postdata9
www.wuolah.com/student/postdata9

sesion4.pdf

Módulo II

2° Sistemas Operativos

Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

KEEP CALM AND ESTUDIA UN POQUITO

WUOLAH + #QuédateEnCasa

Sesión 4:

Comunicación entre procesos utilizando cauces

Índice:

1. Concepto y tipos de cauce
   - 1.1 Cauces sin nombre
   - 1.2 Cauces con nombre

2. Cauces con nombre
   - 2.1 mknod
   - 2.2 mkfifo
   - 2.3 unlink
   - 2.4 Utilización de un cauce FIFO
   - 2.5 Ejercicio 1

3. Cauces sin nombre
   - 3.1 pipe
   - 3.2 dup y dup2
   - 3.3 Esquema de funcionamiento
   - 3.4 Creación de cauces
   - 3.5 Notas finales sobre cauces con y sin nombre
   - 3.6 Ejercicio 2
   - 3.7 Ejercicio 3
   - 3.8 Ejercicio 4
   - 3.9 Ejercicio 5

4. Preguntas de repaso

1. Concepto y tipos de cauce

Un **cauce** es un mecanismo para la comunicación de información y sincronización entre procesos.

- Los datos pueden ser **escritos** por **varios procesos** y **leídos** por **otros procesos** desde un mismo cauce.
- Estos datos se tratan en **orden FIFO**, es decir, el primero en ser escrito es el primero en ser leído.
- La **lectura** de un dato en el cauce produce su **eliminación** del mismo, por lo que un dato sólo puede ser leído una vez.
- Un proceso que intenta leer datos de un cauce **se bloquea** si actualmente no existen dichos datos (si ya se han leído o no se han escrito todavía en el cauce.
- Los cauces proporcionan un método de comunicación entre procesos en **un sólo sentido** (unidireccional, semi-dúplex); si deseamos comunicar en el otro sentido es necesario utilizar otro cauce diferente.
- Un cauce **une** la **salida estándar** de un proceso a la **entrada estándar** de otro.

Hay dos tipos de cauces en los sistemas operativos UNIX: **sin nombre** y **con nombre**.

1.1 Cauces sin nombre

- 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 con la llamada al sistema **pipe**, automáticamente se devuelven **dos descriptores**, uno de lectura y otro de escritura, para trabajar con el cauce. Por tanto, 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 ejemplo de cauce sin nombre es: **$ ls | sort | lp;**

Tenemos dos cauces (|) y tres procesos implicados (ls, sort y lp). La salida de ls se manda como entrada a sort, y la salida de sort es la entrada de lp.

1.2 Cauces con nombre

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

- se crean (con mknod y mkfifo) en el sistema de archivos en disco como un **archivo especial;**
- dicho archivo consta de un **nombre**, y al igual que a cualquier otro archivo en el SA, aparecen contenidos asociados de forma permanente a los directorios donde se crearon;
- los procesos abren y cierran un archivo FIFO usando su nombre con **open** y **close;**
- **cualquier proceso** puede leer/escribir en el cauce, previamente abierto, con **read**/**write;**
- el **archivo FIFO permanece en el SA** una vez realizadas todas las E/S de los procesos que lo han utilizado como mecanismo de comunicación, **hasta que se borre explícitamente** (con **unlink**) como cualquier archivo.

2. Cauces con nombre

2.1 mknod

Llamada al sistema que permite crear archivos especiales, como archivos FIFO o archivos de dispositivo.

```
int mknod (const char *nombre, mode_t modo, dev_t dispositivo);
```

- **nombre**: nombre del fichero especial a crear;
- **modo**: especifica los valores que serán almacenados en el campo st_mode del i-nodo correspondiente al archivo especial, puede tener los siguientes valores:
  - S_IFCHR: archivo de dispositivo orientado a caracteres.
  - S_IFBLK: archivo de dispositivo orientado a bloques.
  - S_IFSOCK: archivo socket.
  - **S_IFIFO: archivo FIFO** (este es el que nos interesa para el cauce).
- **dispositivo**: 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.
- devuelve 0 si ha habido éxito; -1 en caso de error y modifican errno con el tipo de error.

