Fundamentos de los Sistemas Operativos (FSO)

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

Bloque Temático 3: Sistema de Archivos y E/S
Seminario Unidad Temática 7

SUT07: Llamadas al sistema Unix para archivos

fSO

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA

- **Objetivos**
  - Describir el concepto de **descriptor de archivos**
  - Comprender el **concepto de tabla de descriptores de archivos** y su utilidad
  - Estudiar las **llamadas al sistema** de Unix para trabajar con **archivos**
  - Manejar los mecanismo de **redireccionamiento de entrada y salida** de procesos Unix
  - Realizar la **comunicación** de procesos Unix con **tubos(“pipe”)**

Llamadas al sistema Unix para archivos

- **Contenido**
  - Archivos en Unix
  - Llamadas al sistema para archivos
  - Redirecciones y tubos
- **Bibliografía**
  - " Fundamentos de Sistemas Operativos", A. Silberschatz. P.B. Galvin, 7ªEd. Mc Graw Hill, ISBN: 968-444-310-2
  - "Operating System Concepts", A. Silberschatz. P.B. Galvin, 8ªEd., Wiley, ISBN: 978-0-470-12872-5
  - "UNIX Programación Práctica", Kay A. Robbins, Steven Robbins. Prentice Hall. ISBN 968-880-959-4
  - "UNIX Systems Programing ", Kay A. Robbins, Steven Robbins. Prentice Hall. ISBN 0-13-042411-0

- **Contenido**
  - **Archivos en Unix**
  - Llamadas al sistema para archivos
  - Redirecciones y tubos(“pipe”)
  - Llamadas para redirecciones y tubos
  - Ejemplos en C

Archivos en Unix

- **Tipo de archivos en Unix**
  - **Regular**: archivos convencionales de datos (programa, texto, ...) almacenados en memoria secundaria
  - **Directorio**: que asocia nombres a los archivos
  - **Tubos**: archivo sin nombre, con acceso secuencial, para comunicación entre procesos
  - **FIFO**: archivos con nombre, de acceso secuencial, para comunicación entre procesos
  - **Especial**: representa un dispositivo físico o ficticio del sistema
    - Dispositivo físico o ficticio de caracteres, como las consolas **`/dev/ttynn`** o **`/dev/pts/n`** o el sumidero ficticio **`/dev/null`**

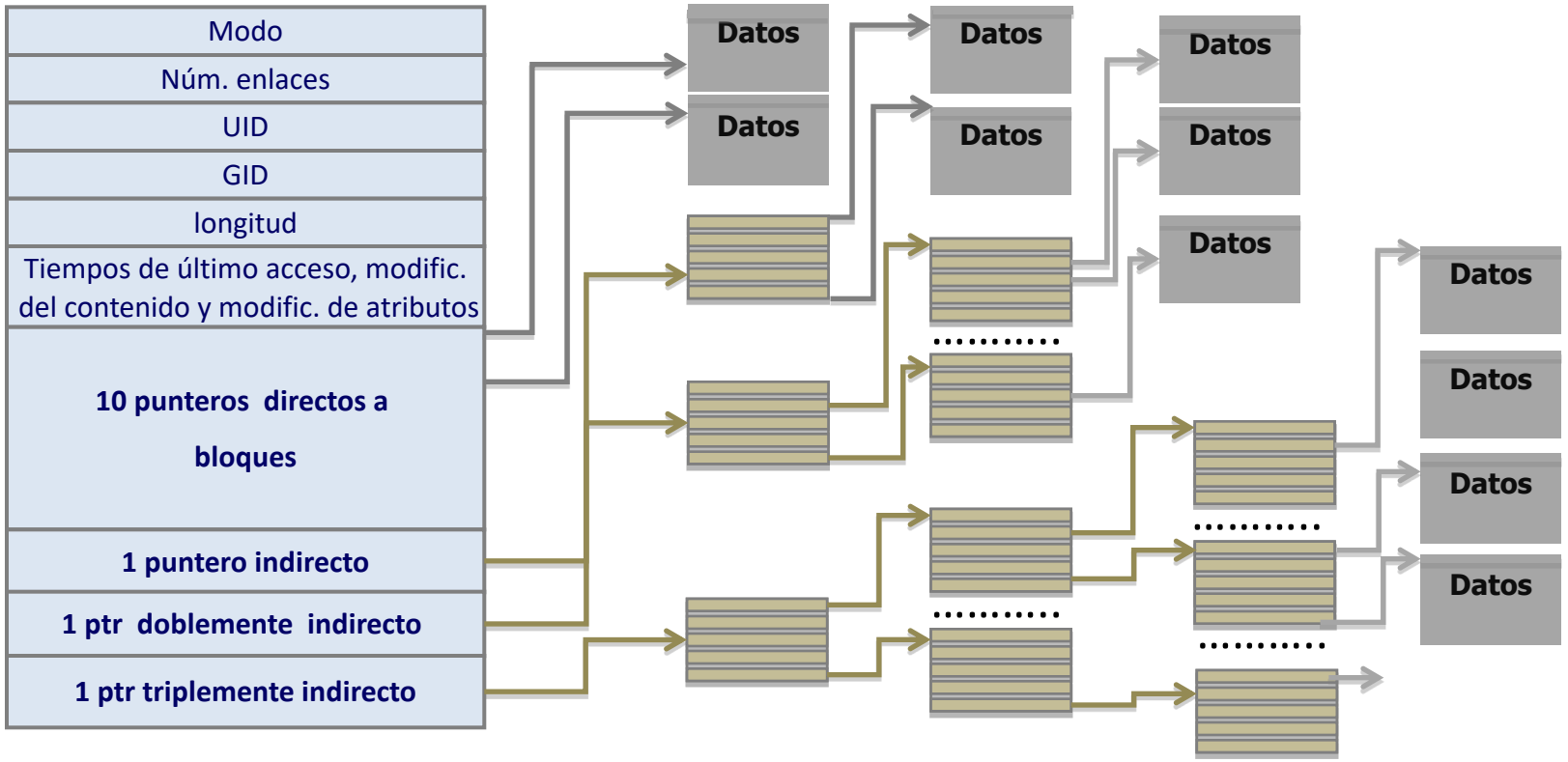
Nota: Desde el shell de UNIX, el tipo de archivo puede verse con la orden "$ls -la" ya que aparece codificado en la primera letra del listado

- archivo regular: marcado como '–'
- archivo directorio: marcado como 'd'
- archivos especiales: marcado con 'c' (carácter) o 'b' (bloques)
- archivos FIFO: marcado como 'p'

