MÓDULO II: USO DE LOS SERVICIOS DEL SO MEDIANTE LA API

Sesión 1: Llamadas al sistema para el Sistema de Archivos (Parte I)

1. Entrada/Salida de archivos regulares

Lo que se trabaja en esta sesión es como abrir, leer y escribir en un archivo mediante el uso de llamadas al sistema. Las funciones que vamos a usar para trabajar con los archivos son: **open, read, write, lseek y close.** Estas funciones se conocen como entrada/salida sin búfer.
Para poder reconocer un archivo ya abierto tenemos los descriptores de archivos "fd" *file descriptor.* Estos son enteros no negativos que devuelven las funciones open y creat.

- Entrada estándar de un proceso (**STDIN_FILENO)**: tiene asociado por convenio el "fd" 0
- Salida estándar de un proceso(**STDOUT_FILENO)**: tiene asociado por convenio el "fd" 1
- Salida de error estándar de un proceso (**STDERR_FILENO**): tiene asociado por convenio el "fd" 2

Cuando leemos un archivo, por defecto se empieza a leer desde el byte 0, pero esto lo podemos cambiar especificando con la opción **O_APPEND.** Esto se puede hacer usando la llamada al sistema lseek.

Llamadas de gestión y procesamiento de archivos regulares:

- **Open:** *int **open** (const char *pathname, int flags, mode_t mode)*. La llamada al sistema open puede usarse bien para crear un archivo o bien para abrirlo. En el caso de la creación (solo se creará el archivo cuando en el campo flags así lo indiquemos con O_CREAT. Si esta bandera no aparece se ignorará el campo mode y no se creará), debemos entender bien la relación entre la máscara **umask** y el campo **mode** que es el que nos permite establecer los permisos de dicho archivo. Hay una serie de flags que debemos tener en cuenta a la hora de crear un archivo como es **O_TRUNC (**consultar en el manual en línea con *man open).* Los permisos del archivo se crean de la siguiente forma: (modo & ~umask). Un ejemplo de como calcular la máscara (consultar mediante la orden *man open* los diferentes permisos):
  Pasamos como argumento en el campo mode:

S_IRUSR	S_IWUSR	S_IRGRP	S_IWGRP	S_IROTH	S_IWOTH
400	200	040	020	004	002

  Los convertimos a binario y hacemos un OR lógico con todos ellos:

100 000 000
010 000 000
000 100 000
000 010 000
000 000 100
000 000 010
110 110 110

  Este número binario se corresponde con 0666. Por defecto umask vale 022 que se corresponde con S_IWGRP | S_IWOTH. Por tanto cuando aplicamos la máscara esos

permisos desaparecerán. Tal y como dice la explicación tenemos que hacer *mode & ~umask* → 0666 & ~022, que significa 0666 and lógico complemento de 022.
Si lo hacemos (AND lógico):

110 110 110
111 101 101
110 100 100

Este número binario es **0644** en octal. Por tanto los permisos del archivo que hemos creado son: rw-r--r--. Como ya he dicho antes los permisos de escritura de group y others viene por defecto en umask, por tanto si pasamos esos flags por parámetro se anularán. También nos podemos dar cuenta que los permisos resultantes son la suma de los permisos en octal que hemos pasado como argumento en *mode,* ojo, teniendo en cuenta lo anterior, los permisos de escritura de grupo y otros se van.

- **Close:** *int **close** (int fd)* La orden **close** se utiliza para cerrar un descriptor de archivo (fd) de forma que este deje de hacer referencia al archivo al que apuntaba (un determinado archivo se identifica por el descriptor de archivo que fue devuelto por open/creat). Usamos esta orden cuando terminamos un programa y le pasamos por parámetros el descriptor de archivo para desreferenciarlo.

- **Lseek:** *off_t **lseek** (int fd, off_t offset, int whence)* La función de esta llamada al sistema es reposicionar el **file_offset** (indica el número de bytes que se va a desplazar el indicador desde una posición que viene dada por **whence,** que puede tener los siguientes valores: **SEEK_SET** (se situará en el offset que se le pase como segundo argumento a la función), **SEEK_END** (final del fichero)**, SEEK_CUR** (se situará donde se encontraba el archivo más el offset que le hayamos pasado como segundo argumento a la función), la posición del archivo donde nosotros queramos. Básicamente el lseek es muy útil cuando queremos escribir algo en una determinada posición del archivo, por ejemplo, contando desde el principio, avanzamos 40B en el archivo, para que el fd apunte a esa posición, y luego con un write escribir lo que queramos. Esto se haría de la siguiente forma: **lseek(fd, 40, SEEK_SET)**
  Cabe destacar en cuanto a lseek, que si por ejemplo el archivo tiene 30B de tamaño y nostros queremos saltar a 40B, esos 10B de diferencia se van a llenar con caracteres nulos, se deja como uan especie de "hueco" en espera de ser rellenado.

- **Read:** *ssize_t **read** (int fd, void *buf, size_t count).* Esta llamada al sistema lee de un archivo abierto indicado por "fd" count bytes y lo vuelca en buf. Es decir:
  char * buf
  read (fd, buf, 10)
  En este sencillo ejemplo vamos a leer del archivo previamente abierto 10 bytes y esa información la vamos poner en buf. Si hay éxito la función devuelve el número de bytes leídos y la posición del archivo se avanza en ese número de bytes, es decir, si vamos a volver a leer el archivo lo hará 10 bytes más adelante que antes.