2.2 mkfifo

Llamada al sistema como la anterior pero específica de FIFO, para crear archivos FIFO.

```
int mkfifo (const char *nombre, mode_t modo);
```

- **nombre**: nombre del fichero especial a crear;
- **modo**: especifica los permisos del archivo.
- devuelve 0 si ha habido éxito; -1 en caso de error y modifican errno con el tipo de error.

Los archivos FIFO se eliminan con la llamada al sistema **unlink**.

2.3 unlink

Elimina un nombre y el fichero asociado a dicho nombre.

```
int unlink(const char *nombre);
```

- **nombre**: nombre/ruta del fichero especial a borrar;
- devuelve 0 si ha habido éxito; -1 en caso de error y modifican errno con el tipo de error.

Cuando el nombre hace referencia a un archivo FIFO el nombre es eliminado, pero aquellos procesos que tengan abierto el archivo pueden continuar usándolo.

2.4 Utilización de un cauce FIFO

Las operaciones de E/S (open, close, read y write) sobre un fichero son esencialmente las mismas que las utilizadas con los ficheros regulares, salvo:

- la llamada **lseek** no se puede usar con archivo FIFO, ya que un cauce lee y escribe de forma FIFO, por lo que no tiene sentido mover el current offset a una posición distinta;
- la llamada **read** es **bloqueante** para los procesos que leen cuando no hay datos en el cauce;
- la llamada **read** desbloquea devolviendo 0 cuando los procesos que escribían en el cauce han cerrado el fichero o han terminado el proceso.

2.5 Ejercicio 1

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 <stdio.h>
#include <sys/stat.h>           #include <stdlib.h>
#include <fcntl.h>              #include <errno.h>
#include <unistd.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:
/* ========================== * ========================== */
```

```c
//productorFIFO.c, productor que usa mecanismo de comunicacion FIFO
#include <sys/types.h>          #include <stdio.h>
#include <sys/stat.h>           #include <stdlib.h>
#include <fcntl.h>              #include <errno.h>
#include <unistd.h>             #include <string.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;
}
```

El programa **consumidor/lector:**

- crea el cauce FIFO de nombre ComunicacionFIFO con todos los permisos (umask (0));
- abre el fichero para lectura/escritura;
- lee del cauce hasta que lea la cadena “fin”. Conforme va leyendo, va mostrando lo que lee por pantalla;
- cuando termine de leer, cierra el cauce FIFO, muestra el mensaje leído y termina.

El programa **productor/escritor:**

- recibe por parámetros el mensaje a escribir en el cauce;
- abre el cauce de nombre “ComunicacionFIFO” de sólo escritura;
- escribe en el cauce todo el mensaje;
- cierra el fichero y termina.

Para que funcione correctamente:

- primero debemos ejecutar el programa consumidor, que es el que crea el cauce FIFO de donde va a leer la información. El proceso consumidor queda esperando a que escriba el productor;
- después ejecutamos el productor, pasándole el mensaje por parámetro. Este proceso en cuanto se ejecute escribirá en el cauce.

Queda reflejado en el sistema que hemos usado un cauce FIFO por el archivo FIFO creado de nombre ComunicacionFIFO, ya que este fichero se crea pero no se borra automáticamente (hay que hacerlo de forma específica con unlink).

3. Cauces sin nombre

3.1 pipe

Llamada al sistema que crea un cauce de comunicación sin nombre.

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

- **fd[0]:** descriptor de fichero para lectura;
- **fd[1]:** descriptor de fichero para escritura;
- devuelve 0 si ha habido éxito; -1 en caso de error y modifican errno con el tipo de error.

La llamada pipe crea el cauce sin nombre y asigna en el fd[2] pasado como argumento los descriptores de archivo para lectura y escritura del cauce.

3.2 dup, dup2

Duplican un descriptor de fichero.