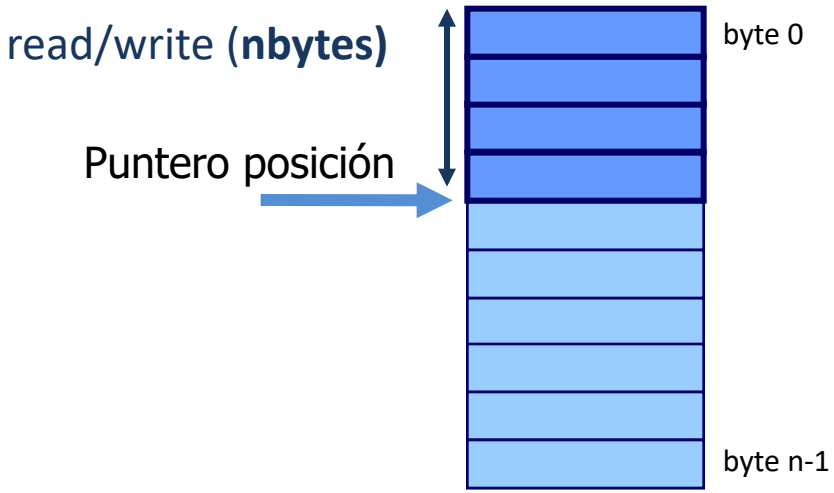
Archivos en Unix

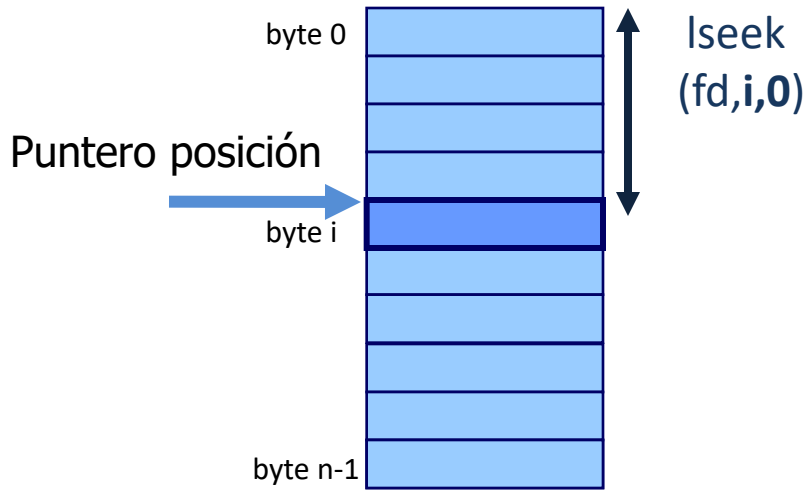
- **Atributos de los archivos Unix**
  - Tipo de archivo
  - Propietario (owner UID)
  - Grupo propietario (owner GID)
  - Permisos de acceso (permission bits, rwx)
  - Número de enlaces
  - Instantes de creación, último acceso y última modificación
  - Tamaño

- **Nodo-i (*i-node*)**
  - Estructura de datos del s.o para guardar todos los atributos del archivo excepto el nombre
    - Cada archivo en Unix se le asigna un nodo-i
    - Utiliza asignación indexada de bloques con punteros a bloques directos, indirectos, doblemente indirectos y triplemente indirectos.



- **Estructura de archivo en Unix**
  - Vector de bytes
- **Modo de acceso en Unix**
  - **Acceso secuencial** con las llamadas **read/write:**
    - **read/write (fd, buffer, nbytes)**
  - La llamada ***lseek*** permite realizar acceso directo especificando un offset desde principio, final de fichero (EOF) o posición actual.
    - **lseek (fd, offset, where)**





Archivos en Unix

- **Descriptores de archivos (** *file description (fd)* **)**
  - Para trabajar con un archivo hay que definir una sesión de trabajo con las llamadas ***open*** (abrir) ***y close*** (cerrar)

Abrir:	`fd = open(filename, mode)`
Acceder:	`read(fd,…), write(fd,…),`
	`lseek(fd,…), …`
Cerrar:	`close (fd)`

  - Un **descriptor de archivo** es un identificador abstracto del archivo, es particular para cada proceso
    - La llamada ***open*** devuelve un descriptor de archivo (fd)
    - Un descriptor es un número que identifica una posición en una tabla
    - Trabajar con descriptores de archivos hace más eficiente el acceso a los archivos, evita buscarlos en disco en cada operación

- **Descriptores de archivo** (*fd*)
  - **Semántica de apertura de archivo** (*open()*)
    - Buscar el archivo en la estructura de directorios y llevar sus atributos a la **tabla de aperturas del sistema** donde se registran algunos atributos adicionales como:
      - Puntero de posición
      - Número de aperturas
      - Ubicación de la información en disco
    - El contenido del archivo es llevado parcialmente a buffers en memoria
  - **Semántica de cerrar archivo** (*close()*)
    - Liberar la entrada correspondiente en la tabla de archivos abiertos

- **Tabla de aperturas del sistema versus tabla de descriptores de archivos**

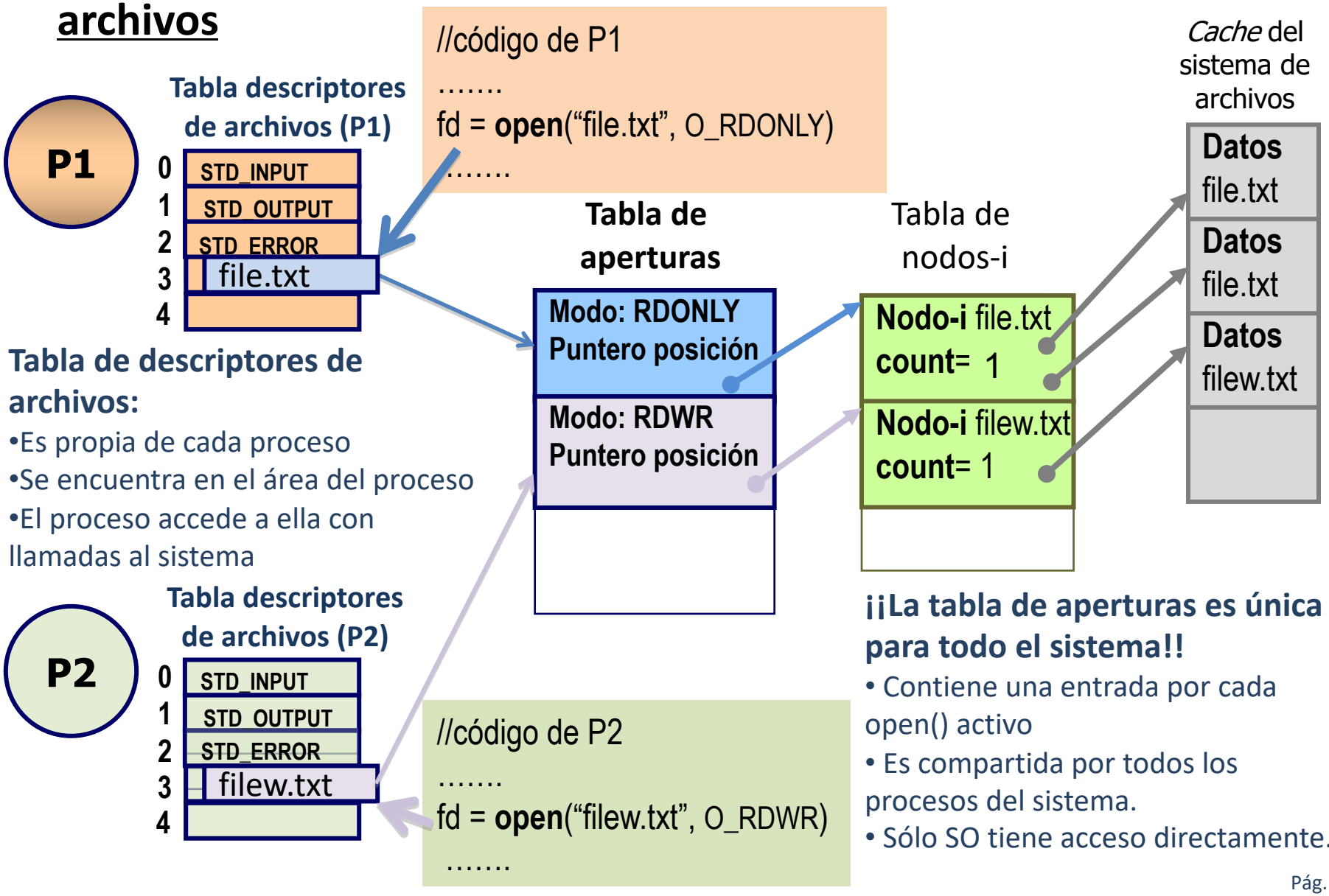


Tabla de descriptores de archivos:

- Es propia de cada proceso
- Se encuentra en el área del proceso
- El proceso accede a ella con llamadas al sistema

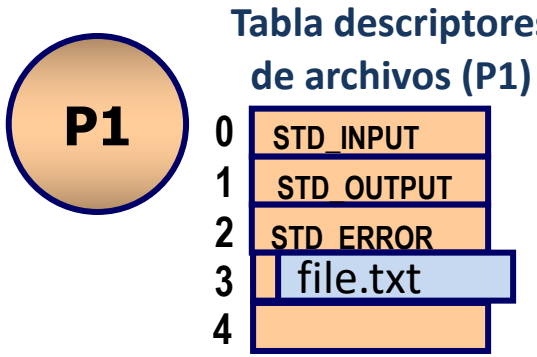
¡¡La tabla de aperturas es única para todo el sistema!!

- Contiene una entrada por cada open() activo
- Es compartida por todos los procesos del sistema.
- Sólo SO tiene acceso directamente.