- **Write:** *ssize_t* ***write*** *(int fd, const void *buf, size_t count)* Esta llamada al sistema nos sirve para escribir lo que tengamos guardado en un determinado buffer desde la dirección a la que apunta el fd hasta contar los bytes que se le pasan por el tercer parámetro. Con el siguiente ejemplo se verá más claro:
  char buf []=”abcdefghij”;
  write(fd, buf, 10)
  Con esto escribiríamos los 10 caracteres de buf desde la posición indicada por fd.
  La llamada al sistema write devuelve el número de Bytes escritos si la operación se ha completado con éxito, si no devuelve -1. Es **importante** tener en cuenta que para consultar esta llamada al sistema en el manual en línea se hace con la orden ***man 2 write.*** Si no especificamos

2. Metadatos de un archivo

Antes hemos visto las llamadas al sistema más básicas sobre archivos regulares, ahora vamos a centrarnos en las características de dichos archivos y sus propiedades. En primer lugar nos vamos a centrar en estudiar la estructura **stat**:

- **Stat:** Para obtener los metadatos de un archivo lo podemos hacer mediante la llamada al sistema stat. Esta nos proporcionará una estructura llamada stat que tienen 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 */
};
```

En concreto st_blocks da el tamaño del fichero en bloques de 512B, mientras que st_blksize da el tamaño de bloque más eficiente para realizar operaciones E/S. En cuanto a los flags existentes para poder trabajar con el campo **st_mode** (Es importante tener en cuenta que este campo no solo indica el tipo de archivo del que se trata sino que, también nos indica los permisos que tenemos sobre este)**:**

S_IFREG	0100000	archivo regular (no POSIX)
S_IFBLK	0060000	dispositivo de bloques (no POSIX)
S_IFDIR	0040000	directorio (no POSIX)
S_IFCHR	0020000	dispositivo de caracteres (no POSIX)
S_IFIFO	0010000	cauce con nombre (FIFO) (no POSIX)
S_ISUID	0004000	bit SUID
S_ISGID	0002000	bit SGID
S_ISVTX	0001000	sticky bit (no POSIX)
S_IRWXU	0000700	**user** (propietario del archivo) tiene permisos de lectura, escritura y ejecución
S_IRUSR	0000400	user tiene permiso de lectura (igual que S_IREAD, no POSIX)
S_IWUSR	0000200	user tiene permiso de escritura (igual que S_IWRITE, no POSIX)
S_IXUSR	0000100	user tiene permiso de ejecución (igual que S_IEXEC, no POSIX)
S_IRWXG	0000070	**group** tiene permisos de lectura, escritura y ejecución
S_IRGRP	0000040	group tiene permiso de lectura
S_IWGRP	0000020	group tiene permiso de escritura
S_IXGRP	0000010	group tiene permiso de ejecución
S_IRWXO	0000007	**other** tienen permisos de lectura, escritura y ejecución
S_IROTH	0000004	other tienen permiso de lectura
S_IWOTH	0000002	other tienen permiso de escritura
S_IXOTH	0000001	other tienen permiso de ejecución

Por ejemplo un archivo cuyo st_mode se inicializa así: S_IFREG | S_IRWXU , quiere decir que es un archivo regular con permisos de escritura, lectura y ejecución para el usuario.
Por otro lado tenemos los siguiente POSIX para saber de qué tipo de archivo se trata:

```
S_ISLNK(st_mode) Verdadero si es un enlace simbólico (soft)
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 una cauce con nombre (FIFO)
S_ISSOCK(st_mode) Verdadero si es un socket
```

En cuanto a los **permisos** que tiene que ver con directorios de archivos destacaría que el permiso de lectura nos deja ver todos los archivos que hay en el directorio y el de ejecución nos permite que ese directorio lo utilicemos en un pathname que le pasemos a una orden, para acceder a uno de los archivos del directorio por ejemplo.
Otra cosa que debemos tener en cuenta es que debemos tener permisos de escritura en un archivo para poder especificar el flag O_TRUNC en la llamada open.

Como apunte, la diferencia entre **stat** y **lstat** es que **stat()** devuelve los atributos o la información de inodo sobre el archivo de destino asociado con el pathname del archivo. Esta información se devuelve en el puntero búfer del segundo argumento de la función. Mientras que, **lstat** () funciona exactamente igual que la primera solo que el primer argumento es un enlace simbólico, de manera que devuelve información sobre el enlace en sí, no del archivo.

int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);

Sesión 2: Llamadas al sistema para el Sistema de Archivos (Parte II)

Es importante para entender esta sesión conocer bien cómo funciona umask y los valores que tiene por defecto. esto viene explicado en la sesión 1.

-Las llamadas al sistema chmod y fchmod

Antes que nada es importante no confundirlas con las que venimos usando en fundamentos del software, estas son chmod(1). Al fin y al cabo tienen la misma funcionalidad pero no se usan de la misma manera. Las que vamos a explicar se encuentran en *man 2 chmod*.

int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);

Estas dos funciones nos sirven para cambiar los permisos de acceso a un archivo que ya existe. La llamada **chmod** opera sobre un archivo indicado por su *pathname mientras que **fchmod** opera sobre un archivo que ha sido previamente abierto con open y por tanto es necesario pasarle como primer argumento su descriptor de archivo.

En el campo mode de la función vendrían los flags que hemos explicado en la sesión anterior pero que de todas formas podéis consultar en el manual en línea usando la orden ***man 2 chmod.***

En caso de que la función se ejecute correctamente, devolverá 0, en caso de error, -1.

-Funciones de manejo de directorios

Realmente podríamos usar las llamadas al sistema que hemos visto ahora para trabajar con directorios pero esto puede causar una serie de problemas y para ello se establece una biblioteca estándar de funciones de manejo de directorios (es importante siempre que trabajemos con estas funciones incluir las librerías <sys/types.h> y <dirent.h>, en esta segunda viene definida la estructura DIR, donde vienen todos los datos de un directorio):

- **opendir:** *DIR ***opendir**(const char *name).* Se le pasa como argumento la ruta de un directorio y se devuelve un puntero a estructura DIR. Para que nos hagamos una idea de lo que hay dentro de esta estructura:

