APUNTES MÓDULO II

Sesión uno:

Todos los archivos abiertos están identificados por **descriptores de archivo** (entero no negativo).

Descriptores de archivo:

- **STDIN_FILENO:** descriptor de archivo 0 (entrada estándar).
- **STDOUT_FILENO:** descriptor de archivo 1 (salida estándar).
- **STDERR_FILENO:** descriptor de archivo 2 (salida de error estándar).

Tipos de archivo:

- **Archivo regular:** contiene datos de cualquier tipo.
- **Archivo de directorio:** un directorio es un archivo que contiene los nombres de otros archivos y punteros a la información de dichos archivos.
- **Archivo especial de dispositivo de caracteres:** representan ciertos tipos de dispositivos en un sistema.
- **Archivo especial de dispositivo de bloques:** representan discos duros.
- **FIFO:** un tipo de archivo utilizado para comunicación entre procesos (cauce con nombre).
- **Enlace simbólico:** tipo de archivo que apunta a otro archivo.
- **Socket:** tipo de archivo usado para comunicación en red entre procesos.

Estructura STAT: con la llamada al sistema **stat** obtenemos los metadatos de un archivo. Tiene la siguiente representación:

```
struct stat {
dev_t st_dev; /* nº de dispositivo (filesystem) */
dev_t st_rdev; /* nº de dispositivo para archivos especiales */
ino_t st_ino; /* nº de inodo */
mode_t st_mode; /* tipo de archivo y mode (permisos) */
nlink_t st_nlink; /* número de enlaces duros (hard) */
uid_t st_uid; /* UID del usuario propietario (owner) */
gid_t st_gid; /* GID del usuario propietario (owner) */
off_t st_size; /* tamaño total en bytes para archivos regulares */
unsigned long st_blksize; /* tamaño bloque E/S para el sistema de archivos*/
unsigned long st_blocks; /* número de bloques asignados */
time_t st_atime; /* hora último acceso */
time_t st_mtime; /* hora última modificación */
time_t st_ctime; /* hora último cambio */
};
```

- Para comprobar el **tipo de fichero**:
  - **S_ISLNK (st_mode):** Verdadero si es un *enlace simbólico*.
  - **S_ISREG (st_mode):** Verdadero si es un *archivo regular*.
  - **S_ISDIR (st_mode):** Verdadero si es un *directorio*.
  - **S_ISCHR (st_mode):** Verdadero si es un *dispositivo de caracteres*.
  - **S_ISBLK (st_mode):** Verdadero si es un *dispositivo de bloques*.
  - **S_ISFIFO (st_mode):** Verdadero si es un *cauce con nombre* (FIFO).
  - **S_ISSOCK (st_mode):** Verdadero si es un *socket*.

- Valor de **st_blocks**: tamaño del fichero en bloques de 512 bytes.

Permisos de acceso a archivos:

El valor **st_mode** codifica los permisos de acceso al archivo, independientemente del tipo de archivo de que se trate.

Categorías: user, group y other.

Modos de apertura:

- El permiso de lectura para un archivo determina si podemos abrir para lectura un archivo existente: los flags **O_RDONLY** y **O_RDWR** para la llamada **open**.
- El permiso de escritura para un archivo determina si podemos abrir para escritura un archivo existente: los flags **O_WRONLY** y **O_RDWR** para la llamada **open**.

Llamada **open**: devuelve un descriptor de fichero. Sus parámetros son, en este orden, una ruta y un modo de apertura.

Llamada **read**: devuelve un entero que representa a los bytes leídos. Sus parámetros son, en este orden, un descriptor de fichero (el de lectura), un vector donde almacenar lo que se va a leer, y el tamaño en bytes que queremos leer.

Llamada **write**: sus parámetros son, en este orden, un descriptor de fichero (el de escritura), el vector donde ha sido almacenado lo leído, y el entero donde se ha almacenado lo que ha devuelto **read**.

