Prácticas de
Sistemas Operativos II

Dpto. Lenguajes y Sistemas Informáticos
E.T.S.I. Informática
Universidad de Granada

Objetivos

El primer objetivo de este guión de prácticas ---y de las prácticas de la asignatura--- es familiarizarse con la programación de sistemas utilizando los servicios del sistema operativo (llamadas al sistema). El lenguaje de programación utilizado en las prácticas es el C ya que es el que tiene más amplia difusión en la programación sobre el SO Linux, que es el que vamos a utilizar como soporte para las prácticas.

Las llamadas al sistema utilizadas siguen el estándar POSIX 1003.1 para interface del sistema operativo. Este estándar define los servicios que debe proporcionar un sistema operativo si va a "venderse" como conforme a POSIX ("POSIX compliant").

El segundo objetivo es que podáis observar cómo los conceptos explicados en teoría se reflejan en una implementación de sistema operativo como es el Linux y podáis acceder a las estructuras de datos que almacenan toda la información relativa a los distintos conceptos (archivo, proceso, etc..) explicados.

Como tercer objetivo parece lógico pensar que una vez aprendido un shell del sistema operativo UNIX (lo habéis practicado en SOI) entendáis que muchas de las órdenes de un shell se implementan mediante el uso de las llamadas al sistema y podáis ver determinadas operaciones a un nivel de abstracción más bajo.

¿Qué documentación necesitamos?

Para enfrentarnos a la programación utilizando llamadas al sistema en un entorno UNIX es conveniente disponer de la siguiente documentación:

- Para utilizar la biblioteca libc o glibc (que contiene: las llamadas al sistema, la biblioteca de matemáticas y las hebras POSIX) podemos consultar la siguiente documentación: l**ibc.info** o **libc.html (glibc.html)**.
- Manual en línea del sistema: **$> man man**

A continuación se presenta una "guía de supervivencia" básica para empezar a manejarse con el manual de un sistema UNIX (Linux) :

man es el paginador del manual del sistema. Las páginas usadas como argumentos al ejecutar man suelen ser normalmente nombres de programas, utilidades o funciones. La página de manual asociada con cada uno de esos argumentos es buscada y presentada. Si la llamada da también la *sección*, man buscará sólo en dicha sección del manual. Normalmente, la búsqueda se lleva a cabo en todas las secciones de manual disponibles según un orden predeterminado, y sólo se presenta la primera página encontrada, incluso si esa página se encuentra en varias secciones.

La siguiente tabla muestra los números de sección del manual y los tipos de páginas que contienen.

1 Programas ejecutables y guiones del intérprete de órdenes
2 Llamadas del sistema (funciones servidas por el núcleo)
3 Llamadas de la biblioteca (funciones contenidas en las bibliotecas del sistema)
4 Ficheros especiales (se encuentran generalmente en /dev)
5 Formato de ficheros y convenios p.ej. /etc/passwd
7 Paquetes de macros y convenios p.ej. man(7), groff(7).
8 Órdenes de admistración del sistema (generalmente solo son para usuario ***root***)

Una página de manual tiene varias partes. Éstas están etiquetadas como NOMBRE, SINOPSIS, DESCRIPCIÓN, OPCIONES, FICHEROS, VÉASE TAMBIÉN, BUGS, y AUTOR.

En la sección SINOPSIS se siguen los siguientes convenios que pueden servir de guía para otras secciones.

texto en negrita debes teclear esto exactamente.
texto en cursiva reemplace esto por el argumento apropiado.
[-abc] uno o todos los argumentos entre corchetes son opcionales.
-a | -b las opciones separadas por | no pueden usarse conjuntamente.
argumento ... argumento es repetible.
[expresión] ... la expresión entre corchetes completa es repetible.

Las formas más comunes de usar man son las siguientes:

man sección elemento Presenta la página de elemento disponible en la sección del manual.
man -a elemento Presenta, secuencialmente, todas las páginas de elemento disponibles en el manual. Entre página y página se puede decidir saltar a la siguiente o salir del paginador completamente.
man -k palabra-clave Busca la palabra-clave entre las descripciones breves y las páginas de manual y presenta todas las que casen.

Parte 1

Llamadas al sistema para el Sistema de Archivos

1. Entrada/Salida de archivos regulares

La mayor parte de las entradas/salidas (E/S) en UNIX pueden realizarse utilizando solamente cinco llamadas: `open`, `read`, `write`, `lseek` y `close`. Las funciones descritas en esta sección se conocen normalmente como entrada/salida sin búfer (unbuffered I/O). La expresión "sin búfer" se refiere al hecho de que cada `read` o `write` invoca una llamada al sistema en el núcleo y no se almacena en un búfer de la biblioteca.

Para el núcleo, todos los archivos abiertos son identificados por medio de *descriptores de archivo*. Un descriptor de archivo es un entero no negativo. Cuando abrimos, `open`, un archivo que ya existe o creamos, `creat`, un nuevo archivo, el núcleo devuelve un descriptor de archivo al proceso. Cuando queremos leer o escribir de/en un archivo identificamos el archivo con el descriptor de archivo que fue devuelto por las llamadas anteriormente descritas.

Por convenio, los shell de UNIX asocian el descriptor de archivo 0 con la entrada estándar de un proceso, el descriptor de archivo 1 con la salida estándar, y el descriptor 2 con el error estándar. Para realizar un programa conforme al estándar POSIX 2..10 ("POSIX 2.10 compliant") debemos utilizar las siguientes constantes simbólicas para referirnos a estos tres descriptores de archivos: `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`, definidas en `<unistd.h>`.

Cada archivo abierto tiene una *posición de lectura/escritura actual* ("***current file offset***"). Está representado por un entero no negativo que mide el número de bytes desde el comienzo del archivo. Las operaciones de lectura y escritura comienzan normalmente en la posición actual y provocan un incremento en dicha posición, igual al número de bytes leídos o escritos. Por defecto, esta posición es inicializada a 0 cuando se abre un archivo, a menos que se especifique al opción `O_APPEND`. La posición actual (`current_offset`) de un archivo abierto puede cambiarse explícitamente utilizando la llamada al sistema `lseek`.

Actividad 1. Trabajo con llamadas de gestión y procesamiento sobre archivos regulares.

Consultar la llamada al sistema `open` en el manual en línea. Fíjate en el hecho de que puede usarse para abrir un archivo ya existente o para crear un nuevo archivo. En el caso de la creación de un nuevo archivo tienes que entender correctamente la relación entre la máscara `umask` y el campo `mode`, que permite establecer los permisos del archivo. El argumento `mode` especifica los permisos a emplear si se crea un nuevo archivo. Es modificado por la máscara `umask` del proceso de la forma habitual: los permisos del fichero creado son (`modo & ~umask`).

Mirar la llamada al sistema `close` en el manual en línea.

Mirar la llamada al sistema `lseek` fijándote en las posibilidades de especificación del nuevo `current _offset`.

Mirar la llamada al sistema `read` fijándote en el número de bytes que devuelve a la hora de leer desde un archivo y los posibles casos límite.

Mirar la llamada al sistema `write` fijándote en que devuelve los bytes que ha escrito en el archivo.

1. ¿Qué hace el siguiente programa? Probar tras la ejecución del programa las siguientes órdenes del shell: $>cat archivo y $> od -c archivo

```
/*
tarea1.c
Trabajo con llamadas al sistema del Sistema de Archivos "POSIX 2.10 compliant"
Probar tras la ejecución del programa: $>cat archivo y $> od -c archivo
*/

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

char buf1[]="abcdefghij";
char buf2[]="ABCDEFGHIJ";

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

if( (fd=open("archivo",O_CREAT|O_WRONLY,S_IRUSR|S_IWUSR))<0) {
		printf("\nError %d en open",errno);
		perror("\nError en open");
		exit(-1);
}
if(write(fd,buf1,10) != 10) {
		perror("\nError en primer write");
		exit(-1);
}

if(lseek(fd,40,SEEK_SET) < 0) {
		perror("\nError en lseek");
		exit(-1);
}

if(write(fd,buf2,10) != 10) {
		perror("\nError en segundo write");
		exit(-1);
}

return 0;
}
```

2. Implementar un programa que acepte como argumento un "pathname", abra el archivo correspondiente y utilizando un tamaño de partición de los bytes del archivo igual a 80 Bytes cree un archivo de salida en el que debe aparecer lo siguiente:

```
Bloque1
//los primeros 80 Bytes
Bloque2
//los siguientes 80 Bytes
...
```

Si no se pasa un argumento al programa se debe utilizar la entrada estándar como archivo de entrada.

Modificación adicional. ¿Cómo tendrías que modificar el programa para que una vez finalizada la escritura en el archivo de salida y antes de cerrarlo, pudiésemos indicar en su primera línea el número de etiquetas "bloque i" escritas de forma que tuviese la siguiente apariencia:

```
El número de bloques es <nº_bloques>
Bloque1
<los primeros 80 Bytes>
```

Bloque2
<los siguientes 80 Bytes>
...