```
typedef struct _dirdesc {
        int dd_fd;
        long dd_loc;
        long dd_size;
        long dd_bbase;
        long dd_entno;
        long dd_bsize;
        char *dd_buf;
} DIR;
```

- **readdir:** *struct dirent ***readdir**(DIR *dirp)*. Lee la entrada donde está situado el puntero de lectura de un directorio ya abierto que se le pasa a la función. Después de

realizar la lectura, adelanta el puntero una posición. Esta función devuelve la entrada leída a través de una estructura (struct dirent), o devuelve NULL si llega al final o hay un error.

```
struct dirent {
      long d_ino; /* número i-nodo */
      char d_name[256]; /* nombre del archivo */
};
```

- **closedir:** *int **closedir**(DIR *dirp)*. Esta función se utiliza para cerrar un directorio. Devuelve 0 si hay tenido éxito, -1 si no.
- **seekdir:** *void **seekdir**(DIR *dirp, long loc).* Permite situar el puntero de lectura de un directorio. El argumento loc debe ser el valor devuelto por la función *telldir.*
- **telldir:** *long **telldir**(DIR *dirp*). Devuelve la posición del puntero de lectura de un directorio. Si tiene éxito devuelve la posición a la que apunta **dirp** si no devuelve -1.
- **rewinddir:** *void **rewinddir** (DIR *dirp).* Posiciona el puntero de lectura al principio del directorio.

-Llamada a nftw para recorrer un sistema de archivos.

Con las funciones que acabamos de presentar podemos recorrer un árbol de directorios manualmente, sin embargo, con la llamada a la función **nftw()** podemos recorrer recursivamente un árbol de directorios y hacer operaciones sobre sus archivos.

*int **nftw** (const char *dirpath, int (*func) (const char *pathname, const struct stat *statbuf, int typeflag, struct FTW *ftwbuf), int nopenfd, int flags);*

A continuación la explicación de los argumentos de esta función:

1. *dirpath: directorio a partir del cual hacemos el recorrido recursivo (hacemos el recorrido en las capas inferiores de este)
2. *func: función que es llamada en cada entrada del árbol (recorrido en preorden, es decir primero los directorios que están más abajo). Como podemos ver a esta función se le pasan 4 argumentos:
    a. *pathname: ruta de la entrada
    b. *statbuf: es un puntero a una estructura stat que se devuelve tras la llamada stat para *pathname.
    c. typeflag: entero (hay ya unos flags definidos como FTW_F) que indica diferentes valores. Ver en el manual las diferentes banderas usando la orden *man nftw.*

    d. *ftwbuf: es un puntero a la estructura FTW descrita a continuación:

```
struct FTW {
      int base;
      int level;
}
```

Base indica el offset del archivo y level la profundidad a la que se encuentra con respecto a *pathname.

3. nopendf: entero que especifica el número de descriptores de archivo que se pueden utilizar.
4. flags: este se crea con OR, igual que hacíamos para el mode cuando abrimos o creamos un archivo. A continuación los más importantes:

FTW_DIR	Realiza un **chdir** (cambia de directorio) en cada directorio antes de procesar su contenido. Se utiliza cuando **func** debe realizar algún trabajo en el directorio en el que el archivo especificado por su argumento **pathname** reside.
FTW_DEPTH	Realiza un recorrido postorden del árbol. Esto significa que **nftw** llama a **func** sobre todos los archivos (y subdirectorios) dentro del directorio antes de ejecutar **func** sobre el propio directorio.
FTW_MOUNT	No cruza un punto de montaje.
FTW_PHYS	Indica a **nftw** que nos desreferencie los enlaces simbólicos. En su lugar, un enlace simbólico se pasa a **func** como un valor **typedflag** de FTW_SL.

La función se encarga de recorrer el árbol de directorios especificado en **dirpath** y la a la función func, definida por el usuario, para cada archivo del árbol. Por defecto nftw realiza un recorrido no ordenado en preorden de forma que procesa primero todos los directorios antes de procesar los archivos y los subdirectorios.

Mientras se va recorriendo el árbol, **nftw** va abriendo un descriptor de archivo por nivel del árbol. Con el parámetro *nopenfd*(explicado arriba) definimos el número máximo de fd que se pueden utilizar. Si hay más niveles que fd's permitidos, se van cerrando unos para abrir otros.

Esta llamada al sistema devuelve 0 si recorre el árbol completamente, -1 si hay error o el primer valor no nulo devuelto por func.

Sesión 3: Llamadas al sistema para el control de Procesos

Esta sesión está dedicada a trabajar con las llamadas relacionadas con el control y la gestión de procesos.

-Creación de procesos:

Como ya sabemos cada proceso tiene un PID único. Existen procesos especiales como el init (PID=1) que es el proceso encargado de inicializar el sistema UNIX y ponerlo a disposición de los programas de aplicación. Este proceso no termina hasta que se detiene el Sistema Operativo y es un proceso normal solo que con privilegios superusuario (root).

Además del identificador de un proceso existen los siguientes identificadores asociados al proceso:

```
#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.
```

Para crear un proceso, esto se debe hacer desde otro proceso e invocando a la llamada al sistema **fork.** Este proceso creado se denomina como proceso hijo. Esta llamada devuelve un valor distinto para cada uno de los procesos. Para el proceso **hijo** el valor devuelto será 0 y para el **padre** será el pid del hijo. Esta es la forma más sencilla de identificar a un proceso hijo ya que el padre puede tener varios.

Por otro lado tenemos las llamadas al sistema **wait, waitpid y exit.** Estas llamadas se complementan con las de creación de procesos.

*pid_t **wait**(int *wstatus);*

*pid_t **waitpid**(pid_t pid, int *wstatus, int options);*

Estas llamadas se usan para esperar el cambio de estado de un proceso hijo y obtener información de dicho proceso. Es considerado como cambio de estado: el hijo termina, el hijo se interrumpe por una señal, o el hijo continúa su ejecución después de recibir una señal. Esta llamada permite al sistema liberar todos los recursos empleados en la ejecución del proceso hijo. Si no se hace, el hijo entra en estado “zombie”.
La primera llamada (wait) suspende la ejecución de la hebra que lanza la llamada hasta que el proceso hijo termine. La segunda (waitpid) hace lo mismo pero con unas opciones adicionales que vienen explicadas en el manual en línea el cual se puede consultar con la orden ***man wait.*** El primer argumento de esta segunda llamada es un pid que puede tomar diferentes valores y que cada uno significa una cosa diferente.

-Familia de llamadas al sistema exec

Un posible uso de la llamada fork es la creación de un proceso (hijo) que ejecute un programa distinto al que está ejecutando el padre, utilizando la llamada a la familia de **exec**.

Cuando un proceso ejecuta una llamada a **exec**, el espacio de direcciones de usuario del proceso se sustituye por uno nuevo, el del programa que se le pasa como argumento, que pasa a ejecutarse en el contexto de proceso hijo (empezando por el main). El PID del proceso se mantiene.

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ...);
int execlp (const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

El primer argumento es siempre el path del archivo ejecutable. El **const char *arg** y los puntos suspensivos de **execl, execlp** y **execle** son los argumentos que se le pasan: arg0,arg1… Es importante destacar que la lista de argumentos tiene que terminar en NULL.

Las funciones **execv** y **execvp** proporcionan un vector de punteros a cadenas de caracteres terminadas en 0, que representan la lista de argumentos disponible para el nuevo programa. Como siempre el primer argumento es el nombre del archivo y la lista de argumentos tiene que acabar en NULL.

En cuanto a la función **execle,** especifica el entorno del proceso que ejecutará el programa mediante un parámetro adicional que va detrás del puntero NULL. Este parámetro adicional es un vector de punteros a cadenas de caracteres acabadas en 0 y debe ser terminada por NULL.

Las funciones exec fallarán de forma generalizada si no hay suficiente espacio para crear el nuevo espacio de direcciones o si no se pasan correctamente los argumentos.

-La llamada clone

Añadimos un concepto nuevo, **tid.** Esto es el identificador de hebra de un proceso. La llamada al sistema **clone,** permite crear procesos e hilos, al igual que fork, pero con un mayor grado de control de sus propiedades. Para poder usarla debemos hacerlo de la siguiente forma

#define _GNU_SOURCE
#include <sched.h>
*int **clone** (int (*func) (void *), void *child_stack, int flags, void *func_arg, /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */)*

Lo que aparece entre /**/ significa que es opcional en esta llamada. Si esta tiene éxito devuelve el pid del proceso creado, sino -1.

Cuando invocamos a clone, creamos un nuevo proceso que empieza a ejecutar la función ***func*** que se le pasa como primer argumento (fork, sin embargo seguía ejecutando la siguiente instrucción del programa). A esta función se le pasa como argumento ***func_arg.*** El hijo finaliza cuando dicha función realiza un exit. Por otro lado, a la función clone es necesario pasarle otro argumento, ***child_stack** que es una pila que previamente le hemos tenido que reservar.

