritura
   ii. En **rojo**, los de lectura
   iii. Nótese el valor de la variable **pid**, que vale **2119** en el padre (el pid del hijo) y **0** en el hijo



Proceso Hijo

Código ejecutable

```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	#basura#
b[2]	#basura#
b[1]	#basura#
b[0]	#basura#
p[1]	
p[0]	
leidos	#basura#
pid	0

Pila (stack)

Ejemplo 3

Proceso Padre

Código ejecutable

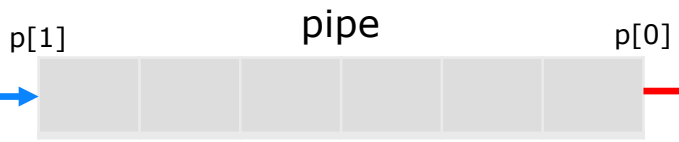
```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	#basura#
b[2]	#basura#
b[1]	#basura#
b[0]	#basura#
p[1]	
p[0]	#Cerrado#
leidos	#basura#
pid	2119

Pila (stack)

1. Una vez creada la pipe, y clonado el proceso, empieza la ejecución concurrente
2. El proceso de clonación copia todas las variables, entre ellas los descriptores de la pipe, que ahora se comparten
3. Ambos procesos cierran los extremos que no usan, lo que de termina la dirección del flujo de datos de la pipe: **padre → hijo**



Proceso Hijo

Código ejecutable

```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	#basura#
b[2]	#basura#
b[1]	#basura#
b[0]	#basura#
p[1]	#Cerrado#
p[0]	
leidos	#basura#
pid	0

Pila (stack)

Ejemplo 3

Proceso Padre

Código ejecutable

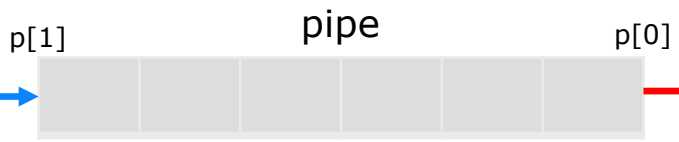
```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	#basura#
b[2]	#basura#
b[1]	#basura#
b[0]	#basura#
p[1]	
p[0]	#Cerrado#
leidos	#basura#
pid	2119

Pila (stack)

1. Una vez creada la pipe, y clonado el proceso, empieza la ejecución concurrente
2. El proceso de clonación copia todas las variables, entre ellas los descriptores de la pipe, que ahora se comparten
3. Ambos procesos cierran los extremos que no usan, lo que de termina la dirección del flujo de datos de la pipe: **padre → hijo**
4. Mientras el padre no escriba, el **read() bloquea** al hijo (línea 17)



Proceso Hijo

Código ejecutable

```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	#basura#
b[2]	#basura#
b[1]	#basura#
b[0]	#basura#
p[1]	#Cerrado#
p[0]	
leidos	#basura#
pid	0

Pila (stack)

Ejemplo 3

Proceso Padre

Código ejecutable

```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	#basura#
b[2]	#basura#
b[1]	#basura#
b[0]	#basura#
p[1]	
p[0]	#Cerrado#
leidos	#basura#
pid	2119

Pila (stack)

1. Una vez creada la pipe, y clonado el proceso, empieza la ejecución concurrente
2. El proceso de clonación copia todas las variables, entre ellas los descriptores de la pipe, que ahora se comparten
3. Ambos procesos cierran los extremos que no usan, lo que de termina la dirección del flujo de datos de la pipe: **padre → hijo**
4. Mientras el padre no escriba, el **read() bloquea** al hijo (línea 17)
5. El padre escribe 5 bytes, cierra la pipe y espera al hijo
    i. Aunque cierre, **la información se mantiene** en la pipe puesto que aún está abierto el extremo de lectura

p[1] pipe p[0]

EOF	'e'	'd'	'c'	'b'	'a'