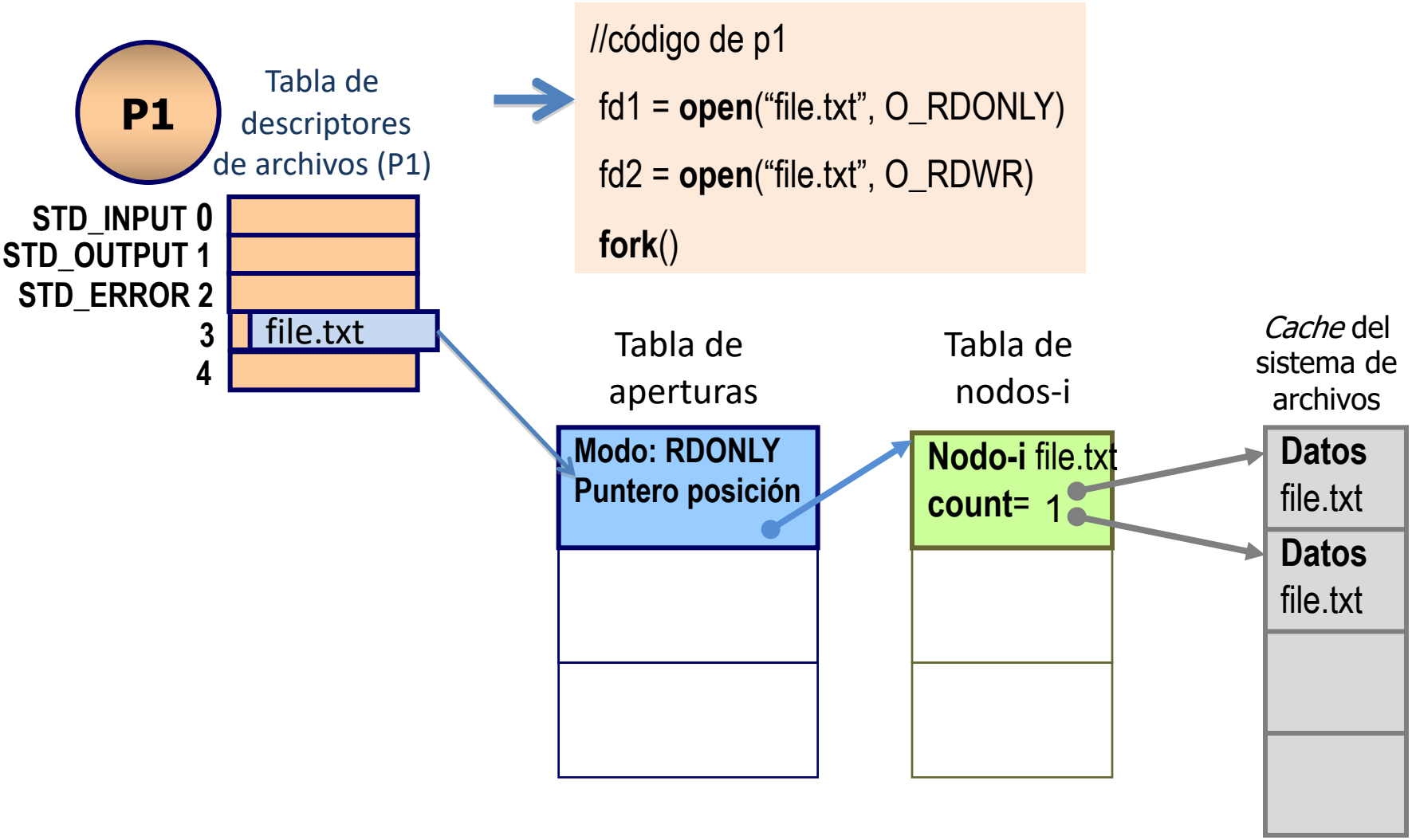
- **Descriptores para dispositivos de entrada/salida estándar**
  - En cada proceso los tres descriptores más bajos (0,1 y 2) de su tabla tienen nombre propio:

Descriptor (fd)	nombre en *<unistd.h>*	Dispositivo físico
0	STDIN_FILENO	entrada estándar o *stdin*
1	STDOUT_FILENO	salida estándar o *stdout*
2	STDERR_FILENO	salida de errores o *stderr*

  - Por omisión, estos descriptores están asociados a la consola
    - Es decir, están asociados a /dev/tty*n* o a /dev/ptn/*n*
    - Esta asociación se puede modificar mediante redirecciones y tubos
  - Ejemplos de Uso:
    - **Desde la biblioteca de C**: *scanf()* lee de la entrada estándar y *printf()* escribe en la salida estándar.
    - **Desde el Shell**: Las órdenes del Shell leen y escriben por la E/S estándar. La orden "ls" escribe el listado de archivos por la salida estándar y mensajes del tipo "No such file or directory" por la salida de errores