2. Archivos y directorios

En la sección anterior, hemos trabajado con llamadas al sistema básicas sobre archivos regulares. Ahora nos centraremos en características adicionales del sistemas de archivos y en las propiedades de un archivo (los metadatos o atributos). Comenzaremos con las funciones de la familia de `stat` y veremos cada uno de los campos de la estructura `stat`, que contiene los atributos de un archivo. A continuación veremos algunas de las llamadas al sistema que permiten modificar dichos atributos. Finalmente trabajaremos con funciones que operan sobre directorios.

La estructura `stat` 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 de bloque de 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 */
};
```

El valor `st_blocks` da el tamaño del fichero en bloques de 512 bytes. El valor `st_blksize` da el tamaño de bloque "preferido" para operaciones de E/S eficientes sobre el sistema de ficheros. (Escribir en un fichero en porciones más pequeñas puede producir una secuencia leer-modificar-reescribir ineficiente). Este tamaño de bloque preferido coincide con el tamaño de bloque de formateo del Sistema de Archivos donde reside.

No todos los sistemas de archivos en Linux implementan todos los campos de hora. Por lo general, `st_atime` es modificado por `mknod(2)`, `utime(2)`, `read(2)`, `write(2)` y `truncate(2)`.

Por lo general, `st_mtime` es modificado por `mknod(2)`, `utime(2)` y `write(2)`. `st_mtime` no se cambia por modificaciones en el propietario, grupo, cuenta de enlaces físicos o modo.

Por lo general, `st_ctime` es modificado al escribir o al poner información del inodo (p.ej.,propietario, grupo, cuenta de enlaces, modo, etc.).

Se definen las siguientes macros POSIX para comprobar el tipo de fichero:

```
S_ISLNK(st_mode)    es un enlace simbólico (soft)?
S_ISREG(st_mode)    un archivo regular?
S_ISDIR(st_mode)    un directorio?
S_ISCHR(st_mode)    un dispositivo de caracteres?
S_ISBLK(st_mode)    un dispositivo de bloques?
S_ISFIFO(st_mode)   una cauce con nombre (FIFO)?
S_ISSOCK(st_mode)   un socket?
```

Se definen las siguientes banderas (flags) para el campo `st_mode`:

S_IFMT	0017000	máscara de bits para los campos de bit del tipo de archivo (no POSIX)
S_IFSOCK	0140000	socket (no POSIX)
S_IFLNK	0120000	enlace simbólico (no POSIX)
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	00700	**user** (propietario del archivo) tiene permisos de lectura, escritura y ejecución
S_IRUSR	00400	user tiene permiso de lectura (igual que S_IREAD, no POSIX)
S_IWUSR	00200	user tiene permiso de escritura (igual que S_IWRITE, no POSIX)
S_IXUSR	00100	user tiene permiso de ejecución (igual que S_IEXEC, no POSIX)
S_IRWXG	00070	**group** tiene permisos de lectura, escritura y ejecución
S_IRGRP	00040	group tiene permiso de lectura
S_IWGRP	00020	group tiene permiso de escritura
S_IXGRP	00010	group tiene permiso de ejecución
S_IRWXO	00007	**other** tienen permisos de lectura, escritura y ejecución
S_IROTH	00004	other tienen permiso de lectura
S_IWOTH	00002	other tienen permiso de escritura
S_IXOTH	00001	other tienen permiso de ejecución

Tipos de archivos

- *Archivo regular*. Contiene datos de cualquier tipo. No existe distinción para el núcleo de UNIX con respecto al tipo de datos del fichero: binario o de texto. Cualquier interpretación de los contenidos de un archivo regular es responsabilidad de la aplicación que procesa dicho archivo.
- *Archivo de directorio*. Un directorio es un archivo que contiene los nombres de otros archivos (incluidos directorios) y punteros a la información de dichos archivos. Cualquier proceso que tenga permiso de lectura para un directorio puede leer los contenidos de un directorio, pero solamente el núcleo puede escribir en un directorio, e.d. hay que crear y borrar archivos utilizando servicios del sistema operativo.
- *Archivo especial de dispositivo de caracteres*. Se usa para representar ciertos tipos de dispositivos en un sistema.
- *Archivo especial de dispositivo de bloques*. Se usa normalmente para representar discos duros, CDROM,... Todos los dispositivos de un sistema están representados por archivos especiales de caracteres o de bloques. (Probar: `$> cat /proc/devices; cat /proc/partitions`)
- *FIFO*. Un tipo de archivo utilizado para comunicación entre procesos (IPC). También llamado cauce con nombre.
- *Enlace simbólico*. Un tipo de archivo que apunta a otro archivo.
- *Socket*. Un tipo de archivo usado para comunicación en red entre procesos. También se puede usar para comunicar procesos en un único nodo (host).

Permisos de acceso a archivos

El valor `st_mode` codifica además del tipo de archivo los permisos de acceso al archivo, independientemente del tipo de archivo de que se trate. Disponemos de tres categorías: *user (owner), group y other* para establecer los permisos de lectura, escritura y ejecución. Los permisos de lectura, escritura y ejecución se utilizan de forma diferente según la llamada al sistema. A continuación describiremos las más relevantes:

- Cada vez que queremos abrir cualquier tipo de archivo ---usamos su `pathname` o el directorio actual o la variable de entorno $PATH--- tenemos que disponer de permiso de ejecución en cada directorio mencionado en el `pathname`. Por esto se suele llamar al bit de permiso de ejecución para directorios: *bit de búsqueda*.
  Hay que tener en cuenta que el permiso de lectura para un directorio y el permiso de ejecución significan cosas diferentes. El permiso de lectura nos permite leer el directorio, obteniendo una lista de todos los nombres de archivo del directorio. El permiso de ejecución nos permite pasar a través del directorio cuando es un componente de un `pathname` al que estamos tratando de acceder.
- 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`.
- Debemos tener permiso de escritura en un archivo para poder especificar el flag `O_TRUNC` en la llamada `open`.
- No podemos crear un nuevo archivo en un directorio a menos que tengamos permisos de escritura y ejecución en dicho directorio.
- Para borrar un archivo existente necesitamos permisos de escritura y ejecución en el directorio que contiene el archivo. No necesitamos permisos de lectura o escritura en el archivo.
- El permiso de ejecución para un archivo debe estar activado si queremos ejecutar el archivo usando cualquier función de la familia `exec` o si es un *script* de un shell. Además el archivo debe ser regular.

Actividad 2. Trabajo con llamadas al sistema de la familia stat.

Mirar las llamadas al sistema `stat` y `lstat` y entender sus diferencias.

1. ¿Qué hace el siguiente programa?

```
/*
tarea2.c
Trabajo con llamadas al sistema del Sistema de Archivos "POSIX 2.10 compliant"
*/

#include<sys/types.h>
#include<unistd.h>
#include<sys/stat.h>
#include<stdio.h>
#include<errno.h>
#include<string.h>

int main(int argc, char *argv[])
{
int i;
struct stat atributos;
char tipoArchivo[30];

if(argc<2) {
		printf("\nSintaxis de ejecucion: tarea2 [<nombre_archivo>]+\n\n");
		exit(-1);
}
for(i=1;i<argc;i++) {
   printf("%s: ", argv[i]);
   if(lstat(argv[i],&atributos) < 0) {
      printf("\nError al intentar acceder a los atributos de %s",argv[i]);
      perror("\nError en lstat");
```

```
        }
        else {
            if(S_ISREG(atributos.st_mode)) strcpy(tipoArchivo,"Regular");
            else if(S_ISDIR(atributos.st_mode)) strcpy(tipoArchivo,"Directorio");
            else if(S_ISCHR(atributos.st_mode)) strcpy(tipoArchivo,"Especial de caracteres");
            else if(S_ISBLK(atributos.st_mode)) strcpy(tipoArchivo,"Especial de bloques");
            else if(S_ISFIFO(atributos.st_mode)) strcpy(tipoArchivo,"Cauce con nombre (FIFO)");
            else if(S_ISLNK(atributos.st_mode)) strcpy(tipoArchivo,"Enlace relativo (soft)");
            else if(S_ISSOCK(atributos.st_mode)) strcpy(tipoArchivo,"Socket");
            else strcpy(tipoArchivo,"Tipo de archivo desconocido");

            printf("%s\n",tipoArchivo);
        }
    }

    return 0;
    }
```

2. Define una macro en lenguaje C que implemente la macro `S_ISREG(mode)` usando para ello los flags definidos en `<sys/stat.h>` para el campo `st_mode` de la `struct stat`. y comprueba que funciona en un programa simple.