```
int dup(int fd);

int dup2(int fd, int fd_new);
```

- **fd:** descriptor de archivo a copiar;
- **fd_new:** descriptor de archivo donde se va a copiar el fd_old;
- devuelve el nuevo descriptor de archivo si ha habido éxito; -1 en caso de error y modifican errno con el tipo de error.
- La diferencia es que dup2 cierra de forma automática fd_new para redirigirlo al nuevo, mientras que con dup hay que realizar un close(fd_new) antes de hacer el duplicado.

dup2 funciona de la siguiente forma:

- Tenemos un programa fichero.txt cuyo descriptor es fd. La tabla de descriptores es la siguiente:

0	Entrada estándar (teclado)
1	Salida estándar (pantalla)
2	Salida estándar de errores (pantalla)
…	…
fd	fichero.txt

- Al hacer dup2(fd,1), que es igual que dup2(fd, STDOUT_FILENO), la tabla de descriptores quedaría:

0	Entrada estándar (teclado)
1	fichero.txt
2	Salida estándar de errores (pantalla)
…	…
fd	fichero.txt

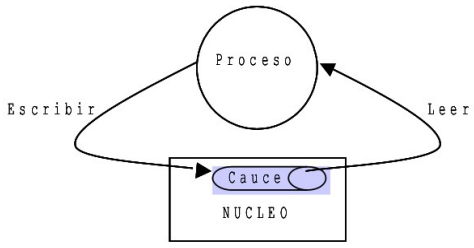
de forma que el descriptor STDOUT_FILENO ya no apunta a la pantalla, sino a fichero.txt. Por tanto, todo lo que se muestre ahora por pantalla, cualquier printf, se almacenará en fichero.txt.

Estas funciones se utilizan en los cauces sin nombre para establecer un proceso como entrada estándar o salida estándar, para leer de un proceso o escribir en otro proceso.

3.2 Esquema de funcionamiento

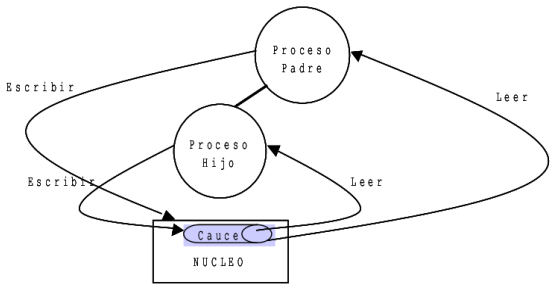
Cuando un procesoP ejecuta pipe, el núcleo instala automáticamente dos descriptores de archivo (**fd**) para usar el cauce creado y los asocia al proceso:

- **fd[0]** se usa para escribir en el cauce (con write);
- **fd[1]** se usa para leer del cauce(con read).



Por tanto, es el procesoP el que puede leer/escribir en el cauce, lo que no tiene sentido (que un proceso se comunique consigo mismo). Este esquema se puede ampliar para que varios procesos, que no comparten memoria, puedan compartir información en el cauce.

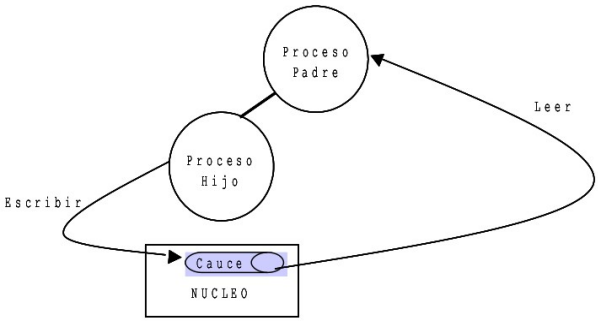
Para ello, el procesoP crea un procesoH hijo (como sabemos, el proceso hijo hereda cualquier descriptor de archivo abierto por el padre), con lo que tenemos dos procesos, procesoP y procesoH, asociados a un cauce, como se ve en la imagen siguiente:



En la situación actual, tanto el padre como el hijo pueden leer y escribir en el cauce, por ello, hay que establecer la dirección en la que los datos van a viajar. Esto es, si queremos que el procesoP lea del cauce y el procesoH escriba en el cauce, la información viajará del hijo al padre, por lo que hay que:

- cerrar en el padre la comunicación de escritura con el cauce;
- cerrar en el hijo la comunicación de lectura con el cauce.

El esquema quedaría de la siguiente forma:



3.3 Creación de cauces

1. Creamos un cauce con la llamada al sistema pipe.
    - En caso de éxito, devuelve 0 y en el fd[2] pasado como argumento devolverá los descriptores de archivo para usar el nuevo cauce → fd[0] lectura, fd[1] escritura.
    - En caso de error, devuelve -1 y modifica la variable errno con el error.
2. Creamos un proceso hijo con fork, que heredará los descriptores de archivos del padre.
3. Establecemos el sentido del flujo de los datos, si hijo → padre ó padre → hijo. En el caso de que sea hijo → padre, se establece el sentido de la siguiente forma:
    - el hijo escribe, pero no lee; por tanto:
        - close(fd[0]) → el hijo debe cerrar la comunicación de lectura del cauce, con lo que cierra el descriptor de archivo de lectura del cauce;
    - el padre lee, pero no escribe; por tanto:
        - close(fd[1]) → el padre debe cerrar la comunicación de escritura del cauce, con lo que cierra el descriptor de archivo de escritura del cauce;
4. Si es necesario, redirigimos la entrada y salida con dup/dup2 al fichero del cauce.

3.4 Notas finales sobre cauces con y sin nombre

Algunos aspectos a tener en cuenta sobre los cauces:

- Para que la comunicación sea tanto de padre a hijo, como de hijo a padre, se crea una **comunicación dúplex** abriendo **dos cauces**.
- La llamada **pipe** debe hacerse **antes de fork**, para que el hijo herede los descriptores del cauce del padre.
- Un cauce, sin o con nombre, debe estar abierto por ambos extremos para permitir la lectura/escritura.
- Cuando el primer proceso que abre un cauce es el proceso lector/consumidor, open bloquea al proceso hasta que algún proceso abra el cauce para escribir.
- Cuando el primer proceso que abre un cauce es el proceso escritor/productor, cada vez que se realice una operación de escritura sin que existan procesos lectores, el sistema envía una señal de SIGPIPE.
- La sincronización entre procesos productores y consumidores es atómica.

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 <stdio.h>
#include <stdlib.h>             #include <errno.h>
#include <fcntl.h>              #include <unistd.h>
#include <string.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);
}
```