Uno de los argumentos más interesantes de clone() son los indicadores de clonación (***flags***). estos tienen dos usos, el byte de orden menor sirve para indicar la terminación del hijo (SIGCHLD) (en clone() no se puede modificar esta señal) y los bytes restantes para los siguientes flags:

Indicador	**Significado**
CLONE_CHILD_CLEARTID	Limpia tid (thread identifier) cuando el hijo invoca a exec() o _exit().
CLONE_CHILD_SETTID	Escribe el tid de hijo en ctid
CLONE_FILES	Padre e hijo comparten la tabla de descriptores de archivos abiertos.
CLONE_FS	Padre e hijo comparten los atributos relacionados con el sistema de archivos (directorio raíz, directorio actual de trabajos y máscara de creación de archivos)
CLONE_IO	El hijo comparte el contexto de E/S del padre.
CLONE_NEWIPC	El hijo obtiene un nuevo namespace System V IPC
CLONE_NEWNET	El hijo obtiene un nuevo namespace de red
CLONE_NEWNS	El hijo obtiene un nuevo namespace de montaje
CLONE_NEWPID	El hijo obtiene un nuevo namespace de PID
CLONE_NEWUSER	El hijo obtiene un nuevo namespace UID
CLONE_NEWUTS	El hijo obtiene un nuevo namespace UTS
CLONE_PARENT	Hace que el padre del hijo sea el padre del llamador.
CLONE_PARENT_SETTID	Escribe el tid del hijo en ptid
CLONE_PID	Obsoleto, utilizado solo en el arranque del

	sistema
CLONE_PTRACE	Si el padre está siendo traceado, el hijo también
CLONE_SETTLS	Describe el almacenamiento local (tls) para el hijo
CLONE_SIGHAND	Padre e hijo comparten la disposición de las señales
CLONE_SYSVSEM	Padre e hijo comparten los valores para deshacer semáforos
CLONE_THREAD	Pone al hijo en el mismo grupo de hilos del padre
CLONE_UNTRACED	No fuerza CLONE_PTRACE en el hijo
CLONE_VFORK	El padre es suspendido hasta que el hijo invoca a exec()
CLONE_VM	Padre e hijo comparten el espacio de memoria virtual

Sesión 4: Comunicación entre procesos utilizando cauces

En esta sesión se va a trabajar con los mecanismos de comunicación de información entre procesos. Estos mecanismos son conocidos como ***pipes*** o en español tuberías.

-Concepto y tipos de cauce

Un cauce se define como un mecanismo para la comunicación de información y sincronización entre procesos. Los datos pueden ser enviados por varios procesos al cauce y recibidos por otros procesos desde dicho cauce.

La comunicación a través de un cauce sigue el paradigma de productor/consumidor a través de un buffer: existen procesos que generan datos (productores) y otros que los toman (recolectores). Estos datos se tratan en orden FIFO. Cuando un dato del cauce es leído por un consumidor, se elimina del cauce. De esta forma, un proceso que está esperando a que le lleguen unos datos determinados por el cauce, se bloquea hasta que los recibe. Los cauces son unidireccionales, es decir, si los procesos quieren comunicarse en otro sentido deben usar otro cauce.

Hay dos tipos de cauces en los sistemas operativos UNIX, cauces con y sin nombre. Los que nosotros conocemos son sin nombre y estos son un método que unen la salida estándar de un proceso a la entrada estándar de otro. Por ejemplo:

ls | sort

La orden ls nos muestra los archivos del directorio en el que nos encontramos. Cuando digo nos muestra, quiere decir que imprime por pantalla (salida estándar) los archivos del directorio actual. El operador | nos indica la comunicación con cauces, por tanto la salida estándar se modifica y se une con la entrada estándar para sort. Por tanto esta orden unida con cauce nos muestra los archivos del directorio actual pero ordenados.

Las características de los cauces **sin nombre** son:

- No tienen un archivo asociado en el sistema de archivos en el disco
- Al crear un cauce sin nombre, se usa la llamada al sistema pipe, donde se devuelven dos descriptores de archivo, 0 para lectura y 1 para escritura. La llamada se haría así: *pipe(fd),* con fd declarado como *int fd[2]*. No hace falta usar la llamada al sistema open.
- Los cauces sin nombre solo pueden ser usados entre el proceso que lo crea y los procesos descendientes de este
- Los cauces se cierran automáticamente por el núcleo una vez se han usado.

Las características de un **cauce con nombre** o archivo FIFO son las siguientes:

- Estos se crean mediante la llamada al sistema **mknod o mkfifo.**
- Los procesos por tanto van a tener que hacer uso de las llamadas open y close para poder trabajar con estos archivos.