```
#define S_ISREG2(mode) ...
```

La llamada al sistema `umask`.

Hasta el momento hemos descrito los nueve bits de permisos asociados con dada archivo. Ahora podemos mostrar la máscara de creación de permisos para archivos que está asociada con cada proceso.

La llamada `umask` fija la máscara de creación de permisos para el proceso y devuelve el valor previamente establecido. El argumento de la llamada está formado por una combinación OR de las nueve constantes de permisos (`rwx` para ugo) vistas anteriormente. A continuación se muestra un resumen de la página del manual del programador de Linux para esta llamada:

NOMBRE

umask - establece la máscara de creación de ficheros

SYNOPSIS

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

mode_t umask(mode_t mask);
```

DESCRIPCIÓN

`umask` establece la máscara de usuario a `mask & 0777`.

La máscara de usuario es usada por `open(2)` para establecer los permisos iniciales de un fichero recién creado. Específicamente, los permisos presentes en la máscara se desactivan del argumento `mode` de `open` (así pues, por ejemplo, el valor común por defecto de `umask`, `022`, provoca que los nuevos ficheros se creen con permisos `0666 & ~022 = 0644 = rw-r--r--` cuando `mode` vale `0666`, que es el caso más normal).

VALOR DEVUELTO

Esta llamada al sistema siempre tiene éxito y devuelve el valor anterior de la máscara.

Las llamadas 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` sobre un archivo especificado por su `pathname` mientras que la función `fchmod` opera sobre un archvo que ha sido previamente abierto con `open`.

A continuación se muestra un resumen de la página del manual del programador de Linux para esta llamada:

NOMBRE

chmod, fchmod - cambia los permisos de un archivo

SINOPSIS

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

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

DESCRIPCIÓN

Cambia el modo del fichero dado mediante `path` o referido por `fildes`. Los modos se especifican mediante un OR lógico de los siguientes valores:

S_ISUID 04000activar la asignación del UID del propietario al UID efectivo del proceso que ejecute el archivo.
S_ISGID 02000activar la asignación del GID del propietario al GID efectivo del proceso que ejecute el archivo.
S_ISVTX 01000 activar sticky bit

S_IRWXU 00700 **user** (propietario del archivo) tiene permisos de lectura, escritura y ejecución
S_IRUSR 00400lectura para el propietario (= S_IREAD no POSIX)
S_IWUSR 00200 escritura para el propietario (= S_IWRITE no POSIX)
S_IXUSR 00100 ejecución/búsqueda para el propietario (=S_IEXEC no POSIX)

S_IRWXG 00070 **group** tiene permisos de lectura, escritura y ejecución
S_IRGRP 00040 lectura para el grupo
S_IWGRP 00020 escritura para el grupo
S_IXGRP 00010 ejecución/búsqueda para el grupo

S_IRWXO 00007 **other** tienen permisos de lectura, escritura y ejecución
S_IROTH 00004 lectura para otros
S_IWOTH 00002 escritura para otros
S_IXOTH 00001 ejecución/búsqueda para otros

VALOR DEVUELTO

En caso de éxito, devuelve 0. En caso de error, -1 y se asigna a la variable **errno** un valor adecuado.

> Avanzado
>
> Para cambiar los bits de permisos de un archivo, el **UID efectivo** del proceso debe ser igual al del propietario del archivo, o el proceso debe tener permisos de superusuario (UID efectivo del proceso debe ser 0).

Si el UID efectivo del proceso no es cero y el grupo del fichero no coincide con el ID de grupo efectivo del proceso o con uno de sus ***ID's de grupo suplementarios***, el bit S_ISGID se desactivará, aunque esto no provocará que se devuelva un error.

Dependiendo del sistema de archivos, los bits S_ISUID y S_ISGID podrían desactivarse si el archivo es escrito. En algunos sistemas de archivos, solo el superusuario puede asignar el 'sticky bit', lo cual puede tener un significado especial (por ejemplo, para directorios, un archivo sólo puede ser borrado por el propietario o el superusuario).

Actividad 3. Trabajo con llamadas al sistema de cambio de permisos.

Mirar las llamadas al sistema `umask` y `chmod`.

1. ¿Qué hace el siguiente programa?

```
/*
tarea3.c
Trabajo con llamadas al sistema del Sistema de Archivos "POSIX 2.10 compliant"
Este programa fuente está pensado para que se cree primero un programa con la parte
 de CREACION DE ARCHIVOS y se haga un ls -l para fijarnos en los permisos y entender
 la llamada umask.
En segundo lugar (una vez creados los archivos) hay que crear un segundo programa
 con la parte de CAMBIO DE PERMISOS para comprender el cambio de permisos relativo
 a los permisos que actualmente tiene un archivo frente a un establecimiento de
 permisos absoluto.
*/

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


int main(int argc, char *argv[])
{
int fd1,fd2;
struct stat atributos;

//CREACION DE ARCHIVOS
if( (fd1=open("archivo1",O_CREAT|O_TRUNC|O_WRONLY,S_IRGRP|S_IWGRP|S_IXGRP))<0) {
		printf("\nError %d en open(archivo1,...)",errno);
		perror("\nError en open");
		exit(-1);
}

umask(0);
if( (fd2=open("archivo2",O_CREAT|O_TRUNC|O_WRONLY,S_IRGRP|S_IWGRP|S_IXGRP))<0) {
		printf("\nError %d en open(archivo2,...)",errno);
		perror("\nError en open");
		exit(-1);
}
```

```
//CAMBIO DE PERMISOS
if(stat("archivo1",&atributos) < 0) {
		printf("\nError al intentar acceder a los atributos de archivo1");
		perror("\nError en lstat");
		exit(-1);
}
if(chmod("archivo1", (atributos.st_mode & ~S_IXGRP) | S_ISGID) < 0) {
		perror("\nError en chmod para archivo1");
		exit(-1);
}
if(chmod("archivo2",S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH) < 0) {
		perror("\nError en chmod para archivo2");
		exit(-1);
}

return 0;
}
```

Leyendo directorios.

Aunque los directorios se pueden leer utilizando las mismas llamadas al sistema que para los archivos normales, como la estructura de los directorios puede cambiar de un sistema a otro, los programas en este caso no serían transportables. Para solucionar este problema, se va a utilizar una biblioteca estándar de funciones de manejo de directorios que se presentan de forma resumida a continuación:

- **opendir**: se le pasa el nombre del directorio a abrir, y devuelve un puntero a la estructura de tipo *DIR,* llamada *stream* de directorio. El tipo *DIR* está definido en `<dirent.h>`.
- **readdir**: lee la entrada donde esté situado el puntero de lectura de un directorio ya abierto cuyo *stream* se pasa a la función. Después de la lectura adelanta el puntero una posición. Devuelve la entrada leída a través de un puntero a una estructura (`struct dirent`), o devuelve NULL si llega al final del directorio o se produce un error.
- **closedir**: cierra un directorio, devolviendo **0** si tiene éxito, en caso contrario devuelve **-1**.
- **seekdir**: permite situar el puntero de lectura de un directorio.
- **telldir**: devuelve la posición del puntero de lectura de un directorio.
- **rewinddir**: posiciona el puntero de lectura al principio del directorio.

A continuación se dan las declaraciones de estas funciones y de las estructuras que se utilizan, contenidas en los archivos `<sys/types.h>` y `<dirent.h>` y del tipo *DIR* y la estructura *dirent* (entrada de directorio).

```
DIR *opendir(char *dirname)
struct dirent *readdir(DIR *dirp)
int closedir(DIR *dirp)
void seekdir(DIR *dirp, log loc)
long telldir(DIR *dirp)
void rewinddir(DIR *dirp)


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;

struc dirent {
        ino_t d_ino;                            /* inodo asociado a la entrada de directorio */
        short d_reclen;                 /* longitud de esta entrada */
        short d_namlen;                 /* longitud de la cadena que hay en d_name */
        char  d_name[_POSIX_PATH_MAX];          /* nombre del archivo */;
};

//La estructura struct dirent conforme a POSIX 2.1 es la siguiente:
#include <sys/types.h>
#include <dirent.h>

struct dirent {
        long      d_ino;                        /* número i-nodo */
        off_t     d_off;                        /* desplazamiento al siguiente dirent */
        unsigned short  d_reclen;       /* longitud de esta entrada */
        unsigned char   d_type;         /* tipo de archivo*/
        char   d_name[256];             /* nombre del archivo */
};
```

Todas estas funciones, en caso de error, devuelven en la variable `errno` el código de error producido, el cual se puede imprimir con la ayuda de la función `perror`. Esta función devuelve un literal descriptivo de la circunstancia concreta que ha originado el error (asociado a la variable `errno`). Además, permite que le pasemos un argumento que será mostrado en pantalla junto con dicho literal, lo cual nos ayuda a personalizar el tratamiento de errores. En el archivo `<errno.h>` se encuentra una lista completa de todas las circunstancias de error contempladas por todas las llamadas al sistema.

Actividad 4. Trabajo con funciones estándar de manejo de directorios.

Mirar las funciones estándar de trabajo con directorios utilizando `man opendir` y viendo el resto de funciones que aparecen en la sección VEASE TAMBIEN de esta página del manual.