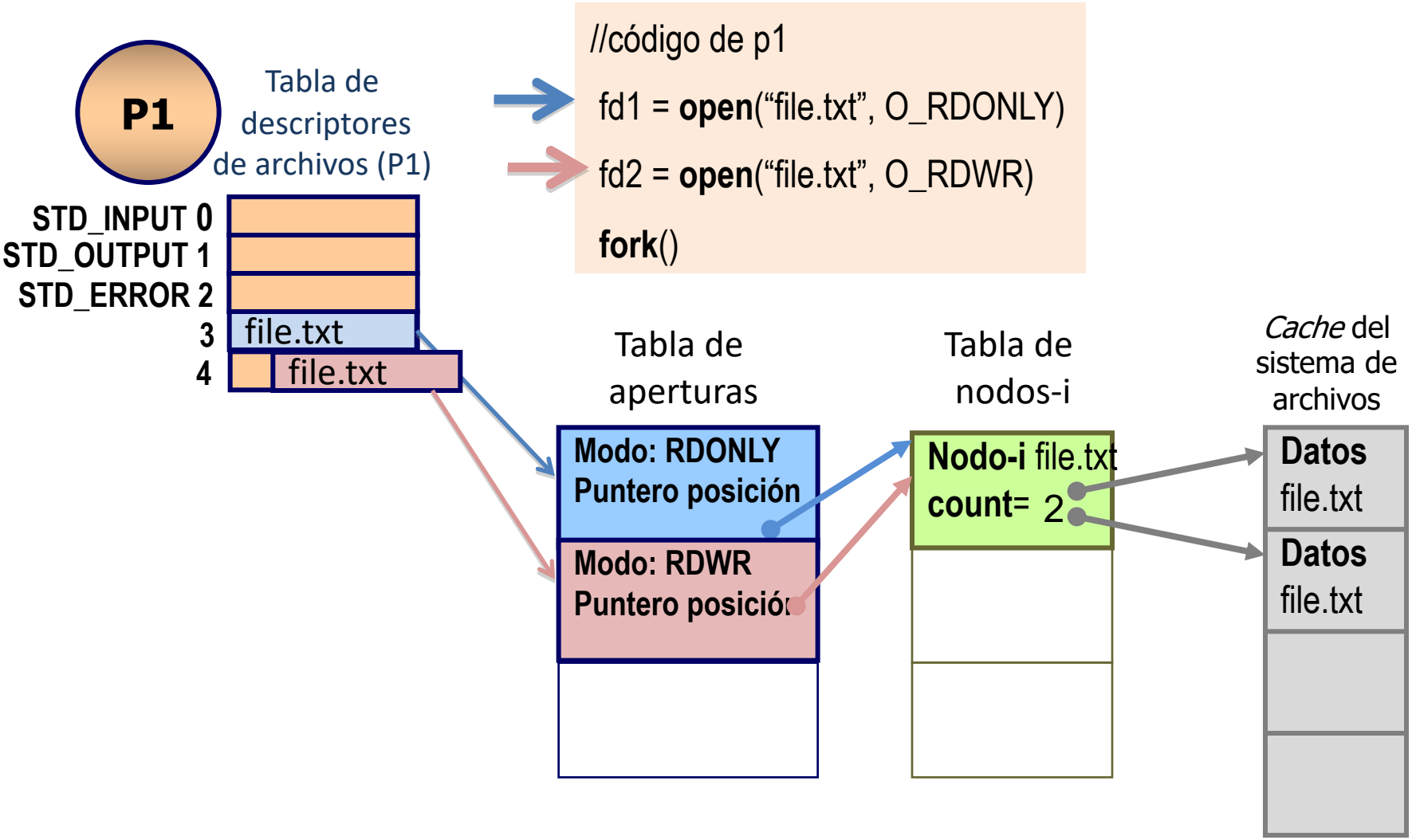


- **Herencia de tabla de descriptores de archivos**

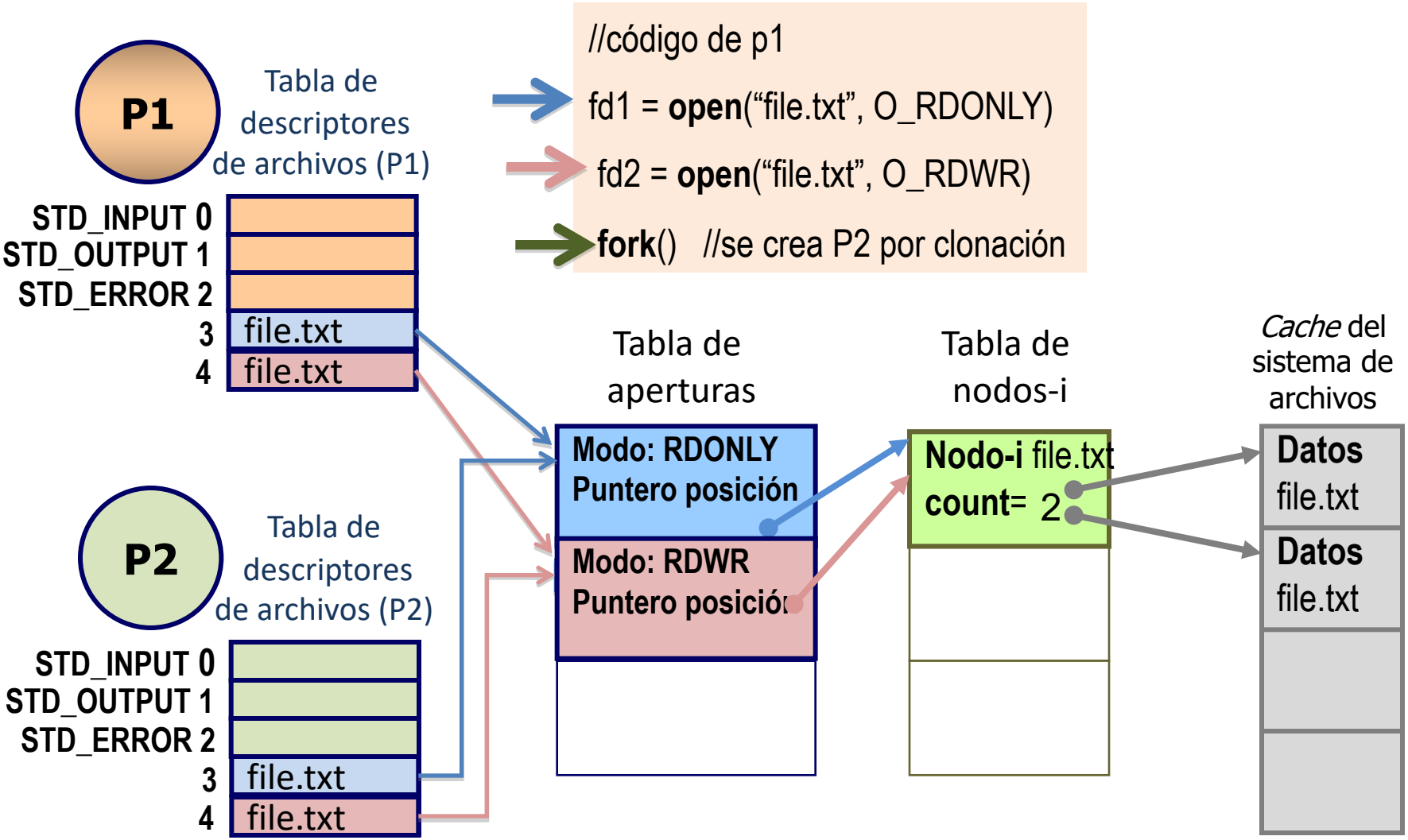


¡¡La tabla de aperturas es única para todo el sistema!!

- **Herencia de tabla de descriptores de archivos**



- **Herencia de tabla de descriptores de archivos**



¡¡Los descriptores de archivos abiertos son atributos heredables!!

- La tabla de descriptores se hereda entre padres e hijos pero no se comparte

- **Contenido**
  - Archivos en Unix
  - **Llamadas al sistema para archivos**
  - Redirecciones y tubos (*"pipe"*)
  - Llamadas para redirecciones y tubos
  - Ejemplos en C

- **Llamadas UNIX para trabajar con archivos o dispositivos**
  - Unix tiene una única interfaz para trabajar con los dispositivos

	Descripción
`open`	Apertura/creación de archivos
`read`	Lectura en archivos
`write`	Escritura en archivos
`close`	Clausura de un archivo
`lseek`	Posiciona en un archivo puntero lectura/escritura
`stat`	Obtener información del nodo-i de un archivo

Nota: Las llamadas de lectura y escritura en archivo no hacen conversión de formato
  - Por tanto, son las funciones de entrada/salida de C (*scanf* y *printf*) las que incluyen el código de la conversión

Llamadas al sistema para archivos

open: apertura/creación archivos

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int flags)
int open(const char *path, int flags, mode_t mode)
```

Descripción

- Asocia un descriptor de archivo con archivo o dispositivo físico
  - Asigna siempre el descriptor más bajo sin ocupar en la tabla de descriptores de archivos
  - Los descriptores de archivos abiertos son atributos heredables
- `flags`

  `O_RDONLY, O_WRONLY, O_RDWR`
  `O_CREAT, O_EXCL, O_TRUNC, O_APPEND`
- `mode` Ejemplos: 0755, 04755

  `S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXU`
  `S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXG`
  `S_IROTH, S_IWOTH, S_IXOTH, S_IRWXO`
  `S_ISUID, S_ISGID`

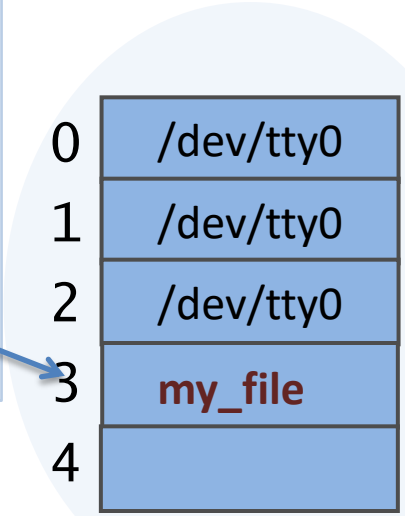
Llamadas al sistema para archivos

- Open (): apertura/creación de un archivo
  - Ejemplo de apertura de un archivo para

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

main ( int argc, char* argv[] )
{int fd;
   fd = open ("my_file", O_RDONLY);

  if (fd == -1)
   {
     perror("Error al abrir fichero:");
        exit(1);
   }
}
```



¡El archivo "my_file" ha usado la posición 3 de la tabla de descriptores → fd=3!

read/write: lectura/escritura archivos

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t nByte)
ssize_t write(int fd, const void *buf, size_t nbyte)
```

Descripción

- **read:** solicita leer nbyte Bytes del archivo con descriptor fd
  - Los bytes leídos se almacenan en buf
  - read puede leer menos bytes de los solicitados si llega a final de fichero
- **write:** solicita escribir nbyte bytes tomados del buffer buf en el archivo con descriptor fd.
- Por defecto read y write son bloqueantes y pueden ser interrumpidas por una señal
- **Valor de retorno**
  - **un entero >0**, que corresponde al número de bytes escritos/leídos
  - **-1 : error** o interrupción por señal
    - errores: fd no es un descriptor válido, `buf` no es una dirección válida, disco lleno, operación no permitida, etc.
  - **0:** intento de leer después de final de fichero

Llamadas al sistema para archivos

close: cierra un descriptor de archivo

