Prácticas de Sistemas Operativos

Módulo II. Uso de los Servicios del SO Linux mediante la API

Sesión 4. Comunicación entre procesos utilizando cauces

1. Objetivos principales

En esta sesión estudiaremos y practicaremos con los mecanismos de comunicación de información entre procesos de más alto nivel presentes en los sistemas operativos UNIX, y que por este motivo son los más ampliamente utilizados. Estos mecanismos para la comunicación entre procesos (*Inter-Process Communication* o también IPC) son los cauces (*pipes)*, que también se traducen por tuberías.

En particular como objetivos más específicos abordaremos:

- Comprender el concepto de cauce, y sus distintos tipos, como mecanismo de alto de nivel y soportado por los sistemas operativos para comunicar y sincronizar procesos.
- Conocer la sintaxis y semántica de las principales llamadas al sistema, y estructuras de datos asociadas, para manejar cauces: creación, apertura y utilización mediante el redireccionamiento de las entradas/salidas estándar de los procesos, envío y recepción de información, y eliminación.
- Ser capaz de construir programas que se comunicarán y sincronizarán con otros mediante cauces, evitando los problemas de interbloqueos que pueden surgir debido a las semántica de bloqueo de algunas de las llamadas al sistema que operan sobre cauces.

2. Concepto y tipos de cauce

Un cauce es un mecanismo para la comunicación de información y sincronización entre procesos. Los datos pueden ser enviados (escritos) por varios procesos al cauce, y a su vez, recibidos (leídos) por otros procesos desde dicho cauce.

La comunicación a través de un cauce sigue el paradigma de interacción productor/consumidor, donde típicamente existen dos tipos de procesos que se comunican mediante un búfer: aquellos que generan datos (productores) y otros que los toman (consumidores). Estos datos se tratan en orden FIFO (*First In First Out*). La lectura de los datos por parte de un proceso produce su eliminación del cauce, por tanto esos datos son consumidos únicamente por el primer proceso que haga una operación de lectura.

La sincronización básica que ocurre se debe a que los datos no pueden ser consumidos mientras no sean enviados al cauce. Así pues, un proceso que intenta leer datos de un cauce

se bloquea si actualmente no existen dichos datos, es decir, no se han escrito aún en el cauce por parte de alguno de los procesos productores, o si previamente los ha tomado ya alguno de los otros procesos consumidores. Los cauces proporcionan un método de comunicación entre procesos en un sólo sentido (unidireccional, semi-dúplex), es decir, si deseamos comunicar en el otro sentido es necesario utilizar otro cauce diferente.

Hay dos tipos de cauces en los sistemas operativos UNIX, a saber, cauces sin y con nombre. Comúnmente usamos un cauce como un método de conexión que une la salida estándar de un proceso a la entrada estándar de otro. Este método se usa bastante en la línea de órdenes de los shell de UNIX, por ejemplo:

```
$> ls | sort | lp
```

El anterior es un ejemplo claro de *pipeline* (tubería formada por dos cauces sin nombre y tres procesos), donde se toma la salida de la orden **`ls`** como entrada de la orden **`sort`**, la cual a su vez entrega su salida a la orden **`lp`**. Los datos fluyen por dos cauces sin nombre (semi-dúplex) viajando de izquierda a derecha. Al igual que los tres procesos implicados (procesos hijos del shell), ambos cauces sin nombre, que permiten comunicar procesos gracias a la jerarquía padre-hijo que mantiene Linux, los crea dinámicamente el propio shell (con la llamada al sistema **`pipe`***)*. Los cauces sin nombre tienen las siguientes características:

- No tienen un archivo asociado en el sistema de archivos en disco, sólo existe el archivo temporalmente y en memoria principal.
- Al crear un cauce sin nombre utilizando la llamada al sistema **`pipe`**, automáticamente se devuelven dos descriptores, uno de lectura y otro de escritura, para trabajar con el cauce. Por consiguiente no es necesario realizar una llamada **`open`**.
- Los cauces sin nombre sólo pueden ser utilizados como mecanismo de comunicación entre el proceso que crea el cauce sin nombre y los procesos descendientes creados a partir de la creación del cauce.
- El cauce sin nombre se cierra y elimina automáticamente por el núcleo cuando los contadores asociados de números de productores y consumidores que lo tienen en uso valen simultáneamente `0`.