El proceso **hijo** escribe en el cauce el mensaje "El primer mensaje transmitido por un cauce!!".

El proceso **padre** lee del cauce el mensaje que ha escrito el hijo y muestra por pantalla el mensaje y el número de bytes leídos.

La salida de la ejecución es:

```
./tarea

        El número de bytes recibidos es: 46

        La cadena enviada a través del cauce es: El primer mensaje transmitido por un cauce!!
```

3.6 Ejercicio 3

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 <stdio.h>
#include <stdlib.h>             #include <errno.h>
#include <fcntl.h>              #include <unistd.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);
}
```

La comunicación del programa va del hijo al padre, por lo que el hijo escribe y el padre lee (hijo → padre).

El proceso **hijo**:

- como es el que escribe, cierra el descriptor de lectura del cauce;
- redirige la salida del cauce a la salida estándar;
- ejecuta la orden ls (listar el directorio actual).

El proceso **padre:**

- como es el que lee, cierra el descriptor de escritura del cauce;
- redirige la entrada del cauce a la entrada estándar, para leer de la salida estándar;
- ejecuta la orden sort, que ordena lo que se le pase.

En este programa:

- el proceso hijo lista el directorio actual, sacándolo por la salida estándar, por lo que será leído por la entrada estándar;
- el proceso padre, lee de la entrada estándar, por lo que lee la salida del hijo;
- el padre, al directorio listado le realiza un sort y ordena la salida del hijo y la muestra por pantalla.

3.7 Ejercicio 4

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 <stdio.h>
#include <stdlib.h>             #include <errno.h>
#include <fcntl.h>              #include <unistd.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
                //estándar (stdout), cerrado previamente en la misma operación
                dup2(fd[1], STDOUT_FILENO);
                execlp("ls","ls",NULL);
        }
        else { // sort. Estoy en el 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);
}
```

Este programa realiza lo mismo que el anterior, pero se diferencian en:

- en el ejercicio 3, se ha usado las órdenes close y dup para establecer las comunicaciones y redirigir la salida y la entrada;
- en este ejercicio, se ha usado la orden dup2 en vez de close y dup, para establecer las comunicaciones y redirigir la salida y la entrada.

La llamada dup2 realiza lo mismo que close y dup juntos, por tanto, ambos programas realiza exactamente lo mismo, utilizando órdenes diferentes para establecer la dirección de la comunicación.