1. Realizar un programa en C utilizando las llamadas al sistema necesarias que acepte como entrada:

- Un argumento que representa el ***'pathname'*** de un directorio.
- Otro argumento que es un ***número octal de 4 dígitos*** (similar al que se puede utilizar para cambiar los permisos en la llamada al sistema `chmod`)

El programa debe proporcionar en la salida estándar una línea para cada archivo del directorio que esté formada por:

```
<nombre_de_archivo> : <permisos_antiguos> <permisos_nuevos>
```

Si no se pueden cambiar los permisos de un determinado archivo se debe especificar la siguiente información en la línea de salida:

```
<nombre_de_archivo> : <errno> <permisos_antiguos>
```

2. Programe una nueva orden que recorra la jerarquía de subdirectorios existentes a partir de uno dado como argumento y devuelva la cuenta de todos aquellos archivos regulares que tengan permiso de ejecución para

el *grupo* y para *otros*. Además del nombre de los archivos encontrados, deberá devolver sus números de i-nodo y la suma total de espacio ocupado por dichos archivos. El formato de la nueva orden será:

```
ws1% buscar <pathname>
```

donde `<pathname>` especifica el nombre del directorio a partir del cual queremos que empiece a analizar la estructura del árbol de subdirectorios. En caso de que no se le de argumento, tomará como punto de partida el *directorio actual*. Ejemplo de la salida después de ejecutar el programa:

Los i-nodos son:

./a.out 55

./bin/ej 123

./bin/ej2 87

...

Existen 24 archivos regulares con permiso x para grupo y otros

El tamaño total ocupado por dichos archivos es 2345674 bytes

Parte 2

Llamadas al sistema para el Subsistema de Procesos

1. Comunicación entre procesos utilizando cauces

Los mecanismos de I***nter-Process Comunication*** que veremos en esta sección serán los cauces, con y sin nombre. Un ***cauce*** es un mecanismo para la comunicación entre procesos. Los datos que un proceso escribe en el cauce pueden ser leídos por otro proceso. Estos datos se tratan en orden First In First Out (FIFO).

1.1 Cauces con nombre (FIFO)

Conceptos básicos

Comúnmente usamos un cauce como un método de conexión que une la **Salida Estándar** de un proceso a la **Entrada Estándar** de otro. Los cauces proporcionan un método de comunicación entre procesos en un sólo sentido (unidireccional, semi-dúplex).

Este método se usa bastante en la línea de órdenes de los shell de UNIX:

```
ls | sort | lp
```

El anterior es un ejemplo claro de "*pipeline*", donde se toma la salida de una orden ls como entrada de una orden sort, la cual a su vez entrega su salida a una orden lp. Los datos fluyen por la cauce (semi-dúplex), viajando de izquierda a derecha.

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

- Los cauces con nombre existen en el sistema de archivos como un archivo especial.
- Los procesos abren un archivo FIFO , `open`, usando su nombre, con el fin de comunicarse a través de él.
- Los procesos de diferentes padres pueden compartir datos mediante una cauce con nombre.
- El archivo FIFO permanece en el sistema de archivos una vez realizadas todas las E/S de los procesos que lo han utilizado como mecanismo de comunicación, hasta que se borre como cualquier archivo: `unlink`.

Creación de un archivo FIFO

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

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

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

La función `'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 que dispositivo se refiere el archivo especial. Su interpretación depende de la clase de archivo especial que se vaya a crear. Para crear una cauce FIFO el valor de este argumento será 0.

Un ejemplo de creación de una cauce FIFO sería el siguiente:

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

En este caso el archivo "/tmp/FIFO" se crea como archivo FIFO y los permisos solicitados son "0666". Los permisos que el sistema finalmente asigna al archivo son el resultado de la siguiente expresión, como ya vimos en la parte 1 del guión:

```
umaskFinal = permisosSolicitados & ~umaskInicial
```

donde 'umaskInicial' es la máscara de permisos que almacena el sistema en el u-area del proceso, con el objetivo de asignar permisos a los archivos de nueva creación.

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

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

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

Utilización de un cauce FIFO

Las operaciones de E/S sobre un archivo FIFO son esencialmente las mismas que las utilizadas con los archivos regulares salvo una diferencia: en el archivo FIFO no podemos hacer lseek, ya que la filosofía de trabajo es la de primero en entrar, primero en salir. Por tanto no tiene sentido mover el offset a una posición dentro del flujo de datos.

Actividad 1. Trabajo con cauces con nombre (FIFOS).

Consulta en el manual en línea las llamadas al sistema para la creación de archivos especiales en general, mknod, y la específica para archivos FIFO, mkfifo.

1. A continuación se especifica el código de dos programas que modelizan el problema del productor-consumidor para dos procesos, usando como mecanismo de comunicación un cauce FIFO. El programa servidor debe ejecutarse en primer lugar (en background), ya que se encarga de crear el archivo FIFO. Si no se ejecutan en este orden, el programa cliente intentará abrir un archivo que no existe y se producirá un error.

```
//consumidorFIFO.c
//Consumidor que usa mecanismo de comunicacin FIFO.
//Ejecutar el programa: $> consumidorFIFO & (en background)
//Despu   ejecutar el programa productorFIFO
```

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


#define ARCHIVO_FIFO "ComunicacionFIFO"

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

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

//Abro el cauce para lectura
if ( (fd=open(ARCHIVO_FIFO,O_RDWR)) <0)  { //O_RDONLY
		perror("open");
		exit(-1);
}
//Aceptar datos a consumir hasta que se envíe la cadena fin
while(1) {
		leidos=read(fd,buffer,80);
		if(strcmp(buffer,"fin")==0) {
		  close(fd);
		    return 0;
		}
		printf("\nMensaje recibido: %s\n", buffer);

}

return 0;

}


/* ======================= * ======================== */
 Y el código de cualquier proceso productor quedaría de la siguiente forma:
/* ======================= * ======================== */


//productorFIFO.c
//Productor que usa mecanismo de comunicacin FIFO.

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

#define ARCHIVO_FIFO "ComunicacionFIFO"
```

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

//Comprobación de 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;

}
```

1.2 Cauces sin nombre (pipes)

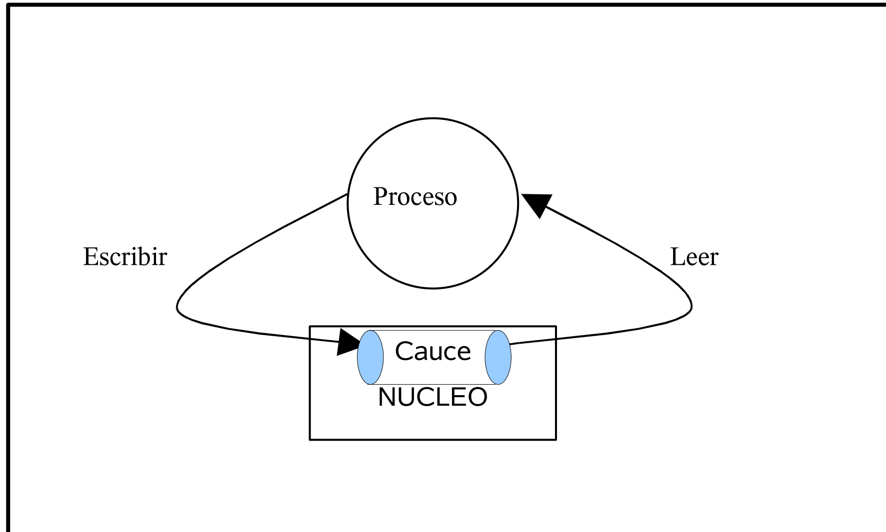
Conceptos básicos

Los cauces sin nombre mantienen el mismo orden en el flujo de datos (First In First Out) que los cauces son nombre, pero presentan las siguientes diferencias:

- Los cauces sin nombre no tienen un archivo asociado en el sistema de archivos en disco.
- Al crear un cauce sin nombre utilizando la llamada al sistema correspondiente, automáticamente se nos 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 el proceso que crea el cauce sin nombre y todos sus descendientes.
- El cauce sin nombre es destruido por el núcleo cuando sus contadores asociados de número de lectores y número de escritores valen 0.