Flags que comprueban permisos:

- **S_IRWXU:** user tiene permisos de lectura, escritura y ejecución.
- **S_IRUSR:** user tiene permiso de lectura.
- **S_IWUSR:** user tiene permiso de escritura.
- **S_IXUSR:** user tiene permiso de ejecución.
- **S_IRWXG:** group tiene permisos de lectura, escritura y ejecución.
- **S_IRGRP:** group tiene permiso de lectura.
- **S_IWGRP:** group tiene permiso de escritura.
- **S_IXGRP:** group tiene permiso de ejecución.
- **S_IRWXO:** other tiene permisos de lectura, escritura y ejecución.
- **S_IROTH:** other tiene permiso de lectura.
- **S_IWOTH:** other tiene permiso de escritura.
- **S_IXOTH:** other tiene permiso de ejecución.

Sesión dos:

Llamada al sistema **umask**: máscara de creación de permisos para el proceso y que devuelve el valor previamente establecido.

mode_t umask (mode_t mask);

Descripción:

- Establece la máscara de usuario a mask & 0777.
- La máscara de usuario es creada por **open** para establecer los permisos iniciales del archivo que se va a crear.

Llamada al sistema **chmod** y **fchmod**: estas dos funciones nos permiten cambiar los permisos de acceso para un archivo que existe en el sistema de archivos. La llamada **chmod** actúa sobre un archivo especificado por su pathname mientras que **fchmod** opera sobre un archivo que ha sido previamente abierto con **open**.

int chmod (const char *path, mode_t mode);

int fchmod (const char *path, mode_t mode);

Manejo de directorios:

- **opendir:** se le pasa el pathname del directorio a abrir y devuelve un puntero a la estructura de tipo DIR.
- **readdir:** lee la entrada donde esté situado el puntero de lectura de un directorio ya abierto. Después de la lectura adelanta el puntero una posición. Devuelve la entrada leída a través de un puntero a una estructura o devuelve NULL si llega al final del directorio o se produce error.
- **closedir:** cierra un directorio, devolviendo 0 si tiene éxito, -1 en caso contrario.

Breve explicación de cómo manejar los permisos:

atributos.st_mode & S_IWOTH == S_IWOTH: sirve para comprobar si ese permiso ya está.

chmod (atributos.st_mode | S_IWOTH): sirve para añadir un permiso.

atributos.st_mode & ~S_IWOTH: sirve para quitar permiso.

Ejercicios:

1. **Crear dos ficheros y copiar lo que hay en uno, en el otro.**

```c
int main(){
    int fd1, fd2; //descriptores de fichero
    int bytesleidos; //almacenamos lo que devuelva read
    char bloque[512]; //se almacenará lo leido

    //abrimos fichero 1 con el modo apertura de solo lectura
    if((fd1 = open("fichero1", O_RDONLY)) < 0){
        perror("ERROR AL ABRIR EL FICHERO 1");
        exit(1);
    }

    //abrimos el fichero 2 con el modo apertura de escritura
    //ya que es donde vamos a copiar el contenido del fichero 1
    if((fd2 = open("fichero2", O_WRONLY)) < 0){
        perror("ERROR AL ABRIR EL FICHERO 2");
        exit(1);
    }

    //en bloque almacenamos el contenido de fd1
    if((bytesleidos = read(fd1, bloque, 512)) < 0){
        perror("ERROR AL LEER");
        exit(1);
    }

    //y lo copiamos en fd2
    if(write(fd2, bloque, bytesleidos) < 0){
        perror("ERROR AL ESCRIBIR");
        exit(1);
    }

    //cerramos los descriptores
    close(fd1);
    close(fd2);
}
```

2. **Programa que recorra todas las entradas de los ficheros regulares contenidos en el directorio actual.**

```c
int main(){
    DIR *dir; //directorio
    struct dirent *ent; //entradas del directorio
    struct stat atributos; //atributos

    //abrimos el directorio
    if((dir = opendir("./")) = NULL){
        perror("ERROR AL ABRIR EL DIRECTORIO");
        exit(1);
    }

    //recorremos las entradas del directorio
    while((ent = readdir(dir)) != NULL){

        //entramos en el fichero y accedemos a los atributos
        if(stat(ent -> d_name, &atributos) < 0){
            perror("ERROR AL ACCEDER A LOS ATRIBUTOS");
            exit(1);
        }

        //comprobamos que sean regulares
        if(S_ISREG(atributos.st_mode)){

            //mostramos el nombre
            printf("Archivo %s\n", ent -> d_name);
        }
    }
    closedir(dir); //cerrramos el directorio
}
```