Proceso Hijo

Código ejecutable

```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	#basura#
b[2]	#basura#
b[1]	#basura#
b[0]	#basura#
p[1]	#Cerrado#
p[0]	
leidos	#basura#
pid	0

Pila (stack)

Ejemplo 3

Proceso Padre

Código ejecutable

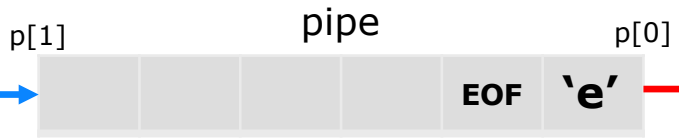
```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	#basura#
b[2]	#basura#
b[1]	#basura#
b[0]	#basura#
p[1]	
p[0]	#Cerrado#
leidos	#basura#
pid	2119

Pila (stack)

1. Una vez creada la pipe, y clonado el proceso, empieza la ejecución concurrente
2. El proceso de clonación copia todas las variables, entre ellas los descriptores de la pipe, que ahora se comparten
3. Ambos procesos cierran los extremos que no usan, lo que de termina la dirección del flujo de datos de la pipe: **padre → hijo**
4. Mientras el padre no escriba, el **read() bloquea** al hijo (línea 17)
5. El padre escribe 5 bytes, cierra la pipe (**EOF**) y espera al hijo
6. El hijo comienza la lectura en bloques de **4 bytes**, e imprime por pantalla **"abcd"** en la primera iteración
    i. La pipe se actualiza



Proceso Hijo

Código ejecutable

```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	'd'
b[2]	'c'
b[1]	'b'
b[0]	'a'
p[1]	#Cerrado#
p[0]	
leidos	4
pid	0

Pila (stack)

Ejemplo 3

Proceso Padre

Código ejecutable

```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	#basura#
b[2]	#basura#
b[1]	#basura#
b[0]	#basura#
p[1]	
p[0]	#Cerrado#
leidos	#basura#
pid	2119

Pila (stack)

1. Una vez creada la pipe, y clonado el proceso, empieza la ejecución concurrente
2. El proceso de clonación copia todas las variables, entre ellas los descriptores de la pipe, que ahora se comparten
3. Ambos procesos cierran los extremos que no usan, lo que de termina la dirección del flujo de datos de la pipe: **padre → hijo**
4. Mientras el padre no escriba, el **read() bloquea** al hijo (línea 17)
5. El padre escribe 5 bytes, cierra la pipe (**EOF**) y espera al hijo
6. El hijo comienza la lectura en bloques de **4 bytes**, e imprime por pantalla **"abcd"** en la primera iteración

p[1] pipe p[0]

7. En la segunda lectura de **4 bytes**, sólo lee **'e'**, que se refleja en **leidos = 1**
   1. La pipe se vacía
   2. Sólo se modifica la primera posición de **b**

Proceso Hijo

Código ejecutable

```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	'd'
b[2]	'c'
b[1]	'b'
b[0]	'e'
p[1]	#Cerrado#
p[0]	
leidos	1
pid	0

Pila (stack)

Ejemplo 3

Proceso Padre

Código ejecutable

```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	#basura#
b[2]	#basura#
b[1]	#basura#
b[0]	#basura#
p[1]	#Cerrado#
p[0]	#Cerrado#
leidos	#basura#
pid	2119

Pila (stack)

1. Una vez creada la pipe, y clonado el proceso, empieza la ejecución concurrente
2. El proceso de clonación copia todas las variables, entre ellas los descriptores de la pipe, que ahora se comparten
3. Ambos procesos cierran los extremos que no usan, lo que de termina la dirección del flujo de datos de la pipe: **padre → hijo**
4. Mientras el padre no escriba, el **read() bloquea** al hijo (línea 17)
5. El padre escribe 5 bytes, cierra la pipe (**EOF**) y espera al hijo
6. El hijo comienza la lectura en bloques de **4 bytes**, e imprime por pantalla **"abcd"** en la primera iteración
7. En la segunda lectura de **4 bytes**, sólo lee **'e'**, que se refleja en **leidos = 1**
8. Finalmente, el último **read**() **devuelve 0**
    i. Se cierran los extremos de la pipe y se devuelven sus recursos al sistema
    ii. Se hace una salida ordenada [exit() + wait()], y el padre devuelve también los recursos del hijo al sistema