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

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

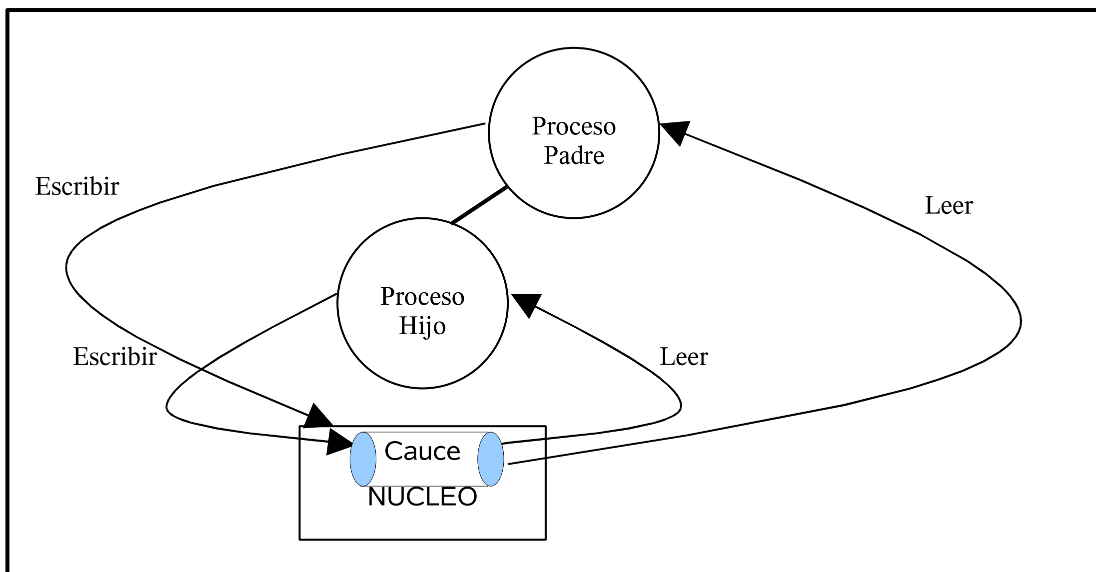


En el esquema anterior podemos ver como el proceso puede usar los descriptores de archivo para enviar datos, `write`, a la cauce y leerlos, `read`, desde ésta. Sin embargo, este esquema que carece de utilidad práctica, se puede ampliar.

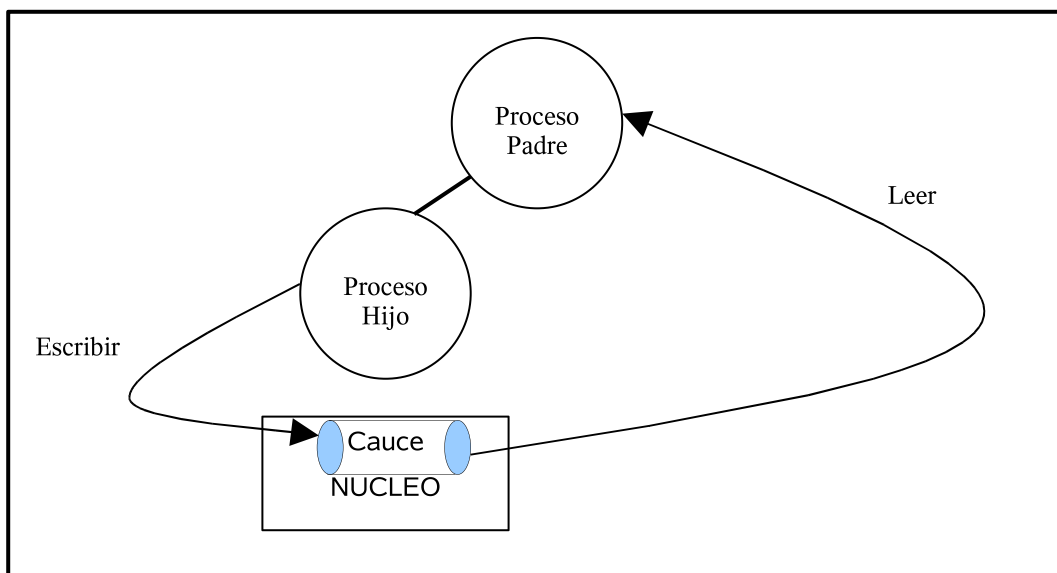
Mientras un cauce conecta inicialmente un proceso a sí mismo, los datos que viajan por él se mueven a nivel de núcleo. Bajo Linux en particular, los cauces se representan internamente por medio de un i-nodo válido (entrada en la tabla de i-nodos). Por supuesto, este i-nodo reside dentro del núcleo mismo, y no dentro de los límites de cualquier sistema de archivos físico.

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

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



esquema quedaría como sigue:



Para realizar operaciones sobre un cauce sin nombre podemos usar las mismas llamadas al sistema que se usan para un archivo de E/S de bajo nivel (ya que las cauces están representadas internamente como un i-nodo válido). Para enviar datos a la cauce usamos la llamada al sistema `write`, y para recibir los datos de la cauce usamos la llamada al sistema `read`.

Creación de cauces sin nombre en C

Para crear un cauce en C, usamos la llamada al sistema `pipe`, la cuál toma como argumento un vector de dos enteros, `int fd[2]`,y si la llamada tiene éxito, el vector contendrá dos nuevos descriptores de archivo que permitirán usar el nuevo cauce.

Por defecto, la llamada crea el primer elemento del vector (*fd[0]*) como un descriptor de archivo para sólo *lectura*, mientras que el segundo elemento (*fd[1]*) está fijado para *escritura*. Una vez creado el cauce, creamos un proceso hijo (que heredará los descriptores de archivos del padre) y establecemos el sentido del flujo de datos (hijo->padre, padre->hijo)[1].

Como los descriptores son compartidos por el proceso padre y el hijo, debemos estar seguros siempre de cerrar el extremo del cauce que no nos interese, para evitar confusiones que podrían derivar en errores al usar el mecanismo.

Actividad 2. Trabajo con cauces sin nombre (pipes).

Consulta en el manual en línea la llamada al sistema para la creación de cauces sin nombre, `pipe`.

1. ¿Qué hace el siguiente programa?

```
/*
tarea4.c
Trabajo con llamadas al sistema del Subsistema de Procesos conforme a POSIX 2.10
Programa ilustrativo del uso de pipes.
*/
```

[1] Si el padre quiere recibir datos del hijo, debe cerrar el descriptor usado para escritura (fd[1]) y el hijo debe cerrar el descriptor usado para lectura (fd[0]). Si 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]).

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

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

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

 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.
   //Cierro 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);
}
```

Ahora, podemos hacer una modificación muy interesante de cara a redireccionar la entrada o salida estándar de cualquiera de los procesos[2].

Para conseguir redireccionar la entrada o salida estándar al descriptor de lectura o escritura del cauce hacemos uso de una llamada al sistema, `dup`, que se encarga de duplicar el descriptor indicado como parámetro de entrada en la entrada libre con número de descriptor más bajo de la tabla de descriptores de archivo usada por el proceso.

De esta manera podemos escribir un programa en C que permita simular el funcionamiento de una orden típica del shell: `ls | sort`, construyendo para ello un cauce.

2. 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.

```
/*
tarea5.c
```

[2]El descriptor de archivo, 0, de cualquier proceso UNIX se redirecciona a la entrada estándar (stdin) que se asigna por defecto al teclado, y el descriptor de archivo, 1, se redirecciona a la salida estándar (stdout) asignada por defecto a la consola activa.

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

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


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

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

 if ( (PID= fork())<0)  {
   perror("fork");
   exit(1);
 }

 if(PID == 0)  { // ls
                 //Establecer la dirección del flujo de datos en el cauce cerrando
                 // el descriptor de lectura de cauce en el proceso hijo
                 close(fd[0]);

                 //Redirigir la salida estándar para enviar datos al cauce.
                 //--------------------------------------------------------

                 //Cerrar la salida estándar del proceso hijo
                 close(STDOUT_FILENO);

   //Duplicar el descriptor de escritura en cauce en el descriptor
   //correspondiente a la salida estándar (stdout)
   dup(fd[1]);

   execlp("ls","ls",NULL);
 }
 else  { // sort. Estoy en el proceso padre porque PID != 0.
                 //Establecer la dirección del flujo de datos en el cauce cerrando
                 // el descriptor de escritura en el cauce del proceso padre.
   close(fd[1]);

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

   //Duplicar el descriptor de lectura de cauce en el descriptor
   //correspondiente a la entrada estándar (stdin).
   dup(fd[0]);

   execlp("sort","sort",NULL);
 }

 return(0);

}
```

Existe otra llamada al sistema, `dup2`, que se puede usar y que permite una *atomicidad* (evita posibles condiciones de competencia) 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: *cerrar descriptor antiguo y duplicar descriptor*. Se garantiza que la llamada es **atómica**, por lo que, si llega una señal al proceso, toda la operación transcurrirá antes de devolverle el control al núcleo para gestionar la señal.

3. 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. Es este caso se utiliza la llamada `dup2`.

```
/*
tarea6.c
Programa ilustrativo del uso de pipes y la redirección de entrada y
salida estándar: "ls | sort", utilizando la llamada dup2.
*/

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

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

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

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

  //Duplicar el descriptor de escritura en cauce en el descriptor
  //correspondiente a la salida estd   ar (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);

}
```

Notas finales sobre cauces con y sin nombre

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

2. Control de Procesos

El control de procesos en UNIX incluye las llamadas al sistema necesarias para implementar la funcionalidad de creación de nuevos procesos, ejecución de programas, terminación de procesos y alguna funcionalidad adicional como sería la sincronización básica entre un proceso padre y sus procesos hijo. Además, veremos los distintos identificadores de proceso que se utilizan en UNIX tanto para el usuario como para el grupo (reales y efectivos) y como se ven afectados por las primitivas de control de procesos.