- También se usarán las llamadas write y read para compartir la información.
- El archivo FIFO permanece en el sistema de archivos aunque esté vacío hasta que se borre explícitamente con la llamada al sistema **unlink.**

-Cauces con nombre

- **Creación de archivos FIFO**

Una vez creado el cauce con nombre cualquier proceso puede abrirlo para lectura/escritura, de la misma forma que un archivo regular. Sin embargo, el cauce debe estar abierto por ambos extremos antes de realizar cualquier operación. Abrir un FIFO para lectura produce un bloqueo hasta que otro proceso abra el mismo cauce para escritura.

Para crear un archivo FIFO en C usamos la orden:

*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 st_mode. Para crear un archivo de tipo FIFO, en el campo MODE debemos de usar **S_IFIFO**.

El argumento **DEV** especifica a qué dispositivo se refiere el archivo especial. Por ejemplo, para crear un cauce FIFO el valor de este argumento será 0. Un ejemplo de creación de cauce FIFO es: ***mknod**(“/tmp/FIFO”,S_IFIFO | 0666,0)*

En el caso del ejemplo de arriba, se crearía un archivo FIFO /tmp/FIFO con los permisos 0666. Debemos tener en cuenta que la mascara de permisos final tendrá esta forma:

umaskFinal = MODE & ~umaskInicial

siendo umaskInicial la máscara de permisos almacenada por el sistema por defecto.

La llamada al sistema mknod permite crear cualquier tipo de archivo especial pero nosotros nos queremos centrar en la creación de archivos FIFO. Estos tienen su propia llamada que usa la siguiente sintaxis:

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

Con esta llamada se crea un archivo FIFO de nombre FILENAME y el argumento MODE, establece los permisos que el queremos dar al archivo.

- **Utilización de un cauce FIFO**

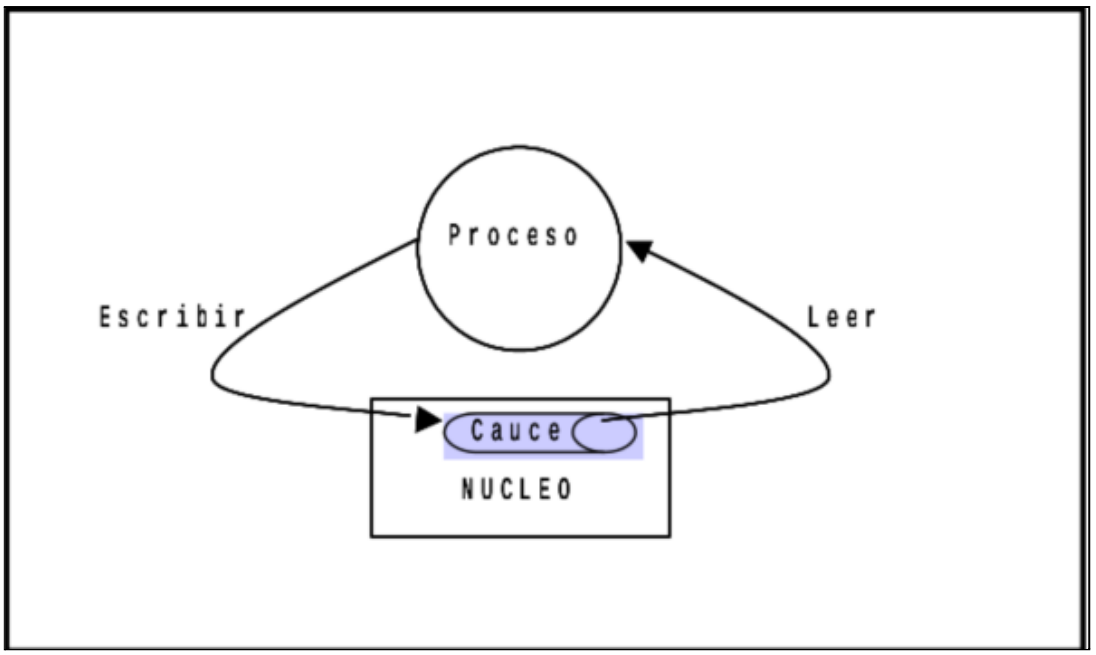
Las operaciones de E/S que podemos hacer sobre este tipo de archivo son prácticamente las mismas que hemos visto durante este módulo. La única diferencia es que no podemos hacer uso de la llamada **lseek** ya que esta nos permite colocarnos en un sitio concreto del archivo. Esto no tiene sentido en un archivo FIFO porque se rompería su filosofía de First In First Out. El resto de llamadas (open, close, read, write) se usan de la misma manera. Debemos tener en cuenta:

- La llamada al sistema read es bloqueante para los procesos consumidores cuando no hay datos que leer en el cauce
- Read desbloquea devolviendo 0 (ningún byte leído), cuando los procesos que actuaban como productores lo han cerrado o terminado.

-Cauces sin nombre

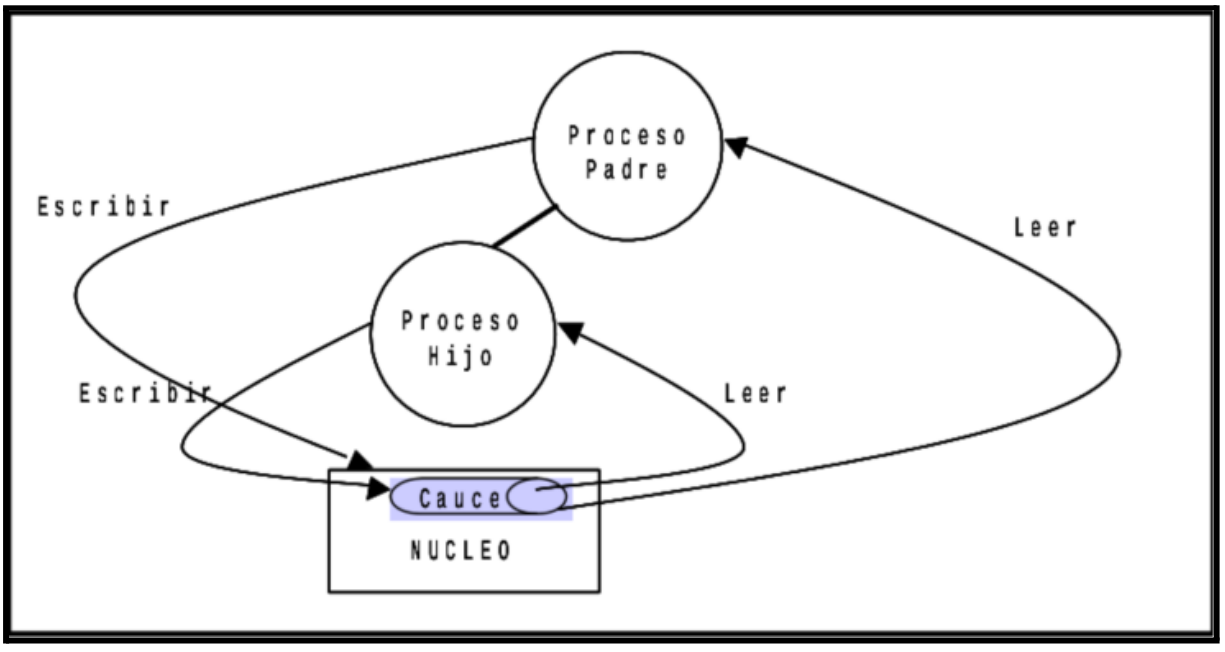
- **Esquema de funcionamiento**

A continuación vamos a explicar lo que ocurre a nivel núcleo cuandos e cera un cauce sin nombre. Al ejecutar la llamada al sistema **pipe** para crear un cauce sin nombre, el núcleo automáticamente crea dos descriptores de archivos que asigna a dicho cauce, uno que se utiliza para el envío de datos (**write**) y otro para la lectura de los mismos (**read**).



En el esquema superior podemos observar como el procesos puede usar los dos descriptores para enviar datos (llamada al sistema write) al cauce y leerlos desde este con la llamada al sistema read. Sin embargo, este mecanismo es útil para comunicar procesos, por tanto es necesario ampliar el esquema, para que el mismo proceso no actúe como productor y consumidor al mismo tiempo.

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 padre e hijo.



En este momento se debe tomar la decisión de quien envía a quién los datos, si bien el padre envía y el hijo recibe, o si por el contrario el padre es quien recibe los datos que el hijo envía.

- **Creación de cauces:**

Para la creación de cauces sin nombre usaremos la llamada al sistema **pipe.** Esta toma como argumento un vector de dos enteros (que van a ser descriptores de archivo) **int fd[2].** Si esta llamada tiene éxito, el vector contendrá dos nuevos valores. Para **fd[0]** un descriptor de sólo lectura y para **fd[1]** un descriptor de solo escritura.
Una vez que hemos creado el cauce debemos decidir la dirección de comunicación entre el proceso padre hijo, si bien sea :

1. Hijo → padre: el hijo manda información al padre por tanto, para el hijo cerramos el descriptor de lectura y para el padre el de escritura.