3. **Programa que recorra todas las entradas de los ficheros regulares contenidos en el parámetro que le pasamos.**

```c
int main(int argc, char *argv[]){
    DIR *dir; //directorio
    struct dirent *ent; //entradas del directorio
    struct stat atributos; //atributos
    char ruta[512]; //ruta del directorio

    //comprobacion de argumentos
    if(argc != 2){
        printf("ERROR: NÚMERO INCORRECTO DE ARGUMENTOS");
        exit(1);
    }
    //abrimos el directorio pasado por argumento
    if((dir = opendir(argv[1])) = NULL){
        perror("ERROR AL ABRIR EL DIRECTORIO");
        exit(1);
    }
    //recorremos las entradas del directorio
    while((ent = readdir(dir)) != NULL){

        //no se muestra por pantalla, sino que guarda en "ruta" lo que le digamos
        sprintf(ruta, "%s/%s", argv[1], ent -> d_name);

        //entramos en el fichero y accedemos a los atributos
        if(stat(ruta, &atributos) < 0){ //se carga en "ruta" los atributos/permisos
            perror("ERROR AL ACCEDER A LOS ATRIBUTOS");
            exit(1);
        }
        //comprobamos que sean regulares
        if(S_ISREG(atributos.st_mode)){

            //mostramos el nombre
            printf("Archivo %s\n", ent -> d_name);
        }
    }
    closedir(dir); //cerrramos el directorio
}
```

4. **Se recibe un directorio por argumento, hay que mirar los archivos regulares de dicho directorio y comprobar que tienen todos permiso de escritura a otros.**

```c
int main(int argc, char *argv[]){
    DIR *dir; //directorio
    struct dirent *ent; //entradas del directorio
    struct stat atributos; //atributos
    char ruta[512]; //ruta del directorio
    umask(0);

    //comprobacion de argumentos
    if(argc != 2){
        printf("ERROR: NÚMERO INCORRECTO DE ARGUMENTOS");
        exit(1);
    }
    //abrimos el directorio pasado por argumento
    if((dir = opendir(argv[1])) = NULL){
        perror("ERROR AL ABRIR EL DIRECTORIO");
        exit(1);
    }
    //recorremos las entradas del directorio
    while((ent = readdir(dir)) != NULL){
        //no se muestra por pantalla, sino que guarda en "ruta" lo que le digamos
        sprintf(ruta, "%s/%s", argv[1], ent -> d_name);
        //entramos en el fichero y accedemos a los atributos
        if(stat(ruta, &atributos) < 0){ //se carga en "ruta" los atributos/permisos
            perror("ERROR AL ACCEDER A LOS ATRIBUTOS");
            exit(1);
        }
        //comprobamos que sean regulares
        if(S_ISREG(atributos.st_mode)){
            //añadimos permiso de escritura a otros
            if(chmod(ruta, atributos.st_mode | S_IWOTH) < 0){
                perror("ERROR AL ESTABLECER LOS PERMISOS");
                exit(1);
            }
        }
    }closedir(dir); //cerrramos el directorio
}
```

5. **Se recibe un directorio por argumento, hay que mirar los archivos regulares de dicho directorio y los que no tengan permiso de ejecución para usuario, se lo vamos a añadir.**

```c
int main(int argc, char *argv[]){
    DIR *dir; //directorio
    struct dirent *ent; //entradas del directorio
    struct stat atributos; //atributos
    char ruta[512]; //ruta del directorio
    umask(0);

    //comprobacion de argumentos
    if(argc != 2){
        printf("ERROR: NÚMERO INCORRECTO DE ARGUMENTOS");
        exit(1);
    }
    //abrimos el directorio pasado por argumento
    if((dir = opendir(argv[1])) = NULL){
        perror("ERROR AL ABRIR EL DIRECTORIO");
        exit(1);
    }

    while((ent = readdir(dir)) != NULL){
        //no se muestra por pantalla, sino que guarda en "ruta" lo que le digamos
        sprintf(ruta, "%s/%s", argv[1], ent -> d_name);
        //entramos en el fichero y accedemos a los atributos
        if(stat(ruta, &atributos) < 0){ //se carga en "ruta" los atributos/permisos
            perror("ERROR AL ACCEDER A LOS ATRIBUTOS");
            exit(1);
        }
        //compruebo que sean regulares y que no tengan permiso de ejecucion para usuario
        if(S_ISREG(atributos.st_mode) && ((atributos.st_mode & S_IXUSR) != S_IXUSR)){
            //a los permisos que tiene le añado los de ejecucion para usuario
            if (chmod(ruta, atributos.st_mode | S_IXUSR) < 0){
                perror("ERROR AL ESTABLECER LOS PERMISOS");
                exit(1);
            }
        }
    }closedir(dir);
}
```