Un cauce con nombre (o archivo FIFO) funciona de forma parecida a un cauce sin nombre aunque presenta las siguientes diferencias:

- Los cauces con nombre se crean (llamadas al sistema **`mknod`** y **`mkfifo`**) en el sistema de archivos en disco como un archivo especial, es decir, consta de un nombre que ayuda a denominarlo exactamente igual que a cualquier otro archivo en el sistema de archivos, y por tanto aparecen contenidos/asociados de forma permanente a los directorios donde se crearon.
- Los procesos abren y cierran un archivo FIFO usando su nombre mediante las ya conocidas llamadas al sistema **`open`** y **`close`**, con el fin de hacer uso de él.
- Cualesquiera procesos pueden compartir datos utilizando las ya conocidas llamadas al sistema **`read`** y **`write`** sobre el cauce con nombre previamente abierto. Es decir, los cauces con nombre permiten comunicar a procesos que no tienen un antecesor común en la jerarquía de procesos de UNIX.

- El archivo FIFO permanece en el sistema de archivos una vez realizadas todas las E/S de los procesos que lo han utilizado como mecanismo de comunicación, hasta que se borre explícitamente (llamada al sistema **`unlink`**) como cualquier archivo.

A continuación se describe en detalle el funcionamiento de los dos tipos de cauces, así como ejemplos y actividades para ayudar a su mejor comprensión y utilización como mecanismo simple y efectivo para comunicar procesos.

3. Cauces con nombre

3.1 Creación de archivos FIFO

Una vez creado el cauce con nombre cualquier proceso puede abrirlo para lectura y/o escritura, de la misma forma que un archivo regular. Sin embargo, el cauce debe estar abierto en ambos extremos simultáneamente antes de que podamos realizar operaciones de lectura o escritura sobre él. Abrir un archivo FIFO para sólo lectura produce un bloqueo hasta que algún otro proceso abra el mismo cauce para escritura.

Para crear un archivo FIFO en el lenguaje C podemos hacer uso de la llamada al sistema `mknod`, que permite crear archivos especiales, tales como los archivos FIFO o los archivos de dispositivo. La biblioteca de GNU incluye esta llamada por compatibilidad con Unix BSD.

```
int mknod (const char *FILENAME, mode_t MODE, dev_t DEV)
```

La llamada al sistema `mknod` crea un archivo especial de nombre `FILENAME`. El parámetro `MODE` especifica los valores que serán almacenados en el campo `st_mode` del i-nodo correspondiente al archivo especial:

- `S_IFCHR`: representa el valor del código de tipo de archivo para un archivo de dispositivo orientado a caracteres.
- `S_IFBLK`: representa el valor del código de tipo de archivo para un archivo de dispositivo orientado a bloques.
- `S_IFSOCK`: representa el valor del código de tipo de archivo para un socket.
- `S_IFIFO`: representa el valor del código de tipo de archivo para un FIFO .

El argumento `DEV` especifica a qué dispositivo se refiere el archivo especial. Su interpretación depende de la clase de archivo especial que se vaya a crear. Para crear un cauce FIFO el valor de este argumento será `0`. Un ejemplo de creación de un cauce FIFO sería el siguiente:

```
mknod("/tmp/FIFO", S_IFIFO|0666,0);
```

En este caso el archivo **`/tmp/FIFO`** se crea como archivo FIFO y los permisos solicitados son **`0666`**. Los permisos que el sistema finalmente asigna al archivo son el resultado de la siguiente expresión, como ya vimos en una sesión anterior:

```
umaskFinal = MODE & ~umaskInicial
```

donde **umaskInicial** es la máscara de permisos que almacena el sistema para el proceso, con el objetivo de asignar permisos a los archivos de nueva creación.