Proceso Hijo

Código ejecutable

```
7  #define T 4
8  int main(int argc, char * argv[]) {
9      pid_t pid;
10     int leidos, p[2];
11     char b[T];
12
13     pipe(p);
14
15     if ((pid = fork()) == 0) {
16         close(p[1]);
17         while((leidos = read(p[0],b,T)) > 0) {
18             write(1, b, leidos);
19         }
20         close(p[0]);
21         exit(0);
22     } else {
23         close(p[0]);
24         write(p[1], "abcde\n", 5);
25         close(p[1]);
26         wait(0);
27     }
28 }
```

Montículo (heap)

################	
...	
b[3]	'd'
b[2]	'c'
b[1]	'b'
b[0]	'e'
p[1]	#Cerrado#
p[0]	#Cerrado#
leidos	0
pid	0

Pila (stack)

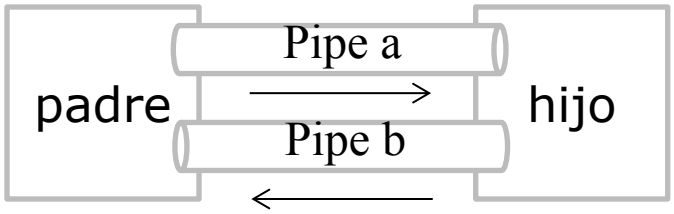
Ejemplo 4

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

#define T 256

int main(int argc, char * argv[]) {
    pid_t pid;
    int leidos, p[2];
    char b[T];

    //1.- Creamos la pipe para compartir
    //    descriptores entre padre e hijo
    if (pipe(p) < 0) {
        perror("pipe");
        exit(-1);
    }

    //2.- Clonamos
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(-1);
    } else if (pid == 0) {
        //3.1.- Cerramos el extremo que
        //      no se usa en el hijo: escritura
        if (close(p[1]) < 0) {
            perror("close");
            exit(-1);
        }

        //3.2.- Leemos de la pipe hasta que
        //      se cierre o falle
        while((leidos = read(p[0],b,T)) > 0) {
            write(1, b, leidos);
        }
        if (leidos < 0) {
            perror("read");
            exit(-1);
        }

        //3.3.- Cerramos el extremo de lectura
        if (close(p[0]) < 0) {
            perror("close");
            exit(-1);
        }

        //3.4.- Salimos ordenadamente
        exit(0);
    } else {
        //3.1.- Cerramos el extremo que
        //      no se usa en el padre: lectura
        if (close(p[0]) < 0) {
            perror("close");
            exit(-1);
        }

        //3.2.- Escribimos por la tubería
        //      comprobando que se escribe lo esperado
        if (write(p[1], "abcde\n", 5) != 5) {
            perror("write");
            exit(-1);
        }

        //3.4.- Cerramos el extremo de escritura
        if (close(p[1]) < 0) {
            perror("close");
            exit(-1);
        }