6. **Se recibe un directorio por argumento, hay que mirar los archivos regulares de dicho directorio y los que tengan permiso de ejecución para el usuario se lo vamos a quitar.**

```c
int main(int argc, char*argv[]){
    DIR *dir; //directorio
    struct dirent *ent; //entradas del directorio
    struct stat atributos; //atributos
    char ruta[512]; //ruta del directorio
    umask(0);

    //comprobacion de argumentos
    if(argc != 2){
        printf("ERROR: NÚMERO INCORRECTO DE ARGUMENTOS");
        exit(1);
    }
    //abrimos el directorio pasado por argumento
    if((dir = opendir(argv[1])) = NULL){
        perror("ERROR AL ABRIR EL DIRECTORIO");
        exit(1);
    }

    while((ent = readdir(dir)) != NULL){
        //no se muestra por pantalla, sino que guarda en "ruta" lo que le digamos
        sprintf(ruta, "%s/%s", argv[1], ent -> d_name);
        //entramos en el fichero y accedemos a los atributos
        if(stat(ruta, &atributos) < 0){ //se carga en "ruta" los atributos/permisos
            perror("ERROR AL ACCEDER A LOS ATRIBUTOS");
            exit(1);
        }
        //compruebo que sean regulares y que no tengan permiso de ejecucion para usuario
        if(S_ISREG(atributos.st_mode) && ((atributos.st_mode & S_IXUSR) != S_IXUSR)){
            //quito el permiso de ejecucion para usuario
            if (chmod(ruta, atributos.st_mode & ~S_IXUSR) < 0){
                perror("ERROR AL ESTABLECER LOS PERMISOS");
                exit(1);
            }
        }
    }closedir(dir);
}
```

Sesión tres:

Cada proceso tiene un único identificador de proceso (PID) que es un número entero no negativo. Existen algunos procesos especiales como el **init** (PID = 1). El proceso **init** es el encargado de inicializar el sistema y ponerlo a disposición de los programas de aplicación.

El proceso **init** es el proceso raíz de la jerarquía de procesos del sistema, que se genera debido a las relaciones entre proceso creador (padre) y proceso creado (hijo).

```
#include <unistd.h>
#include <sys/types.h>

pid_t getpid(void); // devuelve el PID del proceso que la invoca.

pid_t getppid(void); // devuelve el PID del proceso padre del proceso que
                     // la invoca.

uid_t getuid(void); // devuelve el identificador de usuario real del
                    // proceso que la invoca.

uid_t geteuid(void); // devuelve el identificador de usuario efectivo del
                     // proceso que la invoca.

gid_t getgid(void); // devuelve el identificador de grupo real del proceso
                    // que la invoca.

gid_t getegid(void); // devuelve el identificador de grupo efectivo del
                     // proceso que la invoca.
```

Llamada al sistema **fork**: crea un proceso hijo, el cual es una copia casi idéntica del proceso padre, que es el proceso que hace la llamada a **fork**. Devuelve un valor distinto en cada uno de los procesos, padre e hijo. La única diferencia entre los valores devueltos es que en el proceso hijo el valor es 0 y en el proceso padre el valor es el PID del hijo. La razón por la que el identificador del nuevo proceso hijo se devuelve al padre es porque un proceso puede tener más de un hijo, y de esta forma podemos identificar los distintos hijos. La razón por la que **fork** devuelve un 0 al proceso hijo se debe a que un proceso solamente puede tener un único padre.

- **getpid:** conozco mi PID.
- **getppid:** conozco el PID de mi padre.

Llamada al sistema **exec**: Un posible uso de la llamada **fork** es la creación de un proceso (el hijo) que ejecute un programa distinto al que está ejecutando el programa padre, utilizando para esto una de las llamadas al sistema de la familia **exec**. Cuando un proceso ejecuta una llamada **exec**, el espacio de direcciones de usuario del proceso se reemplaza completamente por un nuevo espacio de direcciones.