La llamada al sistema **mknod()** permite crear cualquier tipo de archivo especial. Sin embargo, para el caso particular de los archivos FIFO existe una llamada al sistema específica:

```
int mkfifo (const char *FILENAME, mode_t MODE)
```

Esta llamada crea un archivo FIFO cuyo nombre es **FILEMAME**. El argumento **MODE** se usa para establecer los permisos del archivo.

Los archivos FIFO se eliminan con la llamada al sistema **unlink**, consulte el manual en línea para obtener más información al respecto.

3.2 Utilización de un cauce FIFO

Las operaciones de E/S sobre un archivo FIFO son esencialmente las mismas que las utilizadas con los archivos regulares salvo una diferencia: en el archivo FIFO no podemos hacer uso de **lseek** debido a la filosofía de trabajo FIFO basada en que el primer dato en entrar será el primero en salir. Por tanto no tiene sentido mover el *offset* (o posición actual de lectura/escritura) a una posición dentro del flujo de datos, el núcleo lo hará automáticamente siguiendo la política FIFO. El resto de llamadas (**open**, **close**, **read** y **write**) se pueden utilizar de la misma manera que hemos visto hasta ahora para los archivos regulares, excepto que tal como se ha comentado antes, de aplicación a los cauces de cualquier tipo:

1. La llamada **read** es bloqueante para los procesos consumidores cuando no hay datos que leer en el cauce, y
2. **read** desbloquea devolviendo `0` (ningún *byte* leído) cuando todos los procesos que tenían abierto el cauce en modo escritura, esto es, los procesos que actuaban como productores, lo han cerrado o han terminado.

Actividad 4.1 Trabajo con cauces con nombre

Ejercicio 1. Consulte en el manual las llamadas al sistema para la creación de archivos especiales en general (**mknod**) y la específica para archivos FIFO (**mkfifo**). Pruebe a ejecutar el siguiente código correspondiente a dos programas que modelan el problema del productor/consumidor, los cuales utilizan como mecanismo de comunicación un cauce FIFO. Determine en qué orden y manera se han de ejecutar los dos programas para su correcto funcionamiento y cómo queda reflejado en el sistema que estamos utilizando un cauce FIFO. Justifique la respuesta.

```
//consumidorFIFO.c
//Consumidor que usa mecanismo de comunicacion FIFO

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#define ARCHIVO_FIFO "ComunicacionFIFO"

int main(void)
{
int fd;
char buffer[80];// Almacenamiento del mensaje del cliente
int leidos;

//Crear el cauce con nombre (FIFO) si no existe
umask(0);
mknod(ARCHIVO_FIFO,S_IFIFO|0666,0);
//también vale: mkfifo(ARCHIVO_FIFO,0666);

//Abrir el cauce para lectura-escritura
if ( (fd=open(ARCHIVO_FIFO,O_RDWR)) <0) {
perror("open");
exit(-1);
}

//Aceptar datos a consumir hasta que se envíe la cadena fin
while(1) {
leidos=read(fd,buffer,80);
if(strcmp(buffer,"fin")==0) {
close(fd);
return 0;
}
printf("\nMensaje recibido: %s\n", buffer);
}

return 0;
}

/* ======================== * ======================== */
Y el código de cualquier proceso productor quedaría de la siguiente forma:
/* ======================== * ======================== */
//productorFIFO.c
//Productor que usa mecanismo de comunicacion FIFO

#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#define ARCHIVO_FIFO "ComunicacionFIFO"

int main(int argc, char *argv[])
{
```