```
#include <unistd.h>

    int  close(int fd)
```

Descripción

- Cierra el descriptor de archivo `fd`, liberando esa posición en la tabla de descriptores de archivo
- Valor de retorno
  - Nueva posición del puntero de acceso
  - -1 en caso de error (error EBADF: el descriptor fd no es válido)

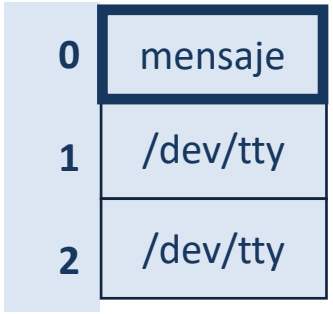
- **Contenido**
  - Archivos en Unix
  - Llamadas al sistema para archivos
  - **Redirecciones y tubos *("pipe")***
  - Llamadas para redirecciones y tubos
  - Ejemplos en C

Redirecciones y tubos

- Redirección de la entrada/salida estándar desde el intérprete de órdenes

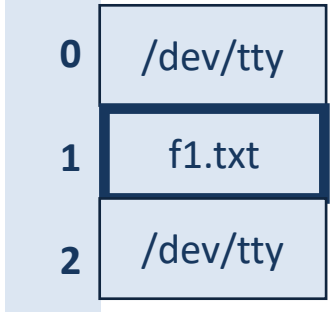
  – Redirección de entrada estándar

```
$ mail gandreu < mensaje
```



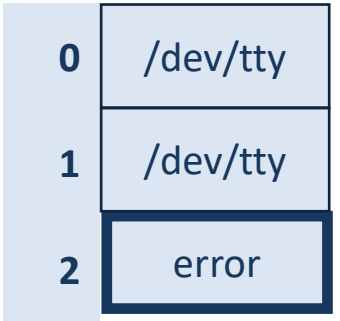
  – Redirección de la salida estándar

```
$ echo hola > f1.txt
```



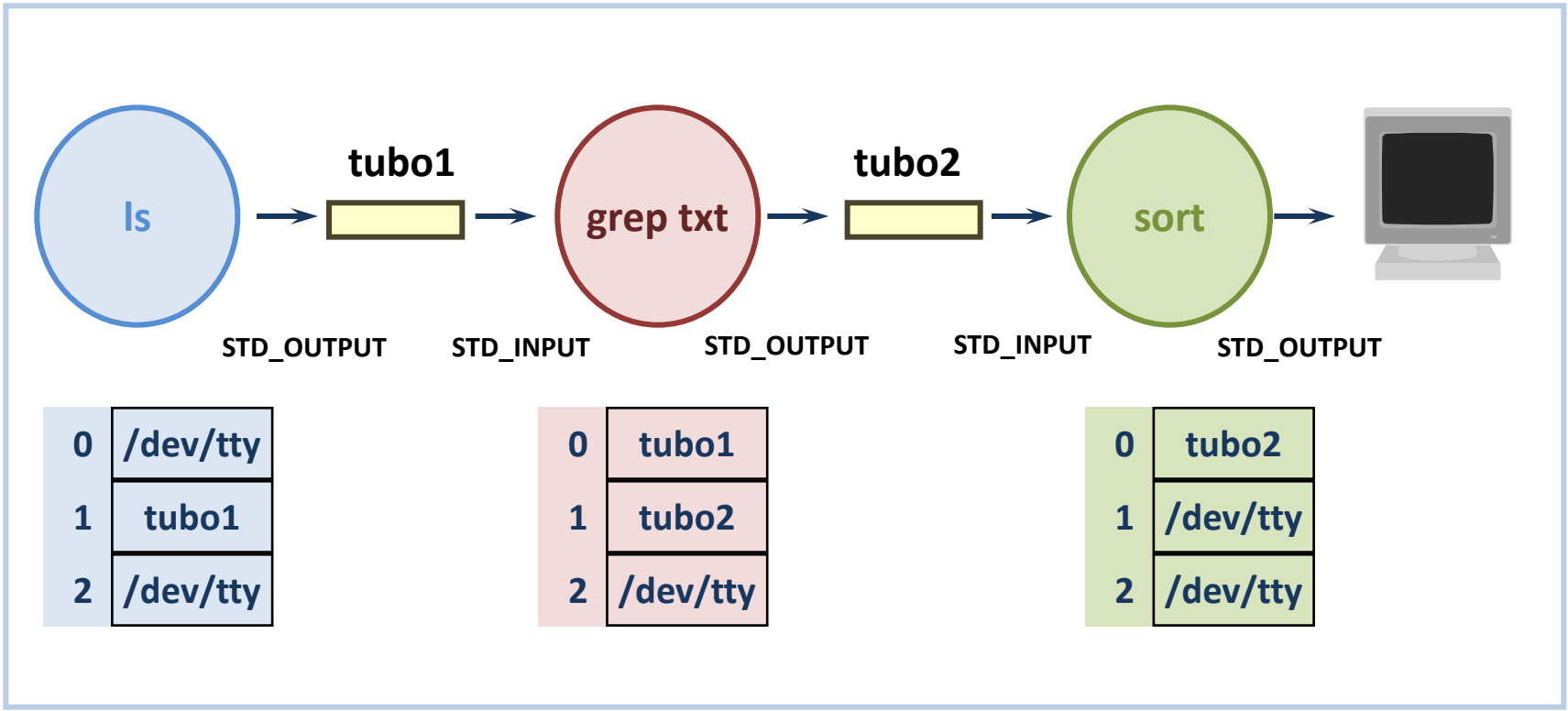
  – Redirección de la salida de error

```
$ gcc prg.c -o prg 2> error
```



- **Comunicación de procesos Unix**
  - La comunicación entre procesos en Unix se realiza mediante tubos
  - **Tubos *("PIPE")***
    - son archivos de capacidad limitada con acceso secuencial
    - pueden compartirse gracias al mecanismo de herencia

$ ls | grep txt | sort



- **Contenido**
  - Archivos en Unix
  - Llamadas al sistema para archivos
  - Redirecciones y tubos (“pipe”)
  - **Llamadas para redirecciones y tubos**
  - Ejemplos en C

- **Llamadas para redirecciones y tubos**
  - nos permitirán establecer comunicación entre procesos padre e hijo a través del mecanismo de herencia

	Descripción
dup2	Duplicar un descriptor de fichero
pipe	Creación de un tubo
mkfifo	Creación de un tubo con nombre (fifo)

dup, dup2: duplicar un descriptor de archivo