        //3.5.- Esperamos a nuestro hijo
        //      para devolver sus recursos al sistema
        wait(0);
    }
}
```

Comunicación bidireccional con pipes

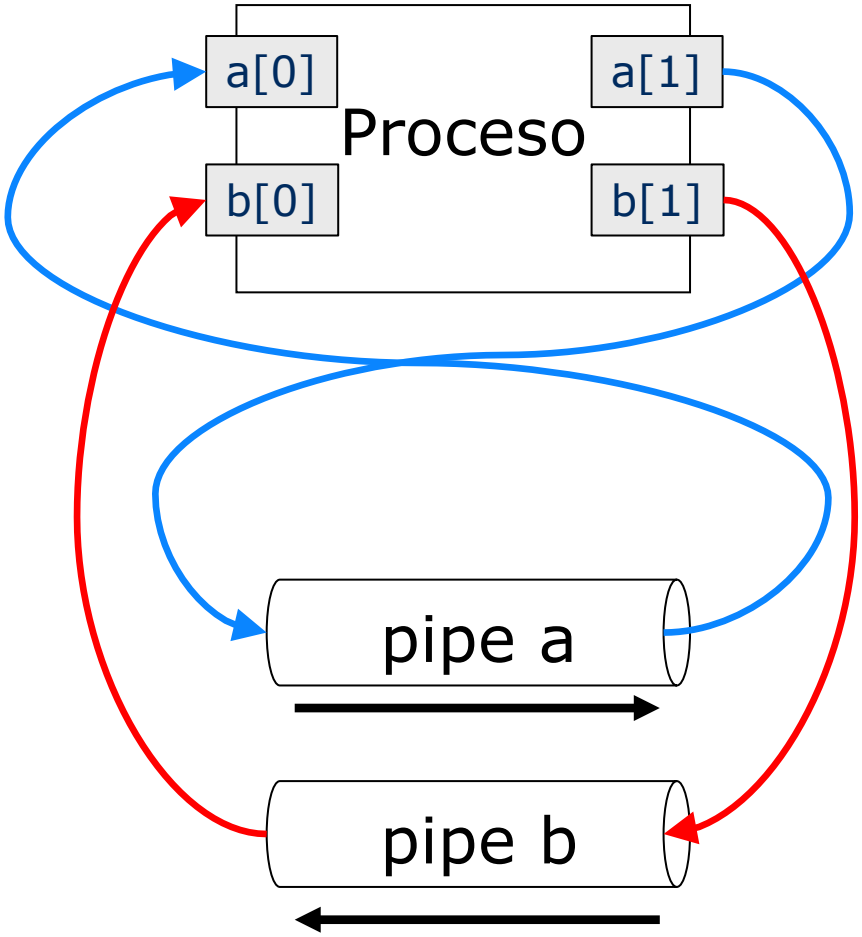
- Dos tuberías diferentes (a[2] y b[2]), una para cada sentido de la comunicación.
  - La pipe a[2] para la comunicación desde el padre al hijo.
  - La pipe b[2] para comunicarnos desde el hijo al padre.

- En cada proceso habrá que cerrar descriptores de ficheros diferentes.
  - En el padre:
    - el lado de lectura a[0]
    - el lado de escritura b[1]
  - En el hijo:
    - el lado de escritura a[1]
    - el lado de lectura b[0]



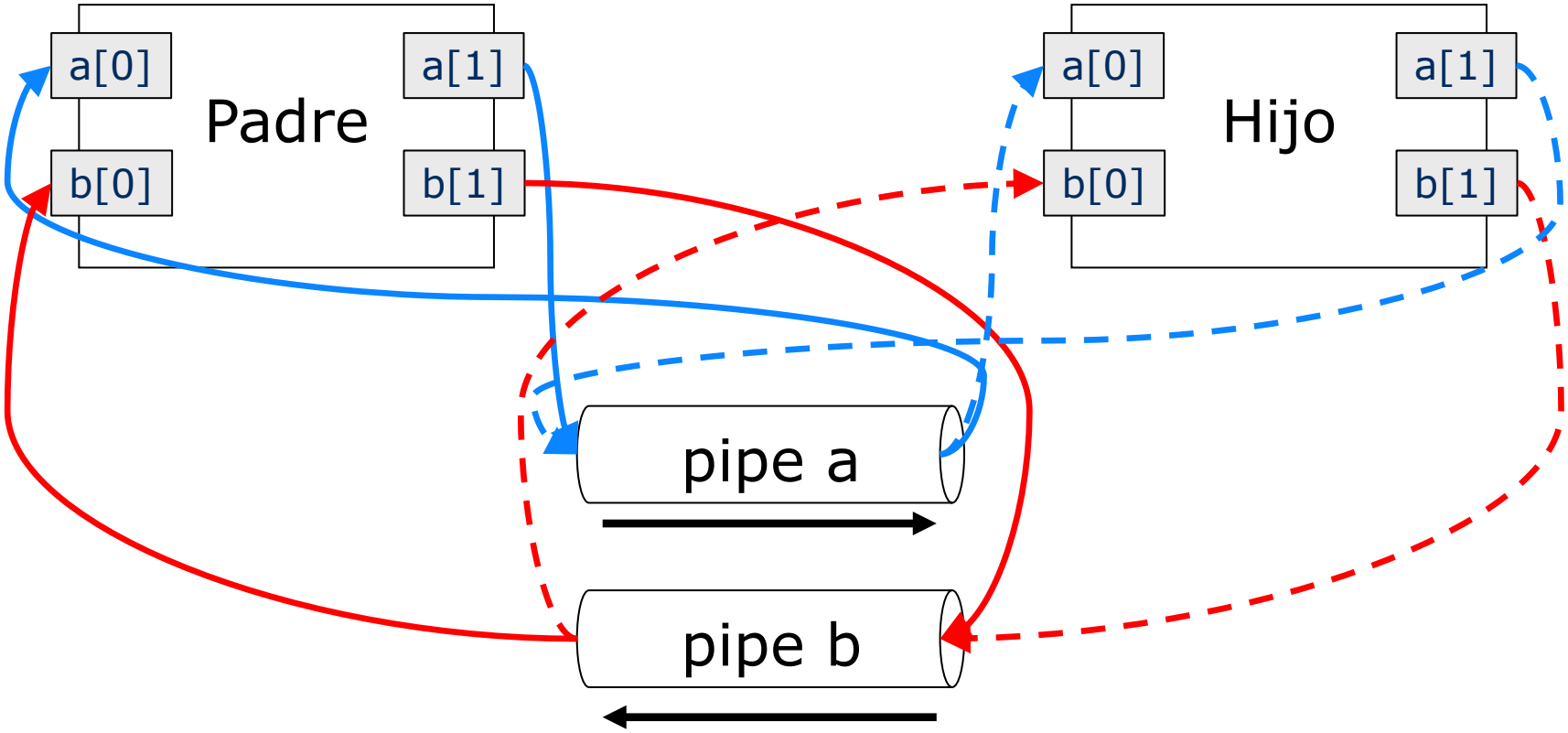
Comunicación bidireccional con pipes

- Después de crear las **pipes**, y antes del **fork**()