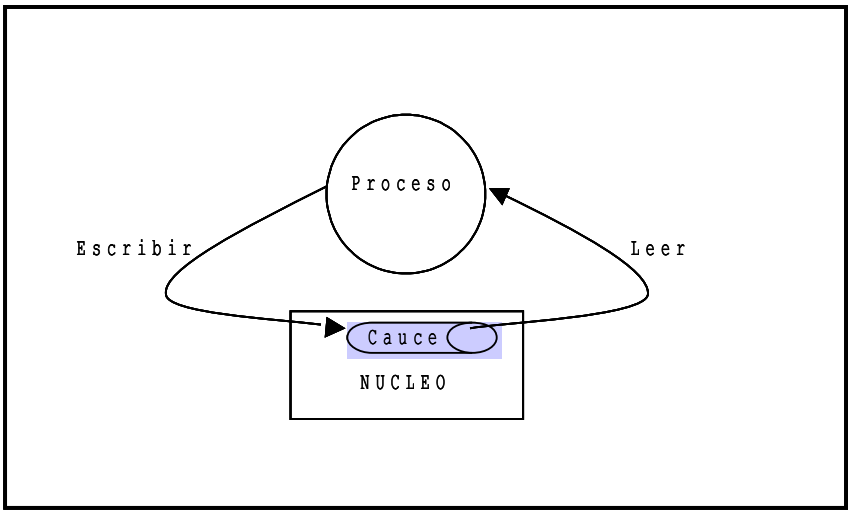
```
int fd;

//Comprobar el uso correcto del programa
if(argc != 2) {
printf("\nproductorFIFO: faltan argumentos (mensaje)");
printf("\nPruebe: productorFIFO <mensaje>, donde <mensaje> es una
cadena de caracteres.\n");
exit(-1);
}

//Intentar abrir para escritura el cauce FIFO
if( (fd=open(ARCHIVO_FIFO,O_WRONLY)) <0) {
perror("\nError en open");
exit(-1);
}

//Escribir en el cauce FIFO el mensaje introducido como argumento
if( (write(fd,argv[1],strlen(argv[1])+1)) != strlen(argv[1])+1) {
perror("\nError al escribir en el FIFO");
exit(-1);
}

close(fd);
return 0;
}
```

4. Cauces sin nombre

4.1 Esquema de funcionamiento

¿Qué ocurre realmente a nivel de núcleo cuando un proceso crea un cauce sin nombre?

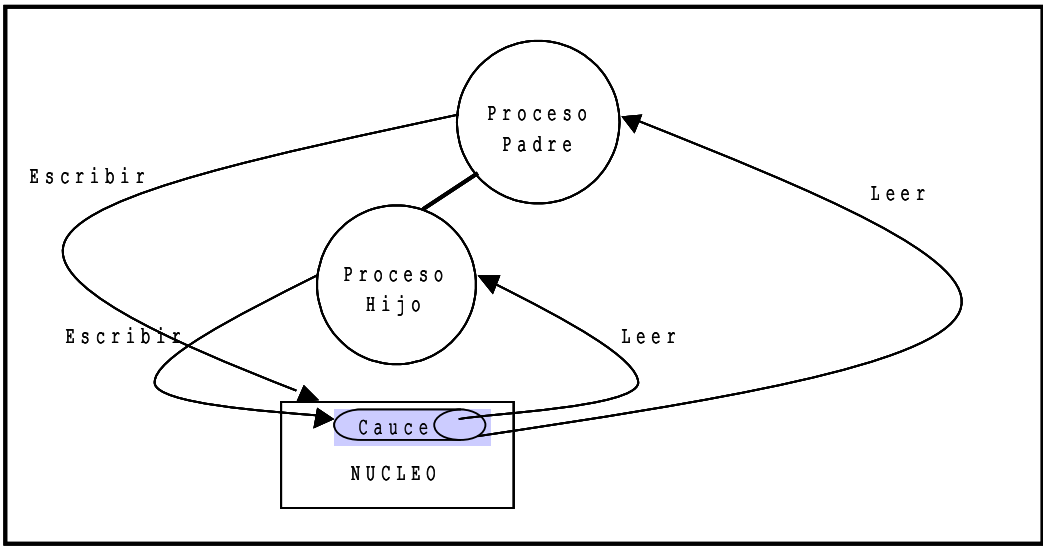
Al ejecutar la llamada al sistema **`pipe`** para crear un cauce sin nombre, el núcleo automáticamente instala dos descriptores de archivo para que los use dicho cauce. Un descriptor se usa para permitir un camino de envío de datos (**`write`**) al cauce, mientras que el otro descriptor se usa para obtener los datos (**`read`**) de éste.