```
#include <unistd.h>
int dup(int fd)
int dup2(int oldfd, int newfd)
```

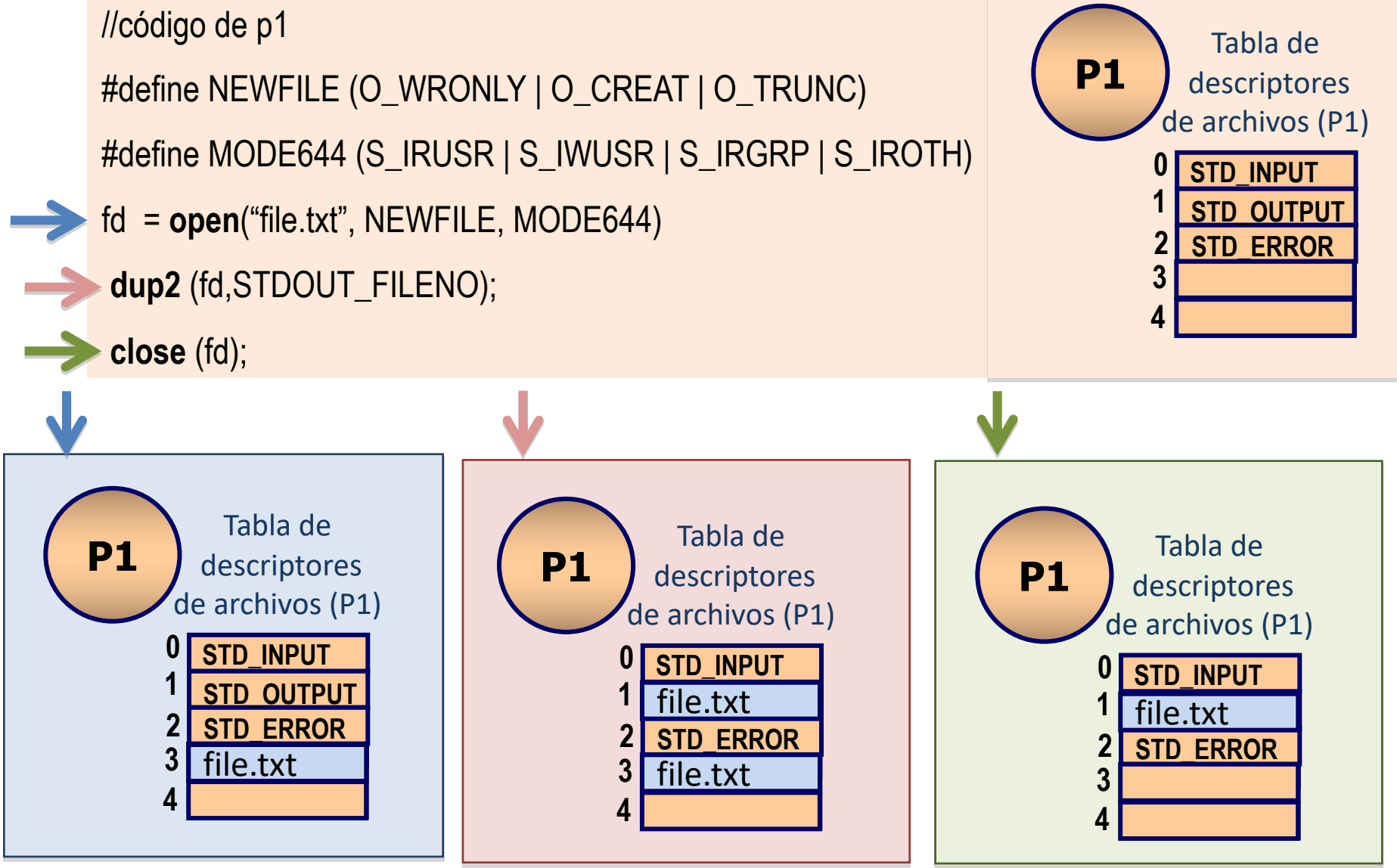
Descripción

- **dup:** retorna un descriptor de archivo (int) cuyo contenido es una copia del parámetro *fd*
  - El descriptor o valor de retorno corresponde a la posición más baja disponible en la tabla de descriptores
- **dup2:** cierra el descriptor *newfd* y luego copia *oldfd* en *newfd*

- Valor de retorno
  - Un descriptor de archivo nuevo
  - -1 en caso de **error**
    - *fd* no es descriptor válido
    - Se supera el máximo de archivos abiertos (OPEN_MAX))

Ejemplo: dup2

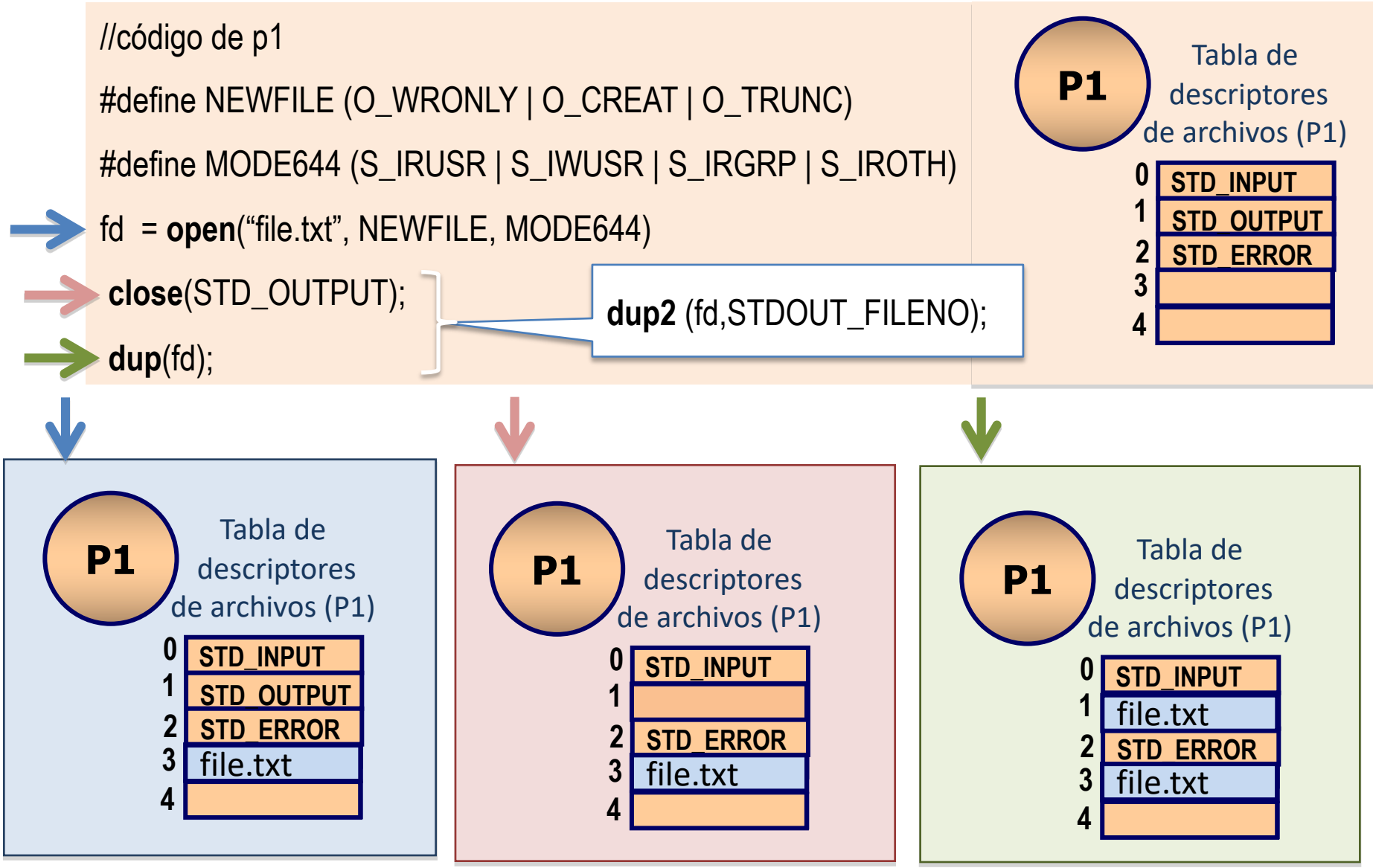
```
//código de p1
#define NEWFILE (O_WRONLY | O_CREAT | O_TRUNC)
#define MODE644 (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
fd  = open("file.txt", NEWFILE, MODE644)
dup2 (fd,STDOUT_FILENO);
close (fd);
```



Ejemplo: dup

```
//código de p1
#define NEWFILE (O_WRONLY | O_CREAT | O_TRUNC)
#define MODE644 (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
fd  = open("file.txt", NEWFILE, MODE644)
close(STD_OUTPUT);
dup(fd);
```

dup2 (fd,STDOUT_FILENO);



pipe: creación de un tubo