2.1 Identificadores de proceso

Cada proceso tiene un único identificador de proceso (PID) que es un número entero no negativo. Existen algunos procesos especiales como son el `swapper` (PID =0) , el `init` (PID=1) y el `pagedaemon` (PID=2 en algunas implementaciones de UNIX). El proceso `swapper` y el `pagedaemon` son procesos de núcleo (sistema) y se encargan de realizar el intercambio y la paginación respectivamente.

El proceso `init` (proceso demonio o hebra núcleo) es el encargado de inicializar el sistema UNIX y ponerlo a disposición de los programas de aplicación, después de que se haya cargado el núcleo. El programa encargado de dicha labor suele ser el `/sbin/init` que normalmente lee los archivos de inicialización dependientes del sistema (que se encuentran en `/etc/rc*`)y lleva al sistema a cierto estado. Este proceso no finaliza hasta que se detiene al sistema operativo. Además, a diferencia de los procesos `swapper` y `pagedaemon` no es un proceso de sistema sino uno normal aunque se ejecute con privilegios de superusuario. El proceso `init` es *el proceso raíz de la jerarquía de procesos del sistema*, que se genera debido a las relaciones de entre proceso creador (padre) y proceso creado (hijo).

Además del identificador de proceso, existen los siguientes identificadores asociados al proceso y que se detallan a continuación, junto con las llamadas al sistema que los devuelven.

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

pid_t getpid(void);          // devuelve el identificador de proceso del proceso que la invoca.
```

```
pid_t getppid(void);        // devuelve el identificador de proceso del padre del proceso
                            //  que la invoca.
uid_t getuid(void);         // devuelve el identificador de usuario real del proceso invocador.
uid_t geteuid(void);        // devuelve el identificador de usuario efectivo del proceso
                            // invocador.
gid_t getgid(void);         // devuelve el identificador de grupo real del proceso invocador.
gid_t getegid(void);        // devuelve el identificador de grupo efectivo del proceso
                            // invocador.
```

El identificador de usuario real,`UID`, proviene de la comprobación que realiza el programa `logon` sobre cada usuario que intenta acceder al sistema proporcionando la pareja: *login, password*. El sistema utiliza este par de valores para identificar la línea del archivo de passwords (`/etc/passwd`) correspondiente al usuario y para comprobar la clave en el archivo de shadow passwords.

El UID efectivo (`euid`) se corresponde con el UID real salvo en el caso en el que el proceso ejecute un programa con el bit SUID activado, en cuyo caso se corresponderá con el UID del propietario del archivo ejecutable.

El significado del GID real y efectivo es similar al del UID real y efectivo pero para el caso del *grupo preferido* del usuario. El grupo preferido es el que aparece en la línea correspondiente al login del usuario en el archivo de passwords.

2.2 Llamada al sistema fork

La única forma de que el núcleo de UNIX cree un nuevo proceso es que un proceso, que ya exista, ejecute `fork` (exceptuando los procesos especiales, algunos de los cuales hemos comentado brevemente en la sección anterior).

El nuevo proceso que se crea tras la ejecución de la llamada `fork` se denomina *proceso hijo*. Esta llamada al sistema se ejecuta una sola vez, pero devuelve dos resultados dependiendo del lugar en el que se encuentre (padre o hijo). La única diferencia entre los valores devueltos es que en el proceso hijo el valor es 0 y en el proceso padre (el que ejecutó la llamada) 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. 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, con lo que el hijo siempre puede ejecutar `getppid` para obtener el PID de su padre.

Desde el punto de vista de la programación, el que dependiendo del padre o del hijo la llamada devuelva un valor distinto nos va a ser muy útil, de cara a poder ejecutar distintas partes de código una vez finalizada la llamada `fork`.

Tanto el padre como el hijo continuarán ejecutando la instrucción siguiente al `fork` y el hijo será una copia idéntica del padre. En general, nunca podemos saber si el hijo se ejecutará antes que el padre o viceversa. Esto dependerá del algoritmo de planificación de CPU que utilice el núcleo.

NOTA: Existía una llamada al sistema similar a `fork`, `vfork`, que tenía un significado un poco diferente a ésta. Tenía la misma secuencia de llamada y los mismos valores de retorno que `fork`, pero `vfork` estaba diseñada para crear un nuevo proceso cuando el propósito del nuevo proceso era ejecutar, `exec`, un nuevo programa. `vfork` creaba el nuevo proceso sin copiar el espacio de direcciones del padre en el hijo, ya que el hijo no iba a hacer referencia a dicho espacio de direcciones sino que realizaba un `exec`, y mientras esto ocurría el proceso padre permanecía bloqueado(estado `sleep` en UNIX).

2.3 Llamadas al sistema exit, wait y waitpid

Una de las formas de terminar la ejecución de un programa es realizando la llamada `exit`.

```
#include <stdlib.h>
```

```
void exit(int status);
```

La llamada `exit` produce la terminación normal del programa y la devolución de `status` al proceso padre. Esta variable se utiliza para incluir un valor que permita al proceso hijo indicar a su padre la causa de su terminación. El proceso padre puede obtener la información de estado utilizando las funciones `wait` o `waitpid`.

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

 pid_t wait(int *status)
 pid_t waitpid(pid_t pid, int *status, int options);
```

Cuando finaliza un proceso, el kernel avisa al proceso padre enviándole la señal `SIGCHLD`. Ya que la finalización de un hijo es un evento asíncrono a la ejecución del padre, esta señal es la notificación al padre de que se ha producido el evento asíncrono: ***finalización de hijo***.

Cuando un proceso llama a `wait` o `waitpid` pueden darse tres casos:

- El proceso se bloquea, si todos sus hijos se están ejecutando y ninguno ha terminado todavía.
- El proceso continua con el valor de estado de finalización de un hijo almacenado en `status` (si ha terminado al menos un hijo y está esperando para que el padre recoja su estado de terminación)
- El proceso continua pero `wait` o `waitpid` devuelven un error debido a que el proceso no tiene ningún hijo.

Las diferencias entre las dos llamadas son:

- `wait` puede bloquear al proceso que la ejecuta hasta que se produzca la finalización de un proceso hijo, mientras que waitpid tiene una opción que previene el bloqueo del proceso llamador.
- `waitpid` no espera la terminación del primer proceso que termine, sino que tiene opciones que controlan el proceso por el que espera.

Actividad 3. Trabajo con llamadas al sistema de control de procesos.

Consulta en el manual en línea las distintas llamadas al sistema para la obtención de los distintos identificadores que tiene un proceso: `getpid, getppid, getuid, geteuid, getgid, getegid`.

Consulta en el manual en línea las llamadas `wait` y `waitpid` para ver sus posibilidades de sincronización entre el proceso padre y su(s) proceso(s) hijo(s).

1. ¿Qué hace el siguiente programa? Intenta entender lo que ocurre con las variables y sobre todo con los mensajes por pantalla cuando el núcleo tiene activado/desactivado el mecanismo de buffering.

```
/*
tarea7.c
Trabajo con llamadas al sistema del Subsistema de Procesos "POSIX 2.10 compliant"
Prueba el programa tal y como está. Después, elimina los comentarios (1) y pruébalo de nuevo.
*/

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

int global=6;
```

```
char buf[]="cualquier mensaje de salida\n";

int main(int argc, char *argv[])
{
int var;
pid_t pid;

var=88;
if(write(STDOUT_FILENO,buf,sizeof(buf)+1) != sizeof(buf)+1) {
                perror("\nError en write");
                exit(-1);
}
//(1)if(setvbuf(stdout,NULL,_IONBF,0)) {
//              perror("\nError en setvbuf");
//}
printf("\nMensaje previo a la ejecución de fork");

if( (pid=fork())<0) {
                perror("\nError en el fork");
                exit(-1);
}
else if(pid==0) {  //proceso hijo ejecutando el programa
                 global++;
                     var++;
                } else  //proceso padre ejecutando el programa
                  sleep(1);

printf("\npid= %d, global= %d, var= %d\n", getpid(),global,var);
exit(0);

}
```

Nota 1: El núcleo no realiza buffering de salida con la llamada al sistema `write`. Esto quiere decir que cuando usamos `write(STDOUT_FILENO,buf,tama)`, los datos se escriben directamente en la salida estándar sin ser almacenados en un buffer temporal. Sin embargo, el núcleo sí realiza buffering de salida en las funciones de la biblioteca estándar de E/S del C, en la cual está incluida `printf`. Para deshabilitar el buffering en la biblioteca estándar de E/S se utiliza la siguiente función:

```
int setvbuf(FILE *stream, char *buf, int mode , size_t size);
```

Nota 2: En la parte de llamadas al sistema para el sistema de archivos vimos que en UNIX se definen tres macros `STDIN_FILENO`, `STDOUT_FILENO` y `STDERR_FILENO` para poder utilizar las llamadas al sistema `read` y `write` (que trabajan con ***descriptores de archivo***) sobre la entrada estándar, la salida estándar y el error estándar del proceso. Además, en `<stdio.h>` se definen tres flujos (`STREAM`) para poder trabajar sobre estos archivos especiales usando las funciones de la biblioteca de E/S del C: `stdin`, `stdout` y `stderr`.

```
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;
```

¡Fíjate que `setvbuf` es una función que trabaja sobre `STREAMS`, no sobre ***descriptores de archivo***!