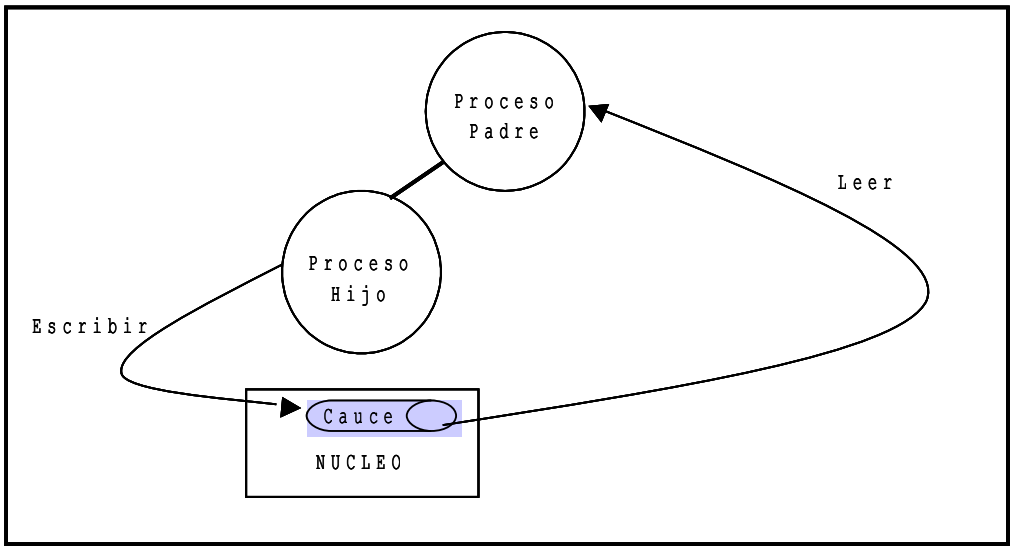
En el esquema anterior se puede observar cómo el proceso puede usar los descriptores de archivo para enviar datos (llamada al sistema **`write`**) al cauce, y leerlos desde éste con la llamada al sistema **`read`**. Sin embargo, este esquema se puede ampliar ya que carece de utilidad práctica; el mecanismo de cauces tiene sentido para comunicar información entre dos o más procesos puesto que no comparten memoria y por tanto no tiene sentido que un mismo proceso actúe como productor y consumidor al mismo tiempo utilizando para ello un cauce.

Mientras un cauce conecta inicialmente un proceso a sí mismo, los datos que viajan por él se mueven a nivel de núcleo. Bajo UNIX en particular, los cauces se representan internamente por medio de un i-nodo válido (entrada en la tabla de i-nodos en memoria principal).

Partiendo de la situación anterior, el proceso que creó el cauce crea un proceso hijo. Como un proceso hijo hereda cualquier descriptor de archivo abierto por el padre, ahora disponemos de una forma de comunicación entre los procesos padre e hijo.