```
#include <unistd.h>
  int pipe(int fildes[2])
```

Descripción

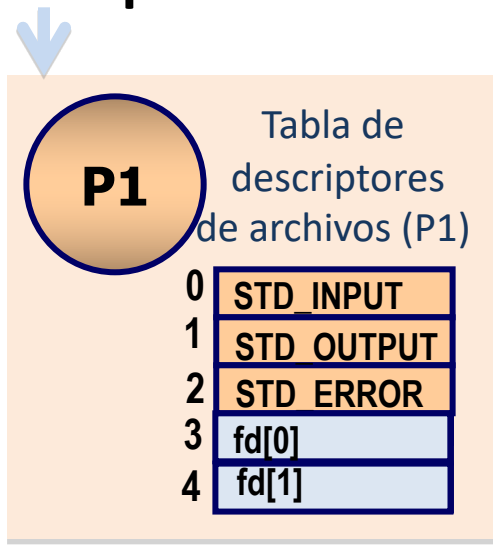
- Crea un *pseudoarchivo*, denominado **tubo**, cuya estructura es una cola FIFO de bytes inicialmente vacía
  - La capacidad máxima del tubo está limitada por la implementación
  - Tras la llamada:
    - fildes[0] es un descriptor de archivo para lectura del tubo y
    - fildes[1] es un descriptor de archivo para escritura del tubo
- Los tubos, junto con la tabla de descriptores, son heredados por los procesos hijos y preservados tras la ejecución de exec()
  - Los tubos son un mecanismo de comunicación entre procesos UNIX
- Valor de retorno
  - 0 si funciona con éxito
  - -1 en caso de error
    - Si supera el máximo de archivos abiertos (OPEN_MAX),
    - fildes no es válido

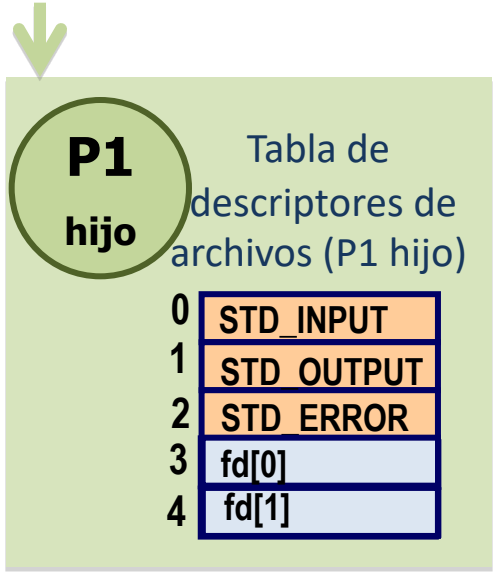
Pipe(): funcionamiento de *read()* y *write()* en un tubo

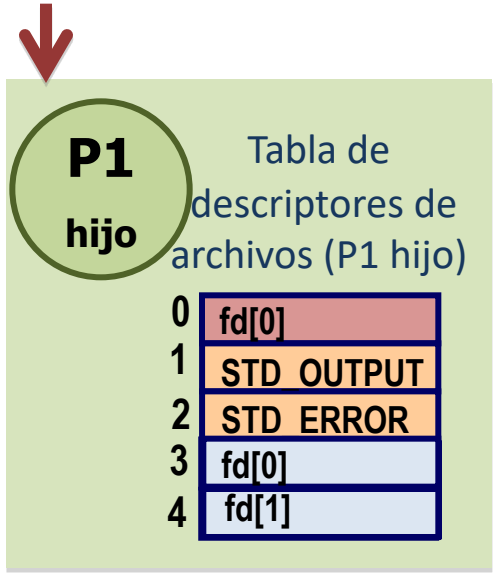
- ***read()***
  - Si existen bytes disponibles, se leen como mucho los nbytes solicitados
  - Si el tubo está vacío, el proceso que invoca *read()* se suspende hasta que exista disponibilidad de bytes en el tubo
  - El proceso que invoca *read()* no se suspende cuando no existen descriptor de escritura asociado al tubo (del proceso lector o de algún otro proceso) y la llamada *read()* retorna 0, indicando así la condición de final de datos (final de fichero)
- **write()**
  - Si existe capacidad suficiente en el tubo para albergar los nbytes solicitados, éstos son almacenados en el tubo en orden FIFO
  - Si no existe capacidad suficiente para albergar los nbytes solicitados (tubo lleno), el proceso escritor se suspende hasta que exista disponibilidad de espacio
  - Si se intenta **escribir en un tubo que no posee ningún descriptor de lectura asociado** (del proceso escritor o de algún otro proceso), **el proceso que intenta escribir recibe la señal SIGPIPE**
    - Este mecanismo facilita la eliminación automática de una cadena de procesos comunicados por tubos cuando se aborta inesperadamente un componente de ésta

Pipe() y dup(): Comunicación de procesos con tubos

```
//código proceso p1
pipe(fd);
if (fork() == 0)
{
// código del proceso hijo
    dup2 (fd[0], STDIN_FILENO);
    close (fd[0]);
    close (fd[1]);
    ...
  }else{
// código del proceso padre
  dup2 (fd[1], STDOUT_FILENO);
  close (fd[0]);
  close (fd[1]);
  ...
}
```

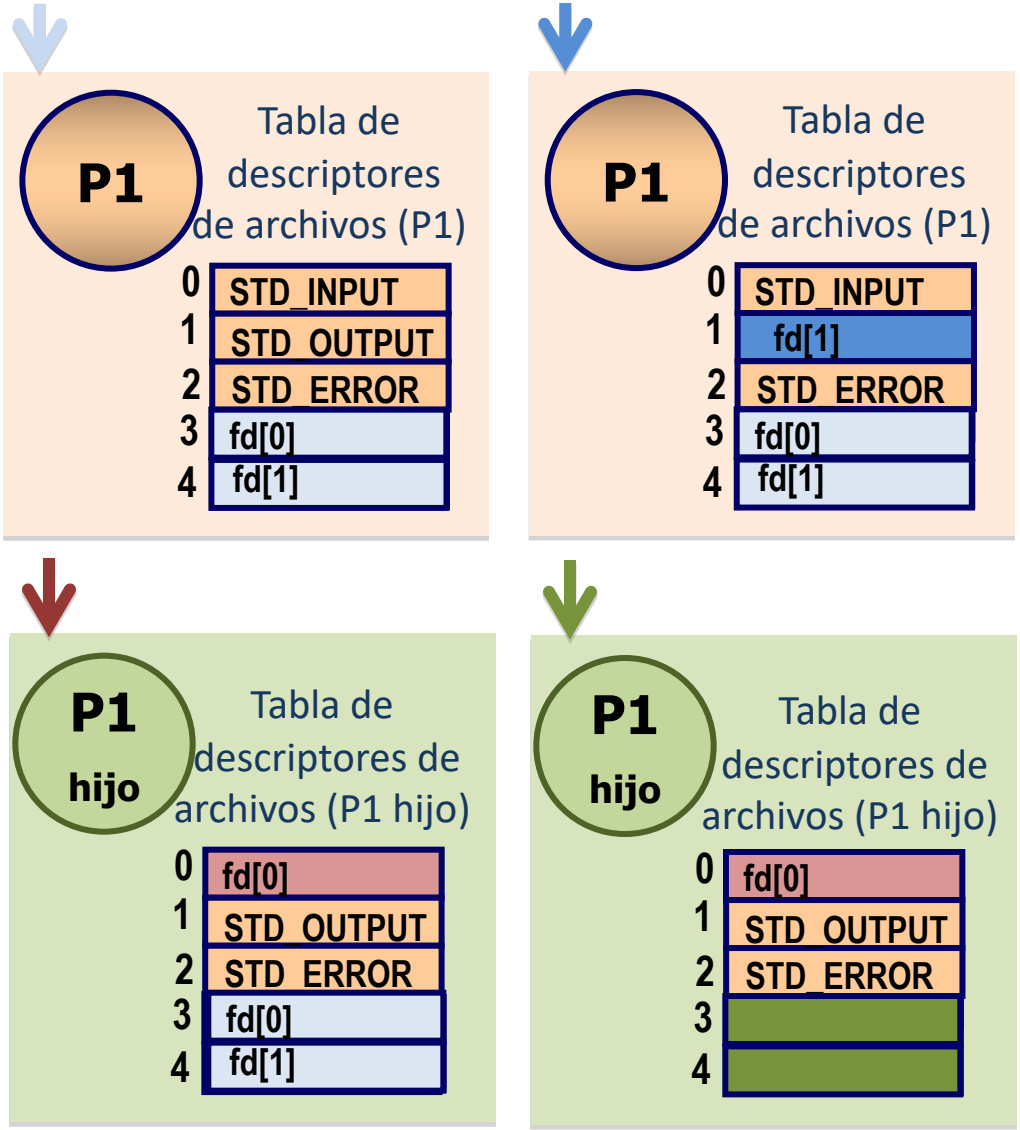






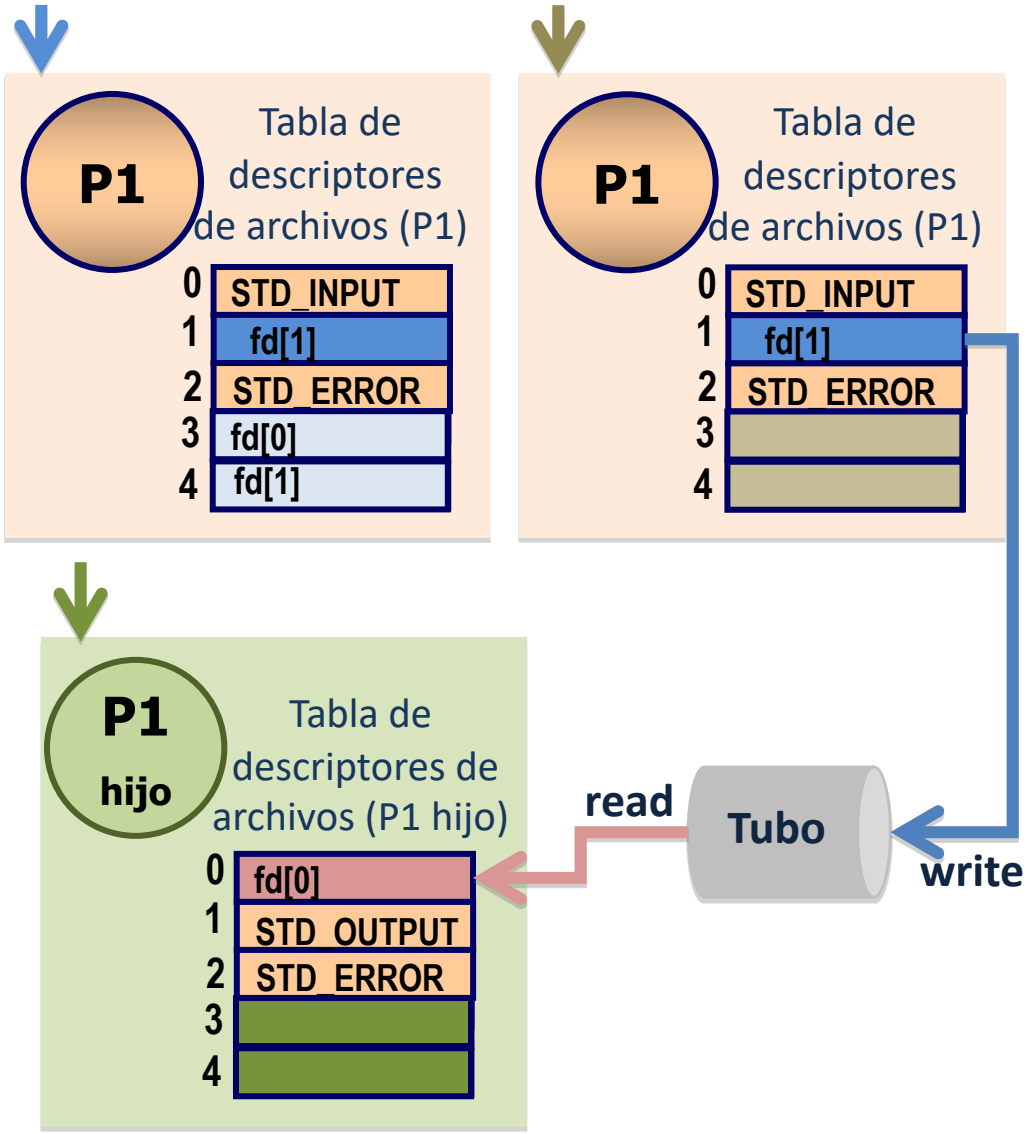
Pipe() y dup2(): Comunicación de procesos con tubos