  - En **azul**, los extremos asociados a la **pipe a: padre → hijo**
  - En **rojo**, los extremos asociados a la **pipe b: hijo → padre**



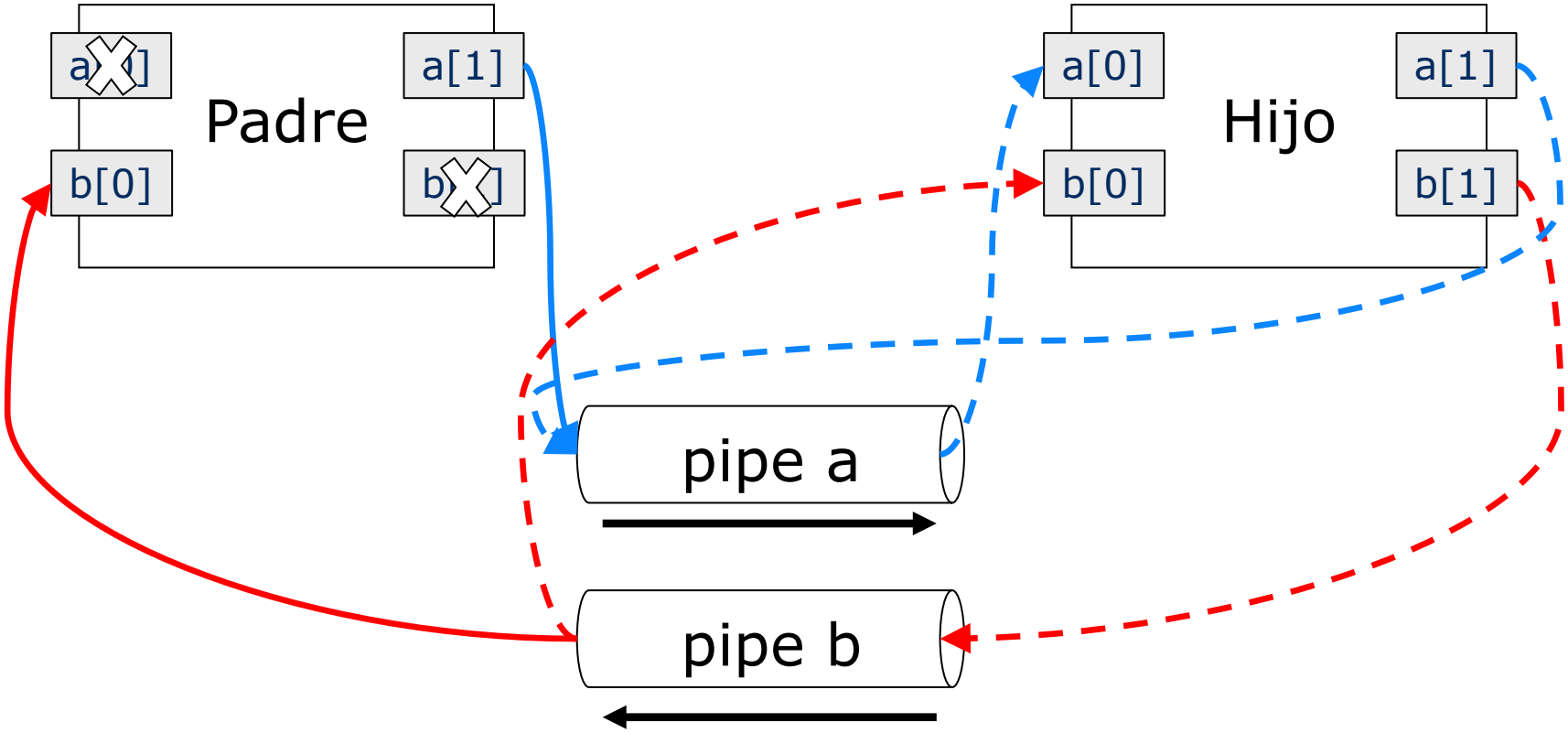
Comunicación bidireccional con pipes

- Después clonar con **fork**()
  - En **azul**, los extremos asociados a la **pipe a: padre → hijo**
  - En **rojo**, los extremos asociados a la **pipe b: hijo → padre**
  - En línea discontinua, los descriptores copiados tras la clonación



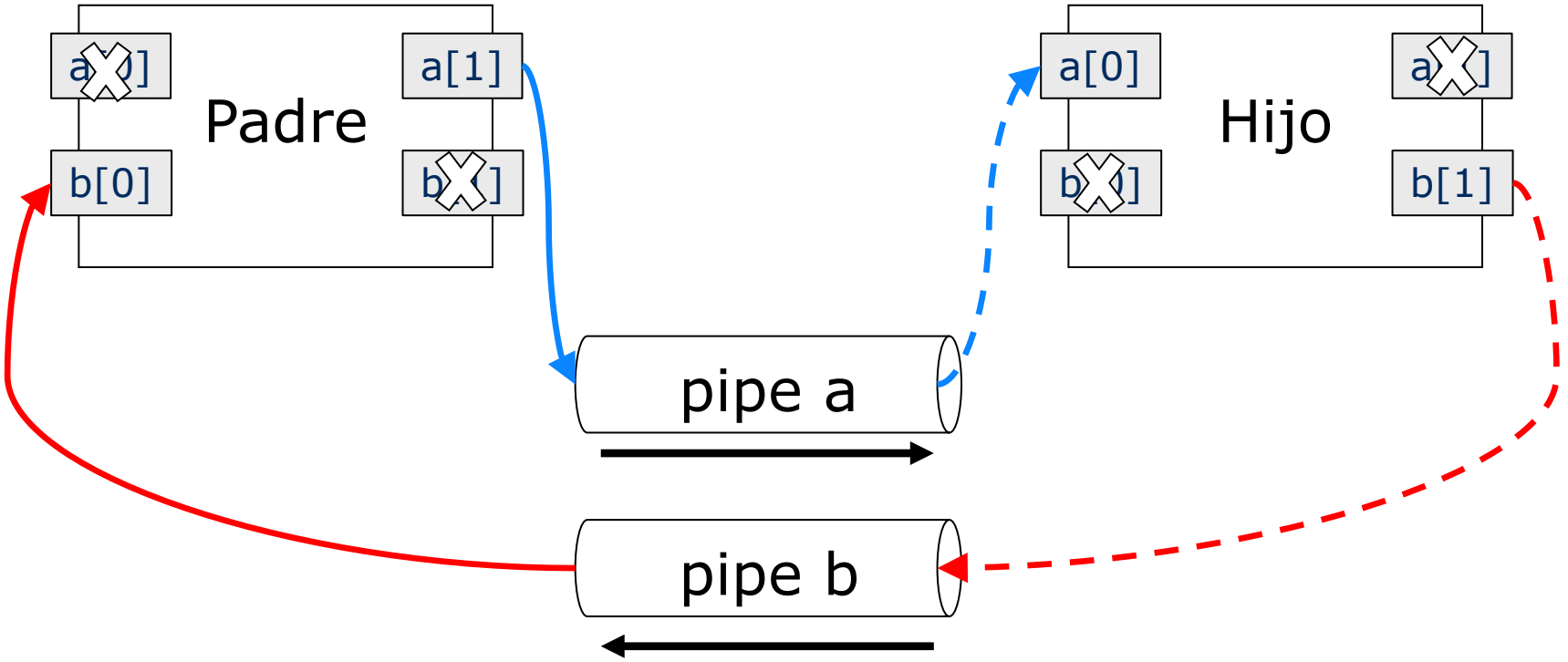
Comunicación bidireccional con pipes

- Después clonar con **fork**()
  - En **azul**, los extremos asociados a la **pipe a: padre → hijo**
  - En **rojo**, los extremos asociados a la **pipe b: hijo → padre**
  - En línea discontinua, los descriptores copiados tras la clonación
  - Padre cierra **a[0]**, **b[1]** para comunicación half-duplex padre → hijo