2. Implementa un programa que lance cinco procesos hijo. Cada uno de ellos se identificará en la salida estándar, mostrando un mensaje del tipo `Soy el hijo PID`. El proceso padre simplemente tendrá que esperar la finalización de todos sus hijos y cada vez que detecte la finalización de uno de sus hijos escribirá en la salida estándar un mensaje del tipo:

```
Acaba de finalizar mi hijo con <PID>
 Sólo me quedan <NUM_HIJOS> hijos vivos.
```

3. Implementa una modificación sobre el anterior programa en la que el proceso padre espera primero a los hijos creados en orden impar (1º,3º,5º) y después a los hijos pares (2º y 4º).

2.4 Familia de llamadas 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; el del programa que se le pasa como argumento, y este programa comienza a ejecutarse en el contexto del proceso hijo empezando en la función `main`. El PID del proceso no cambia ya que no se crea ningún proceso nuevo.

```
#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 de estas llamadas es el camino del archivo ejecutable.

El `const char *arg` y puntos suspensivos siguientes en las funciones `execl`, `execlp`, y `execle` son los argumentos (incluyendo su propio nombre) del programa a ejecutar: `arg0, arg1, ..., argn`. Todos juntos, describen una lista de uno o más punteros a cadenas de caracteres terminadas en cero. El primer argumento, por convenio, debe apuntar al nombre de archivo asociado con el programa que se esté ejecutando. La lista de argumentos debe ser terminada por un puntero NULL.

Las funciones `execv` y `execvp` proporcionan un vector de punteros a cadenas de caracteres terminadas en cero, que representan la lista de argumentos disponible para el nuevo programa. El primer argumento, por convenio, debe apuntar al nombre de archivo asociado con el programa que se esté ejecutando. El vector de punteros debe ser terminado por un puntero NULL.

La función `execle` especifica *el entorno del proceso que ejecutará el programa* mediante un parámetro adicional que va detrás del puntero NULL que termina la lista de argumentos de la lista de parámetros o el puntero al vector `argv`. Este parámetro adicional es un vector de punteros a cadenas de caracteres acabadas en cero y debe ser terminada por un puntero NULL. Las otras funciones obtienen el entorno para la nueva imagen de proceso de la variable externa `environ` en el proceso en curso.

Algunas particularidades de las funciones que hemos descrito previamente:

Las funciones `execlp` y `execvp` duplicarán las acciones del shell al buscar un archivo ejecutable si el nombre de archivo especificado no contiene un carácter de barra inclinada (/). El camino de búsqueda es el especificado en el entorno por la variable `PATH`. Si esta variable no es especificada, se emplea el camino predeterminado "`:/bin:/usr/bin`".

Si a un archivo se le deniega el permiso (`execve` devuelve `EACCES`), estas funciones continuarán buscando en el resto del camino de búsqueda. Si no se encuentra otro archivo devolverán el valor `EACCES` en la variable global `errno` .

Si no se reconoce la cabecera de un archivo (la función `execve` devuelve `ENOEXEC`), estas funciones ejecutarán el shell con el camino del archivo como su primer argumento.

Las funciones `exec` fallarán de forma generalizada en los siguientes casos:

- El sistema operativo no tiene recursos suficientes para crear el nuevo espacio de direcciones de usuario.
- Utilizamos de forma errónea el paso de argumentos a las distintas funciones de la familia `exec`.

Actividad 4. Trabajo con llamadas al sistema de ejecución de programas.

Consulta en el manual en línea las distintas funciones de la familia `exec` y fíjate bien en las diferencias en cuanto a paso de parámetros que existen entre ellas.

1. ¿Qué hace el siguiente programa?

```
/*
tarea8.c
Trabajo con llamadas al sistema del Subsistema de Procesos conforme a POSIX 2.10
*/

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


int main(int argc, char *argv[])
{
pid_t pid;
int estado;

if( (pid=fork())<0) {
		perror("\nError en el fork");
		exit(-1);
}
else if(pid==0) {  //proceso hijo ejecutando el programa
		if( (execl("/usr/bin/ldd","ldd","./tarea8",NULL)<0)) {
		perror("\nError en el execl");
		    exit(-1);
		}
}
wait(&estado);
printf("\nMi hijo %d ha finalizado con el estado %d\n",pid,estado);

exit(0);

}
```

2. Escribe un programa que acepte nombres de programa escritos en la entrada estándar, los ejecute en background o foreground según lo desee el usuario, y proporcione en la salida estándar el resultado de la ejecución de dichos programas. Es decir, construye un shell reducido a la funcionalidad de ejecución de programas.

3 Señales

Las señales son un mecanismo básico de sincronización que utiliza el núcleo de UNIX para indicar a los procesos la ocurrencia de determinados eventos síncronos/asíncronos a su ejecución. Aparte del uso de señales por parte del núcleo, los procesos pueden enviarse señales y, lo que es más importante, pueden determinar que acción realizarán como respuesta a la recepción de una señal determinada. Las llamadas al sistema que podemos utilizar en Linux para trabajar con señales son principalmente:

- `sigaction`, que permite establecer la acción que realizará un proceso como respuesta a la recepción de una señal.
- `sigprocmask`, se emplea para cambiar la lista de señales bloqueadas actualmente.
- `sigpending`, permite el examen de señales pendientes (las que se han producido mientras estaban bloqueadas).
- `sigsuspend`, reemplaza temporalmente la máscara de señal para el proceso con la dada por `mask` y luego suspende el proceso hasta que se recibe una señal.
- `kill`, que se utiliza para enviar una señal a un proceso o conjunto de procesos

A continuación se muestra la declaración de las llamadas que permiten trabajar con el mecanismo de señales en Linux y después iremos viendo cada una de ellas detalladamente:

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
int kill(pid_t pid, int sig);
```

La llamada sigaction

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

La estructura `sigaction` se define como

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

Nota: El elemento `sa_restorer` está obsoleto y no debería utilizarse. POSIX no especifica un elemento `sa_restorer`.

`sa_handler` especifica la acción que se va a asociar con `signum` y puede ser `SIG_DFL` para la acción predeterminada, `SIG_IGN` para no tener en cuenta la señal, o un puntero a una función manejadora para la señal.

`sa_mask` permite establecer una máscara de señales que deberían bloquearse durante la ejecución del manejador de señal. Además, la señal que lance el manejador será bloqueada, a menos que se activen las opciones `SA_NODEFER` o `SA_NOMASK`.

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
Si signum es SIGCHLD, indica al núcleo que el proceso no desea recibir notificación cuando los procesos hijos se paren (esto es, cuando los procesos hijos reciban una de las señales SIGSTOP, SIGTSTP, SIGTTIN o SIGTTOU).

SA_ONESHOT o SA_RESETHAND
Indica al núcleo que restaure la acción para la señal al estado predeterminado una vez que el manejador de señal haya sido llamado.

SA_RESTART
Proporciona un comportamiento compatible con la semántica de señales de BSD haciendo re-ejecutables algunas llamadas al sistema entre señales.

SA_NOMASK o SA_NODEFER
Se pide al núcleo que no impida la recepción de la señal desde su propio manejador.

SA_SIGINFO
El manejador de señal toma 3 argumentos, no uno. En este caso, se debe configurar sa_sigaction en lugar de sa_handler.

El parámetro siginfo_t para sa_sigaction es una estructura con los siguientes elementos

```
siginfo_t {
   int      si_signo;  /* Número de señal */
   int      si_errno;  /* Un valor errno */
   int      si_code;   /* Código de señal */
   pid_t    si_pid;    /* ID del proceso emisor */
   uid_t    si_uid;    /* ID del usuario real del proceso emisor */
   int      si_status; /* Valor de salida o señal */
   clock_t  si_utime;  /* Tiempo de usuario consumido */
   clock_t  si_stime;  /* Tiempo de sistema consumido */
   sigval_t si_value;  /* Valor de señal */
   int      si_int;    /* señal POSIX.1b */
   void *   si_ptr;    /* señal POSIX.1b */
   void *   si_addr;   /* Dirección de memoria que ha producido el fallo */
   int      si_band;   /* Evento de conjunto */
   int      si_fd;     /* Descriptor de fichero */
}
```

- si_signo, si_errno y si_code están definidos para todas las señales. kill, las señales POSIX.1b y SIGCHLD actualizan los campos si_pid y si_uid. SIGCHLD también actualiza los campos si_status, si_utime y si_stime. El emisor de la señal POSIX.1b especifica los campos si_int y si_ptr.
- SIGILL, SIGFPE, SIGSEGV y SIGBUS provocan que el núcleo actualice el campo si_addr con la dirección del fallo. SIGPOLL rellena si_band y si_fd.
- si_code indica la causa por la cuál se ha enviado la señal. Es un valor, no una máscara de bits.