Se le pasan los parámetros por lista o vector.

execl ("nombre", "primer argumento", "segundo argumento", NULL); (igual para execlp).

Llamada **clone**: permite crear procesos e hilos con un grado mayor en el control de sus propiedades.

Uno de los argumentos más interesantes de **clone** son los indicadores de clonación (argumento flags). Estos tienen dos usos, el byte de orden menor sirve para especificar la señal de terminación del hijo, que normalmente es **SIGCHILD**. Si está a cero, no envía señal. Una diferencia con **fork( ),** es que en ésta no podemos seleccionar la señal, que siempre es **SIGCHILD**.

Ejercicios:

1. **Proceso que crea 10 hijos y muestra el pid de cada uno por pantalla.**

```c
int main(){
    pid_t pid;
    int i;

    for(i = 0; i < 10; i++){
        if((pid = fork()) < 0){
            perror("ERROR EN LA CREACIÓN DE HIJOS");
            exit(1);
        }

        //si es el proceso hijo
        else if(pid == 0){
            printf("%u,%u\n", getpid(), getppid());
            exit(0);
        }
    }
}
```

2. **Programa que reciba como argumento un entero "x". Vamos a recorrer un directorio buscando la entrada número "x" y vamos a ejecutar un "ls" sobre dicha entrada. Cuando termine el "ls" vamos a mostrar un mensaje: "FINALIZADO". Si la entrada no existe, da un error y finaliza.**

```c
int main(int argc, char *argv[]){
    DIR *dir; //directorio
    struct dirent *ent; //entradas del directorio
    int contador = 0;

    //comprobacion de argumentos
    if(argc != 2){
        printf("ERROR: NÚMERO INCORRECTO DE ARGUMENTOS");
        exit(1);
    }

    int x = atoi(argv[1]);

    //abrimos el directorio pasado por argumento
    if((dir = opendir(argv[1])) = NULL){
        perror("ERROR AL ABRIR EL DIRECTORIO");
        exit(1);
    }
    //recorremos las entradas del directorio
    while((ent = readdir(dir)) != NULL){
        contador ++;

        if(contador == x){
            execlp("ls", "ls", ent -> d_name, NULL);
            perror("ERROR EN EL EXEC");
            exit(1);
        }

        printf("FINALIZADO\n");
    }

    fprintf(stderr, "NO SE HA ENCONTRADO ESA ENTRADA");
    closedir(dir);
}
```

Sesión cuatro:

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.

Hay dos tipos de cauces, cauces sin y con nombre.

Cauces con nombre:

- Se crean en el sistema de archivos en disco como un archivo especial.
- Los procesos abren y cierran un archivo FIFO usando su nombre mediante las llamadas al sistema **open** y **close**.
- Cualesquiera procesos pueden compartir datos utilizando las ya conocidas llamadas al sistema **read** y **write**.

Para crear un archivo FIFO podemos hacer uso de la llamada al sistema **mknod**.

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

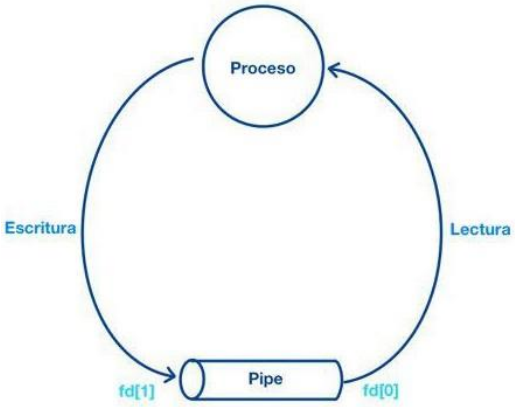
La llamada al sistema **mknod** crea un archivo especial de nombre FILENAME. El parámetro MODE especifica los valores que serán almacenados en el campo **st_mode** del i-nodo correspondiente al archivo especial:

- **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. Para crear un cauce FIFO el valor de este argumento será 0.