3.8 Ejercicio 5

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.**

esclavo.c:

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

/*Hay que ejecutar:
   gcc esclavo.c -lm -o esclavo
si no, sale el siguiente error:
   esclavo.c:(.text+0x1c): referencia a `sqrt' sin definir
   collect2: error: ld returned 1 exit status*/
```

```c
//función que calcula si n es un número primo; devuelve: 0 si no es primo; 1 si es primo
int esPrimo(int n){
	int primo = 1;

	//recorremos desde 2 < k <= sqrt(n)
	for(int i = 2; i <= sqrt(n) && primo; i++){
		//si es divisible, ya no es primo
		if(n%i == 0){
			primo = 0;
		}
	}
	return primo;
}

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

	int lim_inf, lim_sup;
	char primo[4];

	//comprobamos que se ha pasado como argumento el intervalo
	if(argc < 3){
		perror("ERROR en argumentos. \n \ El formato es ./esclavo limite_inferior limite_superior.\n");
		exit(-1);
	}

	//obtenemos el intervalo
	lim_inf = atoi(argv[1]);
	lim_sup = atoi(argv[2]);

	//comprobamos si han insertado primero el lim sup y después el lim inf
	if(lim_inf > lim_sup){
		lim_inf = lim_sup;
		lim_sup = atoi(argv[1]);
	}

	//calculamos los primos en dicho intervalo
	for(int i = lim_inf; i <= lim_sup; i++){

		//vemos si es primo
		if(esPrimo(i)){
			sprintf(primo, "%i", i);

			//si es primo, lo escribimos por la salida estándar para que lo lea el padre
			write(STDOUT_FILENO, primo, sizeof(int));
		}
	}

	return 0;
}
```

maestro.c:

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

/*El sentido de comunicación será de hijo a padre, el hijo escribe y el padre lee (hijo -> padre)*/
int main(int argc, char *argv[]){

	//intervalo 0 (i0) para los argumentos; intervalo 1 (i1) para el hijo 1, intervalo 2 (i2) para el hijo 2
	int i0[2], i1[2], i2[2];

	//procesos hijo 1 y hijo 2
	pid_t h1, h2;

	//descriptores de fichero del cauce para el hijo 1 y otro para el hijo 2
	int c1[2], c2[2];

	//mensaje que el padre irá leyendo del hijo
	int mensaje = 0;
	char primo[4], arg1[4], arg2[4];

	//comprobamos que se ha pasado como argumento el intervalo
	if(argc < 3){
		perror("ERROR en argumentos. \n \El formato es ./maestro limite_inferior limite_superior.\n");
		exit(-1);
	}

	//obtenemos el intervalo
	i0[0] = atoi(argv[1]);
	i0[1] = atoi(argv[2]);

	//comprobamos si han insertado primero el límite sup y después el límite inf
	if(i0[0] > i0[1]){
		i0[0] = i0[1];
		i0[1] = atoi(argv[1]);
	}

	//hacemos la división de intervalos el intervalo1 será para el hijo1
	i1[0] = i0[0];
	i1[1] = ((i0[0] + i0[1])/2);

	//el intervalo2 será para el hijo2
	i2[0] = i1[1] + 1;
	i2[1] = i0[1];

	/* CAUCE 1, PRIMER HIJO */
	//creamos el cauce1
	pipe(c1);

	//creamos el primer hijo y comprobamos que no ha habido error
	if( (h1 = fork()) < 0){
		perror("ERROR en la creación del hijo 1.\n");
```


```
        exit(-2);
}
//si es el hijo, ejecuta el esclavo
else if(h1 == 0){

        //como el hijo escribe, cerramos el descriptor de lectura del cauce
        close(c1[0]);

        //redirigimos la salida estándar al descriptor del cauce
        dup2(c1[1], STDOUT_FILENO);

        //pasamos los límites a cadena para pasarlos como argumento a exec
        sprintf(arg1, "%i", i1[0]);
        sprintf(arg2, "%i", i1[1]);

        //ejecutamos el esclavo y comprobamos que no ha habido error
        if( execl("./esclavo", "./esclavo", arg1, arg2, (char*) 0) < 0 ){
                perror("ERROR al ejecutar esclavo1\n");
                exit(-3);
        }
}
//si es el padre
else{
        //como el padre lee, cerramos el descriptor de escritura del cauce
        close(c1[1]);

        //redirigimos la entrada estándar al descriptor del cauce
        dup2(c1[0], STDIN_FILENO);

        printf("Hijo 1: \n");
        //leemos de la entrada estándar el primo que calcula el hijo
        while( mensaje = read(c1[0], primo, sizeof(int)) > 0){
                printf("\t%s es primo\n", primo);
        }

        //cuando el padre termina de leer, cerramos el descriptor de lectura
        //para que el cauce se cierre
        close(c1[0]);
}


/* CAUCE 2, SEGUNDO HIJO */
pipe(c2);

//creamos el primer hijo y comprobamos que no ha habido error
if( (h2 = fork()) < 0){
        perror("ERROR en la creación del hijo 2.\n");
        exit(-2);
}
//si es el hijo, ejecuta el esclavo
else if(h2 == 0){
```

```
                //como el hijo escribe, cerramos el descriptor de lectura del cauce
                close(c2[0]);
                //redirigimos la salida estándar al descriptor del cauce
                dup2(c2[1], STDOUT_FILENO);
                //pasamos los límites a cadena para pasarlos como argumento a exec
                sprintf(arg1, "%i", i2[0]);
                sprintf(arg2, "%i", i2[1]);

                //ejecutamos el esclavo y comprobamos que no ha habido error
                if( execl("./esclavo", "./esclavo", arg1, arg2, (char*) 0) < 0 ){
                        perror("ERROR al ejecutar esclavo1\n");
                        exit(-3);
                }
        }
        //si es el padre
        else{
                //como el padre lee, cerramos el descriptor de escritura del cauce
                close(c2[1]);

                //redirigimos la entrada estándar al descriptor del cauce
                dup2(c2[0], STDIN_FILENO);

                printf("Hijo 2: \n");
                //leemos de la entrada estándar el primo que calcula el hijo
                while( mensaje = read(c2[0], primo, sizeof(int)) > 0){
                        printf("\t%s es primo\n", primo);
                }
                //cuando el padre termina de leer, cerramos el descriptor de lectura
                //para que el cauce se cierre
                close(c2[0]);
        }

        return 0;
}
```