En este momento, se debe tomar una decisión crítica: ¿en qué dirección queremos que viajen los datos? ¿el proceso hijo envía información al padre (o viceversa)? Los dos procesos deben adecuarse a la decisión y cerrar los correspondientes extremos no necesarios (uno en cada proceso). Pongamos como ejemplo que el hijo realiza algún tipo de procesamiento y devuelve información al padre usando para ello el cauce. El esquema quedaría finalmente como se muestra en la siguiente figura.



4.2 Creación de cauces

Para crear un cauce sin nombre en el lenguaje C utilizaremos la llamada al sistema **pipe**, la cuál toma como argumento un vector de dos enteros **int fd[2]**. Si la llamada tiene éxito, el vector contendrá dos nuevos descriptores de archivo que permitirán usar el nuevo cauce. Por defecto, se suele tomar el primer elemento del vector (**fd[0]**) como un descriptor de archivo para sólo lectura, mientras que el segundo elemento (**fd[1]**) se toma para escritura.

Una vez creado el cauce, creamos un proceso hijo (que heredará los descriptores de archivos del padre) y establecemos el sentido del flujo de datos (hijo->padre o padre->hijo). Como los descriptores son compartidos por el proceso padre y el hijo, debemos estar seguros siempre de cerrar con la llamada al sistema **close** el extremo del cauce que no nos interese en cada uno de los procesos, para evitar confusiones que podrían derivar en errores al usar el mecanismo. Si el padre quiere recibir datos del hijo, debe cerrar el descriptor usado para escritura (**fd[1]**) y el hijo debe cerrar el descriptor usado para lectura (**fd[0]**). Si por el contrario el padre quiere enviarle datos al hijo, debe cerrar el descriptor usado para lectura (**fd[0]**) y el hijo debe cerrar el descriptor usado para escritura (**fd[1]**).

Si deseamos conseguir redireccionar la entrada o salida estándar al descriptor de lectura o escritura del cauce podemos hacer uso de las llamadas al sistema **close**, **dup** y **dup2**.

La llamada al sistema `dup` que se encarga de duplicar el descriptor indicado como parámetro de entrada en la primera entrada libre de la tabla de descriptores de archivo usada por el proceso. Puede interesar ejecutar **close** justo con anterioridad a **dup** con el objetivo de dejar la entrada deseada libre. Recuerde que el descriptor de archivo 0 (**STDIN_FILENO**) de cualquier proceso UNIX direcciona la entrada estándar (**stdin**) que se asigna por defecto al teclado, y el descriptor de archivo 1 (**STDOUT_FILENO**) direcciona la salida estándar (**stdout**) asignada por defecto a la consola activa.

La llamada al sistema **dup2** permite una atomicidad (evita posibles condiciones de carrera) en las operaciones sobre duplicación de descriptores de archivos que no proporciona **dup**. Con ésta, disponemos en una sola llamada al sistema de las operaciones relativas a cerrar descriptor antiguo y duplicar descriptor. Se garantiza que la llamada es atómica, por lo que si por ejemplo, si llega una señal al proceso, toda la operación transcurrirá antes de devolverle el control al núcleo para gestionar la señal.

4.3 Notas finales sobre cauces con y sin nombre

A continuación se describen brevemente algunos aspectos adicionales que suelen ser necesarios tener en cuenta con carácter general cuando se utilizan cauces:

- Se puede crear un método de comunicación dúplex entre dos procesos abriendo dos cauces.
- La llamada al sistema **`pipe`** debe realizarse siempre antes que la llamada **`fork`**. Si no se sigue esta norma, el proceso hijo no heredará los descriptores del cauce.
- Un cauce sin nombre o un archivo FIFO tienen que estar abiertos simultáneamente por ambos extremos para permitir la lectura/escritura. Se pueden producir las siguientes situaciones a la hora de utilizar un cauce:
  1. El primer proceso que abre el cauce (en modo sólo lectura) es el proceso lector. Entonces, la llamada **`open`** bloquea a dicho proceso hasta que algún proceso abra dicho cauce para escribir.
  2. El primer proceso que abre el cauce (en modo sólo escritura) es el proceso escritor. En este caso, la llamada al sistema `open` no bloquea al proceso, pero cada vez que se realiza una operación de escritura sin que existan procesos lectores, el sistema envía al proceso escritor una señal **`SIGPIPE`**. El proceso escritor debe manejar la señal si no quiere finalizar (acción por defecto de la señal **`SIGPIPE`**). Se practicará con las señales en la siguiente sesión.
  3. ¿Qué pasaría si abre el cauce para lectura y escritura tanto en el proceso lector como en el escritor?
- La sincronización entre procesos productores y consumidores es atómica.