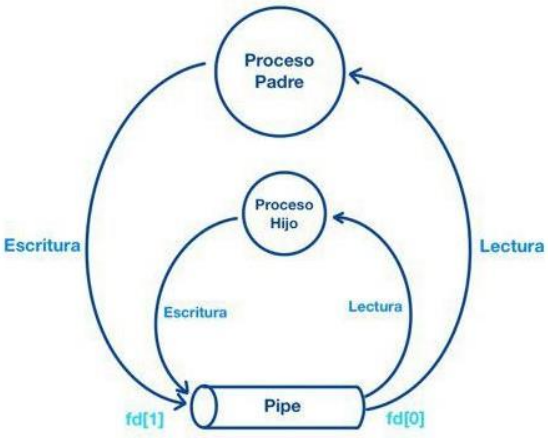
Cauces sin nombre:

- 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 solo pueden ser utilizados como mecanismo de comunicación entre proceso que crea el cauce sin nombre y los procesos descendientes creados a partir de la creación de un cauce.

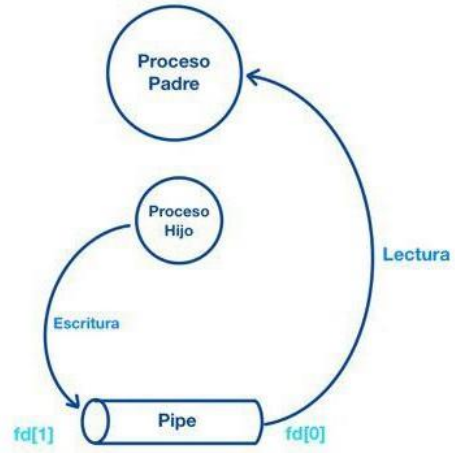


Un descriptor se usa para permitir un camino de envío de datos (escritura / **write**) al cauce, y el otro descriptor se usa para obtener los datos (lectura / **read**) de éste.

La gracia de este mecanismo es utilizarlo para la comunicación entre un proceso padre y un proceso hijo, ya que el proceso hijo hereda cualquier descriptor abierto por el padre.



En este momento, debemos tomar una decisión: ¿en qué sentido van a viajar los datos? Una vez decidido, debemos cerrar los extremos que no son necesarios (uno para cada proceso). Por ejemplo, si es el proceso hijo el que va a hacer algún tipo de procesamiento y va a devolverle la información al padre usando el cauce, el esquema quedaría así:



Crearemos el cauce mediante la llamada al sistema **pipe**, la cual tiene como argumento un vector de dos enteros **int fd[2]**, dicho vector contiene dos descriptores de archivo. El primer elemento, **fd[0]** se usa como descriptor de archivo de lectura, y el segundo elemento, **fd[1]**, se usa para la escritura.

Una vez creado el cauce, creamos el proceso hijo y establecemos el sentido del flujo de datos. Cabe decir que la llamada al sistema **pipe** debe realizarse siempre antes que la llamada **fork**, ya que, si no se hace así, el proceso hijo no heredará los descriptores del cauce.

Como los descriptores son compartidos por el proceso padre y el hijo, debemos cerrar con la llamada al sistema **close** el extremo del cauce que no nos interese en cada uno de los procesos. 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]**). En caso contrario, si 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]**).

Llamada al sistema **dup**: 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** junto con anterioridad a **dup** con el objetivo de dejar la entrada deseada libre. Recordar que el **descriptor de archivo 0** (STDIN_FILENO) de cualquier proceso 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.

Llamada al sistema **dup2** (la que más se usa): permite una atomicidad 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.

Ejercicios:

1. **Programa que recibe como argumento el nombre de un directorio. Se va a crear un pipe. El proceso hijo va a recorrer todas las entradas del directorio y las va a contar. Va a escribir en el cauce cuantos elementos hay y el padre va a leer lo que hay en el cauce y lo mostrará por pantalla.**

```
int main(int argc, char *argv[]){
    DIR *dir;
    struct dirent *ent;
    int contador = 0,
    pid_t pid;
    int fd[2];

    if(argc != 2){
        printf("ERROR DE ARGUMENTOS");
        exit(1);
    }

    if((pipe(fd)) < 0){
        perror("ERROR EN EL PIPE");
        exit(1);
    }

    if((pid=fork()) < 0){
        perror("ERROR EN LA CREACIÓN DE HIJOS");
        exit(1);
    }

    else if(pid == 0){
        close(fd[0]);
        if((dir = opendir(argv[1])) == NULL){
            perror("ERROR AL ABRIR EL DIRECTORIO");
            exit(1);
        }

        while((ent = readdir(dir)) != NULL){
            contador ++;
        }

        if(write(fd[1], &contador, sizeof(int))){
            perror("ERROR AL ESCRIBIR");
            exit(1);
        }

    }
    else{
        close(fd[1]);
        if(read(fd[0], &contador, sizeof(int)) < 0){
            perror("ERROR AL LEER");
            exit(1);
        }

        printf("Hay %d entradas \n", contador);
    }
}
```