```
//código proceso p1
pipe(fd);
if (fork() == 0)
{
// código del proceso hijo
   dup2 (fd[0], STDIN_FILENO);
   close (fd[0]);
   close (fd[1]);
   ...
  }else{
// código del proceso padre
  dup2 (fd[1], STDOUT_FILENO);
  close (fd[0]);
  close (fd[1]);
  ...
}
```



Pipe() y dup(): Comunicación de procesos con tubos

```
//código proceso p1
pipe(fd);
if (fork() == 0)
{
// código del proceso hijo
    dup2 (fd[0], STDIN_FILENO);
    close (fd[0]);
    close (fd[1]);
    ...
 }else{
// código del proceso padre
  dup2 (fd[1], STDOUT_FILENO);
   close (fd[0]);
  close (fd[1]);
   ...
}
```



- **Contenido**
  - Archivos en Unix
  - Llamadas al sistema para archivos
  - Redirecciones y tubos (“pipe)
  - Llamadas para redirecciones y tubos
  - **Ejemplos en C**

Ejemplos en C

- **Ejemplo: open, write,read**

```
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>

#define BLKSIZE 1
#define NEWFILE (O_WRONLY| O_CREAT | O_EXCL)
#define MODE600 (S_IRUSR | S_IWUSR)
```

```
int main(int argc, char *argv[]) {
  int from_fd, to_fd;
  int count;
  char buf[BLKSIZE];

  if (argc != 3) {
    fprintf(stderr, "Usage: %s from_file to_file\n", argv[0]);
    exit(1);}
  if ( (from_fd = open(argv[1], O_RDONLY)) == -1) {
    fprintf(stderr,"Could not open %s: %s\n", argv[1],strerror(errno));
    exit(1);
  }
  if ( (to_fd = open(argv[2], NEWFILE,MODE600)) == -1) {
    fprintf(stderr, "Could not create %s: %s\n", argv[2],strerror(errno));
    exit(1);
  }
  while ( (count= read(from_fd, buf, sizeof(buf))) >0 ) {
    if (write(to_fd, buf, count) != count){
      fprintf(stderr,"Could not write %s: %s\n", argv[2],strerror(errno));
      exit(1); }
    }
  if (count==-1){
    fprintf(stderr, "Could not read %s: %s\n", argv[1],strerror(errno));
    exit(1);
  }
  close(from_fd);
  close(to_fd);
  exit(0);
}
```

Para compilar y ejecutar:

```
$ gcc my_copy.c –o my_copy
$ echo 'Hola read write' > hola.txt
$ ./my_copy hola.txt hola_copia.txt
$ cat hola_copia.txt
```

- **Ejemplo: dup2**

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

#define NEWFILE (O_WRONLY | O_CREAT | O_EXCL)
#define MODE644 (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
```

```c
int redirect_output(const char *file) {
  int fd;
  if ((fd = open(file, NEWFILE, MODE644)) == -1) return -1;
  if ( dup2(fd, STDOUT_FILENO) == -1) return -1;
  close (fd);
  return 0;
}
int main(int argc, char *argv[]) {
  int from_fd, to_fd;
  if (argc < 3) {
    fprintf(stderr,"Usage: %s to_file command args\n",argv[0]);
    exit(1);
  }
  if ( redirect_output(argv[1]) == -1){
    fprintf(stderr,"Could not redirect output to: %s\n",argv[1]);
    exit(1);
  }
  if ( execvp(argv[2], &argv[2]) < 0){
    fprintf(stderr, "Could not execute: %s\n", argv[2]);
    exit(1);
  }
  return 0;
}
```

Para compilar y ejecutar:

```
$ gcc dup2.c -o dup2
$ ./dup2 prueba ls
$ cat prueba
```

- **Ejemplo comunicación de procesos con tubos**

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

```
int main(int argc, char *argv[]) {
  int i, fd[2];
  if (argc < 2) {
    fprintf(stderr, "Usage: %s filter\n", argv[0]);
    exit(1);
  }
  pipe(fd);
  for(i=0; i<2; i++) {
    if (fork() == 0) { // children
      dup2 (fd[1], STDOUT_FILENO);
      close (fd[0]);
      close (fd[1]);
      execlp("/bin/ls", "ls", NULL);
      perror("The exec of ls failed");
    }
  }
  // parent
  dup2 (fd[0], STDIN_FILENO);
  close (fd[0]);
  close (fd[1]);
  execvp(argv[1],&argv[1]);
  fprintf(stderr,"The exec of %s failed", argv[1]);
  exit(1);
}
```

Para compilar y ejecutar:

```
$ gcc pipe.c –o pipe
$ ./pipe wc
```