Actividad 4.2. Trabajo con cauces sin nombre

Ejercicio 2. Consulte en el manual en línea la llamada al sistema **`pipe`** para la creación de cauces sin nombre. Pruebe a ejecutar el siguiente programa que utiliza un cauce sin nombre y describa la función que realiza. Justifique la respuesta.

```
/*
tarea6.c
Trabajo con llamadas al sistema del Subsistema de Procesos y Cauces conforme a
POSIX 2.10
*/

#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>

int main(int argc, char *argv[])
{
int fd[2], numBytes;
pid_t PID;
char mensaje[]= "\nEl primer mensaje transmitido por un cauce!!\n";
char buffer[80];

pipe(fd); // Llamada al sistema para crear un cauce sin nombre

if ( (PID= fork())<0) {
  perror("fork");
  exit(-1);
}
if (PID == 0) {
//Cierre del descriptor de lectura en el proceso hijo
close(fd[0]);
// Enviar el mensaje a través del cauce usando el descriptor de escritura
write(fd[1],mensaje,strlen(mensaje)+1);
exit(0);
}
else { // Estoy en el proceso padre porque PID != 0
//Cerrar el descriptor de escritura en el proceso padre
close(fd[1]);
//Leer datos desde el cauce.
numBytes= read(fd[0],buffer,sizeof(buffer));
printf("\nEl número de bytes recibidos es: %d",numBytes);
printf("\nLa cadena enviada a través del cauce es: %s", buffer);
}

return(0);
}
```

Ejercicio 3. Redirigiendo las entradas y salidas estándares de los procesos a los cauces podemos escribir un programa en lenguaje C que permita comunicar órdenes existentes sin necesidad de reprogramarlas, tal como hace el shell (por ejemplo **`ls | sort`**). En particular, ejecute el siguiente programa que ilustra la comunicación entre proceso padre e hijo a través de un cauce sin nombre redirigiendo la entrada estándar y la salida estándar del padre y el hijo respectivamente.

```
/*
tarea7.c
Programa ilustrativo del uso de pipes y la redirección de entrada y
salida estándar: "ls | sort"
*/

#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>

int main(int argc, char *argv[])
{
int fd[2];
pid_t PID;

pipe(fd); // Llamada al sistema para crear un pipe

if ( (PID= fork())<0) {
perror("fork");
exit(-1);
}
if(PID == 0) { // ls
//Establecer la dirección del flujo de datos en el cauce cerrando
// el descriptor de lectura de cauce en el proceso hijo
close(fd[0]);

//Redirigir la salida estándar para enviar datos al cauce
//-------------------------------------------------------
//Cerrar la salida estándar del proceso hijo
close(STDOUT_FILENO);

//Duplicar el descriptor de escritura en cauce en el descriptor
//correspondiente a la salida estándar (stdout)
dup(fd[1]);
execlp("ls","ls",NULL);
}
else { // sort. Estoy en el proceso padre porque PID != 0

//Establecer la dirección del flujo de datos en el cauce cerrando
// el descriptor de escritura en el cauce del proceso padre.
close(fd[1]);

//Redirigir la entrada estándar para tomar los datos del cauce.
//Cerrar la entrada estándar del proceso padre
close(STDIN_FILENO);

//Duplicar el descriptor de lectura de cauce en el descriptor
//correspondiente a la entrada estándar (stdin)
dup(fd[0]);
execlp("sort","sort",NULL);
}

return(0);
}
```