```
if (pid == 0){          //recordamos que si pid es 0 se trata del proceso hijo
        close(fd[0]);
        ….
}
else{
        close(fd[1]);
        ….
}
```

2. Padre → hijo: el padre manda información al hijo y por tanto será al revés que en el caso anterior.

```
if (pid == 0){          //recordamos que si pid es 0 se trata del proceso hijo
        close(fd[1]);
        ….
}
else{
        close(fd[0]);
        ….
}
```

Si queremos redireccionar la entrada o salida estándar al descriptor de lectura o escritura de un 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.
Si consultamos el manual en línea de estas dos órdenes nos fijamos que vienen incluídas en la librería <unistd.h>

*int **dup** (int oldfd)*
*int **dup2** (int oldfd, int newfd)*

¿Cómo funcionan estas llamadas?

1. dup: crea una copia del descriptor de archivo *oldfd* usando el menor de los descriptores (que no se estén usando) para el nuevo descriptor de archivo. Vamos a ver un ejemplo para entenderlo mejor:

```
/*
en este programa lo que vamos a hacer es implementar un cauce entre el proceso hijo que envía información al proceso padre redireccionando la salida y entrada estándar
*/
int fd[2];
pid_t PID;
PID = fork();
pipe(fd);
if (PID == 0) {
        //cerramos este descriptor porque el proceso hijo solo escribe
        close(fd[0])
        //dejamos libre el descriptor de archivo de la salida estándar
        close(STDOUT_FILENO);
        /*
        según la definición de dup, la duplicación del descriptor de archivo se hará en aquel descriptor de menor valor que no se esté usando, es decir, el de salida (recordamos que el descriptor de la salida estándar es 0, el de entrada 1 y el de error es 2)
        */
        //redireccionamos la salida con la salida estándar
        dup(fd[1]);

}
```

2. dup2: En este caso no hace falta cerrar el descriptor de salida o entrada estándar pues lo indicamos como argumento en la propia llamada. Si quisiéramos hacer lo mismo que hemos hecho en el ejemplo anterior podríamos omitir la llamada close con la salida estándar y sustituir la orden dup por:

```
dup2(fd[1], STDIN_FILENO);
```

Sesión 5: Llamadas al sistema para gestión y control de señales

En esta sesión se van a trabajar con las señales que podemos enviar a un proceso y cómo podemos manipular para un proceso lo que realiza esa señal. Esto lo programaremos nosotros dentro de nuestros programas.
Las señales constituyen un mecanismo básico de sincronización que usa el núcleo de Linux para indicar a los procesos la ocurrencia de determinados eventos síncronos o asíncronos con su ejecución. A parte del núcleo, los procesos pueden enviarse señales entre sí.
Un manejador de señal es una función que se invoca cuando se manda una señal a un programa.
No es lo mismo una señal bloqueada que una señal ignorada. Esta última es desechada por el proceso mientras que la primera permanece pendiente hasta que el proceso las desbloquee.
La lista de señales y su tratamiento por defecto aparece en el manual en línea ejecutando ***man 7 signal.*** Aunque también, si queremos ver todas las señales y el número entero con el que se representan podemos hacer ***kill -l.***

Símbolo	**Acción**	**Significado**
SIGHUP	Term	Desconexión del terminal (referencia a la función termio(7) del man). También se utiliza para reanudar los demonios init, httpd e inetd. Esta señal la envía un proceso padre a un proceso hijo cuando el padre finaliza.
SIGINT	Term	Interrupción procedente del teclado
SIGQUIT	Core	Terminación procedente del teclado
SIGILL	Core	Excepción producida por la ejecución de una instrucción ilegal
SIGABRT	Core	Señal de aborto procedente de la llamada al sistema
SIGFPE	Core	Excepción de coma flotante
SIGKILL	Term	Señal para terminar un proceso (no se puede ignorar ni manejar).
SIGSEGV	Core	Referencia inválida a memoria
SIGPIPE	Term	Tubería rota: escritura sin lectores
SIGALRM	Term	Señal de alarma procedente de la llamada al sistema
SIGTERM	Term	Señal de terminación
SIGUSR1	Term	Señal definida por el usuario (1)
SIGUSR2	Term	Señal definida por el usuario (2)

SIGCHLD	Ign	Proceso hijo terminado o parado
SIGCONT	Cont	Reanudar el proceso si estaba parado
SIGSTOP	Stop	Parar proceso (no se puede ignorar ni manejar).
SIGTSTP	Stop	Parar la escritura en la tty
SIGTTIN	Stop	Entrada de la tty para un proceso de fondo
SIGTTOU	Stop	Salida a la tty para un proceso de fondo

Term: La acción por defecto es terminar el proceso
Ign: L acción por defecto es ignorar la señal
Core: La acción por defecto es terminar el proceso y realizar un volcado de memoria
Stop: La acción por defecto es detener el proceso
Cont: La acción por defecto es continuar con la ejecución del proceso

Tenemos una serie de llamadas al sistema para poder trabajar con señales en Linux. Estas son: **kill, sigaction, sigprocmask, sigpending, sigsuspend**

-Llamada al sistema KILL
Esta llamada al sistema se usa para enviar una señal a un proceso o grupo de procesos. Si lo hacemos desde la terminal podemos consultar como hacerlo con *man kill,* si lo hacemos desde un programa es en *man 2 kill* y su función sería la siguiente:

*int **kill** (pid_t pid, int sig)*

Debemos tener en cuenta que:

• Si pid es positivo, entonces se envía la señal sig al proceso con identificador de proceso igual a pid. En este caso, se devuelve 0 si hay éxito, o un valor negativo si se produce un error.
• Si pid es 0, entonces sig se envía a cada proceso en el grupo de procesos del proceso actual.
• Si pid es igual a -1, entonces se envía la señal sig a cada proceso, excepto al primero, desde los números más altos en la tabla de procesos hasta los más bajos.
• Si pid es menor que -1, entonces se envía sig a cada proceso en el grupo de procesos -pid.
• Si sig es 0, entonces no se envía ninguna señal, pero sí se realiza la comprobación de errores

-Llamada al sistema SIGACTION

Esta llamada al sistema se emplea para cambiar la acción tomada por un proceso cuando recibe una determinada señal.

*int **sigaction**(int signum, const struct sigaction *act, struct sigaction *oldact);*

El significado de los parámetros de la llamada es el siguiente:
• **signum** especifica la señal y puede ser cualquier señal válida salvo SIGKILL o SIGSTOP.
• Si **act** no es NULL, la nueva acción para la señal signum se instala como act.
• Si **oldact** no es NULL, la acción anterior se guarda en oldact.
Devuelve 0 en caso de éxito, -1 en caso de error.
Para saber cómo cambiar la acción por defecto de una señal (para las señales **SIGKILL y SIGSTOP** no se puede modificar su acción) necesitamos conocer por encima la estructura de ***sigaction.***

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
                    }
```