2. **Programa que recibe como argumento el nombre de un directorio y recorre ese directorio en busca de ficheros regulares. Para cada fichero regular que encuentre, el padre escribe el nombre del fichero y se lo manda al hijo a través de un cauce pipe, y el hijo por cada nombre de fichero que reciba comprueba si tiene permiso de lectura para usuarios, si lo encuentra muestra el propietario del fichero, y si no lo encuentra, muestra el tamaño en bytes de dicho fichero.**

```
int main(int argc, char *argv[]){
    DIR *dir; //directorio
    struct dirent *ent; //entradas del directorio
    struct stat atributos; //atributos
    char ruta[512]; //ruta
    int fd[2]; //pipe

    pid_t pid;
    int leidos;

    //comprobacion de argumentos
    if(argc != 2){
        printf("ERROR: NÚMERO INCORRECTO DE ARGUMENTOS");
        exit(1);
    }

    if(pipe(fd) < 0){
        perror("ERROR EN EL PIPE");
        exit(1);
    }

    if((pid = fork()) < 0){
        perror("ERROR EN LA CREACIÓN DE HIJOS");
        exit(1);
    }
    //si es el padre
    else if(pid != 0){

        //cerramos lectura
        close(fd[0]);
        if((dir = opendir(argv[1])) == ){
            perror("ERROR AL ABRIR EL DIRECTORIO");
            exit(1);
        }

        //leemos las entradas
        while((ent = readdir(dir)) != NULL){
            sprintf(ruta,"%s/%s", argv[1], ent -> d_name);

            //entra en el fichero y accedemos a los atributos
            if(stat(ruta, &atributos) < 0){
                perror("ERROR AL ACCEDER A LOS ATRIBUTOS");
                exit(1);
            }

            //compruebo que sean regulares
            if(S_ISREG(atributos.st_mode)){

                //escribimos en el cauce
                if(write(fd[1], ruta, strlen(ruta)*sizeof(char)) < 0){
                    perror("ERROR AL ESCRIBIR");
                    exit(1);
                }
            }
        }
        //cerramos directorio
        closedir(dir);
    }
    else{ //si es el hijo, se leen todos los ficheros pasados por el padre
        //cerramos escritura
        close(fd[1]);

        //leemos del cauce (read devuelve un entero)
        while((leidos = read(fd[0], ruta, sizeof(char)*512)) < 0){

            //entramos en el fichero y accedemos a los atributos
            if(stat(ruta,&atributos) < 0){
                perror("ERROR AL ACCEDER A LOS ATRIBUTOS");
                exit(1);
            }
```

```
            //comprobamos que tenga permiso de lectura para usuario
            if((atributos.st_mode & S_IRUSR) == S_IRUSR){

                //mostramos el nombre del propietario
                printf("Nombre del identificador de fichero %u\n", atributos.st_mode);
            }
            //mostramos el tamaño de fichero
            else{
                printf("Tamaño en bytes del fichero %u\n", atributos.st_size);
            }
        }
    }
    //nos aseguramos de que el proceso no queda zombie
    wait(NULL);

}
```