Ejercicio 4. Compare el siguiente programa con el anterior y ejecútelo. Describa la principal diferencia, si existe, tanto en su código como en el resultado de la ejecución.

```
/*
tarea8.c
Programa ilustrativo del uso de pipes y la redirección de entrada y
salida estándar: "ls | sort", utilizando la llamada dup2.
*/

#include<sys/types.h>
#include<fcntl.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>

int main(int argc, char *argv[])
{
int fd[2];
pid_t PID;

pipe(fd); // Llamada al sistema para crear un pipe

if ( (PID= fork())<0) {
perror("\Error en fork");
exit(-1);
}
if (PID == 0) { // ls
//Cerrar el descriptor de lectura de cauce en el proceso hijo
close(fd[0]);

//Duplicar el descriptor de escritura en cauce en el descriptor
//correspondiente a la salida estda r (stdout), cerrado previamente en
//la misma operación
dup2(fd[1],STDOUT_FILENO);
execlp("ls","ls",NULL);
}
else { // sort. Proceso padre porque PID != 0.
//Cerrar el descriptor de escritura en cauce situado en el proceso padre
close(fd[1]);

//Duplicar el descriptor de lectura de cauce en el descriptor
//correspondiente a la entrada estándar (stdin), cerrado previamente en
//la misma operación
dup2(fd[0],STDIN_FILENO);
execlp("sort","sort",NULL);
}

return(0);
}
```

Ejercicio 5.

Este ejercicio se basa en la idea de utilizar varios procesos para realizar partes de una computación en paralelo. Para ello, deberá construir un programa que siga el esquema de computación maestro-esclavo, en el cual existen varios procesos trabajadores (esclavos) idénticos y un único proceso que reparte trabajo y reúne resultados (maestro). Cada esclavo

es capaz de realizar una computación que le asigne el maestro y enviar a este último los resultados para que sean mostrados en pantalla por el maestro.

El ejercicio concreto a programar consistirá en el cálculo de los números primos que hay en un intervalo. Será necesario construir dos programas, ***maestro*** y ***esclavo***. Ten en cuenta la siguiente especificación:

1. El intervalo de números naturales donde calcular los número primos se pasará como argumento al programa ***maestro***. El maestro creará dos procesos esclavos y dividirá el intervalo en dos subintervalos de igual tamaño pasando cada subintervalo como argumento a cada programa ***esclavo***. Por ejemplo, si al maestro le proporcionamos el intervalo entre 1000 y 2000, entonces un esclavo debe calcular y devolver los números primos comprendidos en el subintervalo entre 1000 y 1500, y el otro esclavo entre 1501 y 2000. El maestro creará dos cauces sin nombre y se encargará de su redirección para comunicarse con los procesos esclavos. El maestro irá recibiendo y mostrando en pantalla (también uno a uno) los números primos calculados por los esclavos en orden creciente.

2. El programa ***esclavo*** tiene como argumentos el extremo inferior y superior del intervalo sobre el que buscará números primos. Para identificar un número primo utiliza el siguiente método concreto: un número n es primo si no es divisible por ningún k tal que `2 < k <= sqrt(n)`, donde `sqrt` corresponde a la función de cálculo de la raíz cuadrada (consulte dicha función en el manual). El esclavo envía al maestro cada primo encontrado como un dato entero (4 bytes) que escribe en la salida estándar, la cuál se tiene que encontrar redireccionada a un cauce sin nombre. Los dos cauces sin nombre necesarios, cada uno para comunicar cada esclavo con el maestro, los creará el maestro inicialmente. Una vez que un esclavo haya calculado y enviado (uno a uno) al maestro todos los primos en su correspondiente intervalo terminará.