Vamos a ver ahora que es cada componente dentro de esta estructura:

- **sa_handler:** Especifica la acción que se va a asociar a la señal *signum.* Puede tomar tres valores diferentes:
    - SIG_DFL para la acción predeterminada
    - SIG_IGN para ignorar la señal
    - Puntero a función manejadora de la señal.
- **sa_mask:** permite establecer una máscara de señales que deberían bloquearse durante la ejecución del manejador de la señal. Además, la señal que lance el manejador será bloqueada, a menos que se activen las opciones SA_NODEFER o SA_NOMASK. Para asignar valores a sa_mask, tenemos las siguientes funciones:
    - int **sigemptyset**(sigset_t *set) → conjunto vacío de señales
    - int **sigfillset**(sigset_t *set) → conjunto con todas las señales
    - int **sigismember**(const sigset_t *set, int senyal) → determina si una señal pertenece al grupo
    - int **sigaddset**(sigset_t *set, int signo) → añade una señal al conjunto
    - int **sigdelset**(sigset_t *set, int signo) → elimina una señal del conjunto
- **sa_flags:** especifica un conjunto de opciones que modifican el comportamiento del proceso de manejo de señales. Se forma por la aplicación del operador de bits OR a cero o más de las siguientes constantes:
    - SA_NOCLDSTOP
    - SA_ONESHOT o SA_RESETHAND
    - SA_RESTART
    - SA_NOMASK o SA_NODEFER
    - SA_SIGINFO

-La llamada a SIGPROCMASK

La llamada al sistema sigprocmask se emplea para examinar y cambiar la máscara de señales.

*int **sigprocmask**(int how, const sigset_t *set, sigset_t *oldset)*

El argumento how nos indica el tipo de cambio que vamos a realizar, pudiendo ser:

- SIG_BLOCK: Las señales bloqueadas son la unión del conjunto actual y el argumento set.
- SIG_UNBLOCK: Las señales que hay en set se eliminan de señales bloqueadas.
- SIG_SETMASK: El conjunto de señales bloqueadas se pone según el set.

El argumento set representa un puntero al nuevo conjunto de señales enmascaradas. Si set es diferente de NULL, apunta a un conjunto de señales, si no, sigprocmask solo sirve para consultar.

Por último, el argumento oldset presenta el anterior conjunto de señales enmascaradas. Si oldset no es NULL, el valor anterior de la máscara de señal se guarda en oldset. EN caso contrario, devolverá la máscara anterior.

La llamada al sistema sigprocmask devolverá 0 en caso de éxito, sino -1.

-La llamada SIGPENDING

La llamada **sigpending** permite examinar el conjunto de señales bloqueadas y/o pendientes de entrega. La máscara de señales pendientes se guardará en el set.

*int **sigpending**(sigset_t *set);*

El argumento set es un puntero al conjunto de señales pendientes

La llamada a sigpending devolverá 0 en caso de éxito y -1 en caso de error

-La llamada SIGSUSPEND

La llamada **sigsuspend** reemplaza temporalmente la máscara de señal para el proceso con la dada por el argumento **mask** y luego suspende el proceso hasta que recibe una señal.

*int **sigsuspend**(const sigset_t *mask);*

El argumento más representa el puntero al nuevo conjunto de señales enmascaradas.

Devuelve 0 en caso de éxito y -1 en caso de error.

Notas importantes:

- No es posible bloquear SIGKILL, ni SIGSTOP con llamada a sigprocmask.

Sesión 6: Control de archivos y archivos proyectados en memoria