Comunicación bidireccional con pipes

- Después clonar con **fork**()
  - En **azul**, los extremos asociados a la **pipe a: padre → hijo**
  - En **rojo**, los extremos asociados a la **pipe b: hijo → padre**
  - En línea discontinua, los descriptores clonados
  - Padre cierra **a[0]**, **b[1]** para comunicación half-duplex padre → hijo
  - Hijo cierra **a[1]**, **b[0]** para comunicación half-duplex hijo → padre



Ejemplo de pipe 2

```
 9  int main(int argc, char *argv[]){
10      pid_t pid;
11      int a[2], b[2], readbytes;
12      char buffer[SIZE];
13      pipe(a);
14      pipe(b);
15      if ((pid = fork()) == 0) {// hijo
16          close(a[1]); /* cerramos el lado de escritura del pipe */
17          close(b[0]); /* cerramos el lado de lectura del pipe */
18          while ((readbytes = read(a[0], buffer, SIZE)) > 0)
19              write(1, buffer, readbytes);
20          close(a[0]);
21          strcpy(buffer, "Soy tu hijo hablandote por la otra tuberia.\n");
22          write(b[1], buffer, strlen(buffer));
23          close(b[1]);
24          exit(0);
25      } else { // padre
26          close(a[0]); /* cerramos el lado de lectura del pipe */
27          close(b[1]); /* cerramos el lado de escritura del pipe */
28          strcpy(buffer, "Soy tu padre hablandote por una tuberia.\n");
29          write(a[1], buffer, strlen(buffer));
30          close(a[1]);
31          while ((readbytes = read(b[0], buffer, SIZE)) > 0)
32              write(1, buffer, readbytes);
33          close(b[0]);
34          wait(0);
35      }
36  }
```