3. **Programa donde el proceso padre lee las entradas de un directorio actual y envía el nombre de cada entrada por el pipe. El hijo, para cada entrada comprueba si es un fichero o un directorio. Si es un fichero, muestra su nombre y sus permisos, y si es un directorio, muestra su nombre y el uid.**

```
int main(){
    DIR *dir;
    struct dirent *ent;
    struct stat atributos;

    char fichero[256];
    pid_t pid;
    int fd[2];


    if(pipe(fd) < 0){
        perror("ERROR EN EL PIPE");
        exit(1);
    }

    if((pid = fork()) < 0){
        perror("ERROR EN LA CREACIÓN DE HIJOS");
        exit(1);
    }
    else if(pid != 0){
        close(fd[0]);
        if((dir = opendir("./")) == NULL){
            perror("ERROR AL ABRIR EL DIRECTORIO");
            exit(1);
        }

        while((ent=readdir(dir)) != NULL){
            printf("Entrada: %s\n", ent -> d_name);
            if(write(fd[1], ent -> d_name, sizeof(ent -> d_name)) < 0){
                perror("ERROR AL ESCRIBIR");
                exit(1);
            }
        }
        close(fd[1]);
    }

    else{
        close(fd[1]);
        while(read(fd[0], fichero, sizeof(char)*256) > 0){
            if(stat(fichero, &atributos) < 0){
                perror("ERROR AL ACCEDER A LOS ATRIBUTOS");
                exit(1);
            }

            if(S_ISDIR(atributos.st_mode)){
                printf("Nombre: %s, uid: %d\n", fichero, atributos.st_uid);
            }
            else if(S_ISREG(atributos.st_mode)){
                printf("Nombre: %s, permisos: %o\n", fichero, atributos.st_mode);
            }
        }
        close(fd[0]);
        exit(0);
    }
    wait(NULL);
}
```

4. **Programa que recorra todas las entradas de un directorio (pasado como argumento) y por cada una de ellas ha de crear un hijo que recibirá como argumento la ruta correspondiente de la entrada y enviará al padre el uid, gid y el número de inodo. El padre mostrará por pantalla los valores recibidos por el hijo.**

```c
struct Estructura{
    ino_t inodo;
    uid_t uid;
    gid_t gid;
};

int main(int argc, char *argv[]){
    DIR *dir; //directorio
    struct dirent *ent; //entradas
    struct stat atributos; //atributos
    char ruta[512];

    int fd[2];
    pid_t pid;
    int contador = 0;
    struct Estructura estr;

    if(argc != 2){
        perror("ERROR EN LOS ARGUMENTOS");
        exit(1);
    }

    if(pipe(fd) < 0){
        perror("ERROR EN EL PIPE");
        exit(1);
    }

    if((dir = opendir(argv[1])) == NULL){
        perror("ERROR AL ABRIR EL DIRECTORIO");
        exit(1);
    }

    while((ent = readdir(dir)) != NULL){
        sprintf(ruta, "%s/%s", argv[1], ent -> d_name);

        if((pid = fork()) < 0){
            perror("ERROR EN LA CREACIÓN DE HIJOS");
            exit(1);
        }

        else if(pid == 0){
            close(fd[0]);
            if(stat(ruta, &atributos) < 0){
                perror("ERROR AL ACCEDER A LOS ATRIBUTOS");
                exit(1);
            }

            estr.inodo = atributos.st_ino;
            estr.uid = atributos.st_uid;
            estr.gid = atributos.st_gid;

            if(write(fd[1], &estr, sizeof(struct Estructura)) < 0){
                perror("ERROR AL ESCRIBIR");
                exit(1);
            }

            exit(0);
        }
        else{
            contador ++;
        }
    }
```

```
    int i;
    for(i = 0; i < contador; i++){
        if(read(fd[0], &estr, sizeof(struct Estructura)) < 0){
            perror("ERROR AL LEER");
            exit(1);
        }

        printf("uid: %u, gid: %u, inodo: %id\n", estr.uid, estr.gid, estr.inodo);
    }
}
```

5. **Programa que ejecute "cut –f1 –d: /etc/passwd | grep argv[1]". El hijo hace el "grep" y el padre pasa los datos del "cut" al hijo.**

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

    if(argc != 2){
        printf("ERROR EN LOS ARGUMENTOS");
        exit(1);
    }

    if(pipe(fd) < 0){
        perror("ERROR EN EL PIPE");
        exit(1);
    }

    if((pid = fork()) < 0){
        perror("ERROR EN LA CREACIÓN DE HIJOS");
        exit(1);
    }

    if(pid != 0){
        close(fd[0]);
        dup2(fd[1], STDOUT_FILENO);
        execlp("cut", "cut", "-f1", "-d: ", "etc/passwd", NULL);
        close(fd[1]);
    }
    else{
        close(fd[1]);
        dup2(fd[0], STDIN_FILENO);
        execlp("grep", "grep", argv[1], NULL);
        close(fd[0]);
        exit(1);
    }

    wait(NULL);
}
```