Para compilar:

$ gcc esclavo.c -lm o esclavo (-lm para que no haya problema con la función sqrt)

$ gcc -o maestro maestro.c

La salida con el intervalo [0, 20]:

$ ./maestro 0 20

Hijo 1:
- 0 es primo
- 1 es primo
- 2 es primo
- 3 es primo
- 5 es primo
- 7 es primo

Hijo 2:
- 11 es primo
- 13 es primo
- 17 es primo
- 19 es primo

4. Preguntas de repaso

1. **¿Cuál es la diferencia entre un cauce sin nombre y uno con nombre?**
   - Sin nombre son temporales y con nombre se crea un archivo FIFO.
   - Se crean con diferentes llamadas al sistema.
   - En los cauces sin nombre, el cauce finaliza cuando termina la comunicación, cuando se cierran los dos descriptores del cauce; en el cauce con nombre, el cauce no termina cuando se cierra el archivo FIFO, sino que se puede volver a él y para eliminar el cauce, hay que eliminar el archivo FIFO.
   - El cauce sin nombre sólo puede ser usado por el proceso que lo crea y sus descendientes, mientras que en el cauce con nombre puede ser usado por cualquier proceso que abra el archivo FIFO.

2. **¿Qué funciones se utilizan para crear cada uno de los cauces (sin nombre y con nombre)?**

   Con cauce: mknod/mfifo.

   Sin nombre: pipe.

3. **¿Cómo podríamos programar la eliminación de un cauce con nombre?**

   unlink(ARCHIVO_FIFO); donde ARCHIVO_FIFO es el nombre del fichero del cauce.

4. **¿Cómo podríamos redireccionar la salida y la entrada estándar de un proceso?**

   dup2(fd[1], STDOUT_FILENO), con lo que se establece que el fichero fd sea la salida estándar, por tanto, todo lo que se ejecute después se escribirá en el descriptor fd[1];

   dup2(fd[0], STDIN_FILENO), con lo que se establece que el fichero fd sea la entrada estándar, por tanto, todo lo que se ejecute después se leerá del descriptor fd[0].