Los valores que son posibles para cualquier señal se listan en la siguiente tabla:

si_code	
Valor	Origen de la señal
SI_USER	kill, sigsend o raise
SI_KERNEL	El núcleo
SI_QUEUE	sigqueue
SI_TIMER	el cronómetro ha finalizado
SI_MESGQ	ha cambiado el estado de mesq
SI_ASYNCIO	ha terminado una E/S asíncrona
SI_SIGIO	SIGIO encolada

SIGILL	
ILL_ILLOPC	código de operación ilegal
ILL_ILLOPN	operando ilegal
ILL_ILLADR	modo de direccionamiento ilegal
ILL_ILLTRP	trampa ilegal
ILL_PRVOPC	código de operación privilegiada
ILL_PRVREG	registro privilegiado
ILL_COPROC	error del coprocesador
ILL_BADSTK	error de la pila interna

SIGFPE	
FPE_INTDIV	entero dividido por cero
FPE_INTOVF	desbordamiento de entero
FPE_FLTDIV	punto flotante dividido por cero
FPE_FLTOVF	desbordamiento de punto flotante
FPE_FLTUND	desbordamiento de punto flotante por defecto
FPE_FLTRES	resultado de punto flotante inexacto
FPE_FLTINV	operación de punto flotante inválida
FPE_FLTSUB	subíndice (subscript) fuera de rango

SIGSEGV	
SEGV_MAPERR	dirección no asociada a un objeto
SEGV_ACCERR	permisos inválidos para un objeto presente en memoria

SIGBUS	
BUS_ADRALN	alineamiento de dirección inválido
BUS_ADRERR	dirección física inexistente
BUS_OBJERR	error hardware específico del objeto

SIGTRAP	
TRAP_BRKPT	punto de parada de un proceso
TRAP_TRACE	trampa de seguimiento paso a paso de un proceso

SIGCHLD	
CLD_EXITED	ha terminado un hijo
CLD_KILLED	se ha matado a un hijo
CLD_DUMPED	un hijo ha terminado anormalmente
CLD_TRAPPED	un hijo con seguimiento paso a paso ha sido detenido
CLD_STOPPED	ha parado un hijo
CLD_CONTINUED	un hijo parado ha continuado

SIGPOLL	
POLL_IN	datos de entrada disponibles
POLL_OUT	buffers de salida disponibles
POLL_MSG	mensaje de entrada disponible
POLL_ERR	error de E/S
POLL_PRI	entrada de alta prioridad disponible
POLL_HUP	dispositivo desconectado

Llamadas sigprocmask, sigpending y sigsuspend

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

La llamada `sigprocmask` se emplea para cambiar la lista de señales bloqueadas actualmente. El comportamiento de la llamada depende del valor de `how`, como sigue:

`SIG_BLOCK`
El conjunto de señales bloqueadas es la unión del conjunto actual y el argumento `set`.

`SIG_UNBLOCK`

Las señales en `set` se quitan del conjunto actual de señales bloqueadas. Es legal intentar el desbloqueo de una señal que no está bloqueada.

`SIG_SETMASK`
El conjunto de señales bloqueadas se pone según el argumento `set`.

Si `oldset` no es `NULL`, el valor anterior de la máscara de señal se guarda en `oldset`.

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

La llamada `sigpending` permite el examen de señales pendientes (las que han sido producidas mientras estaban bloqueadas). La máscara de señal de las señales pendientes se guarda en `set`.

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

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

Notas finales

- No es posible bloquear `SIGKILL` ni `SIGSTOP` con una llamada a `sigprocmask`. Los intentos de hacerlo no serán tenidos en cuenta por el núcleo.
- De acuerdo con POSIX, el comportamiento de un proceso está indefinido después de que no haga caso de una señal `SIGFPE`, `SIGILL` o `SIGSEGV` que no haya sido generada por las llamadas `kill` o `raise`. La división entera por cero da un resultado indefinido. En algunas arquitecturas generará una señal `SIGFPE`. No hacer caso de esta señal puede llevar a un bucle infinito.
- `sigaction` puede llamarse con un segundo argumento nulo para saber el manejador de señal en curso. También puede emplearse para comprobar si una señal dada es válida para la máquina donde se está, llamándola con el segundo y el tercer argumento nulos.
- POSIX (B.3.3.1.3) anula el establecimiento de `SIG_IGN` como acción para `SIGCHLD`. Los comportamientos de BSD y SYSV difieren, provocando el fallo en Linux de aquellos programas BSD que asignan `SIG_IGN` como acción para `SIGCHLD`.
- La especificación POSIX sólo define `SA_NOCLDSTOP`. El empleo de otros valores en `sa_flags` no es portable.
- La opción `SA_RESETHAND` es compatible con la de SVr4 del mismo nombre.
- La opción `SA_NODEFER` es compatible con la de SVr4 del mismo nombre bajo a partir del núcleo 1.3.9.
- Los nombres `SA_RESETHAND` y `SA_NODEFER` para compatibilidad con SVr4 están presentes solamente en la versión de la biblioteca 3.0.9 y superiores.
- La opción `SA_SIGINFO` viene especificada por POSIX.1b. El soporte para ella se añadió en la versión 2.2 de Linux.

La llamada `kill`

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

int kill(pid_t pid, int sig);
```

La llamada `kill` se puede usar para enviar cualquier señal a un proceso o grupo de procesos.

- Si `pid` es positivo, entonces la señal `sig` es enviada al proceso con identificador de proceso igual a `pid`. En este caso, se devuelve 0 si hay éxito, o un valor negativo si hay 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.

Actividad 5. Trabajo con las llamadas al sistema `sigaction` y `kill`.

1. A continuación se muestra el código fuente de dos programas. El programa `envioSignal` permite el envío de una señal a un proceso identificado por medio de su `PID`. El programa `reciboSignal` se ejecuta en background y permite la recepción de señales.

```
/*
 reciboSignal.c
 Trabajo con llamadas al sistema del Subsistema de Procesos conforme a POSIX 2.10
 Utilización de la llamada sigaction para cambiar el comportamiento del proceso frente
  a la recepción de una señal.
*/

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

static void sig_USR_hdlr(int sigNum)
{
if(sigNum == SIGUSR1)
printf("\nRecibida la señal SIGUSR1\n\n");
else if(sigNum == SIGUSR2)
		printf("\nRecibida la señal SIGUSR2\n\n");
}


int main(int argc, char *argv[])
{
struct sigaction sig_USR_nact;

if(setvbuf(stdout,NULL,_IONBF,0)) {
		perror("\nError en setvbuf");
}

//Inicializar la estructura sig_USR_na para especificar la nueva acción para la señal.
sig_USR_nact.sa_handler= sig_USR_hdlr;
//'sigemptyset' inicia el conjunto de señales dado al conjunto vacío.
sigemptyset (&sig_USR_nact.sa_mask);
sig_USR_nact.sa_flags = 0;
```

```
//Establecer mi manejador particular de señal para SIGUSR1
if( sigaction(SIGUSR1,&sig_USR_nact,NULL) <0) {
		perror("\nError al intentar establecer el manejador de señal para SIGUSR1");
		exit(-1);
}
//Establecer mi manejador particular de señal para SIGUSR2
if( sigaction(SIGUSR2,&sig_USR_nact,NULL) <0) {
		perror("\nError al intentar establecer el manejador de señal para SIGUSR2");
		exit(-1);
}

for(;;)
{
}

}


/*
 envioSignal.c
 Trabajo con llamadas al sistema del Subsistema de Procesos conforme a POSIX 2.10
 Utilización de la llamada kill para enviar una señal:
		0: SIGTERM
		1: SIGUSR1
		2: SIGUSR2
  a un proceso cuyo identificador de proceso es PID.
 SINTAXIS:
				envioSignal [012] <PID>
*/

#include <sys/types.h>//Primitive system data types for abstraction of implementation-dependent data
types.
						//POSIX Standard: 2.6 Primitive System Data Types
<sys/types.h>
#include<limits.h>  //Incluye <bits/posix1_lim.h> POSIX Standard: 2.9.2 Minimum Values    Added to
<limits.h>
					//y <bits/posix2_lim.h>
#include <unistd.h>		//POSIX Standard: 2.10 Symbolic Constants		<unistd.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>


int main(int argc, char *argv[])
{
long int pid;
int signal;

if(argc<3) {
		printf("\nSintaxis de ejecución: envioSignal [012] <PID>\n\n");
		exit(-1);
}

pid= strtol(argv[2],NULL,10);
if(pid == LONG_MIN || pid == LONG_MAX) {
		if(pid == LONG_MIN)
		printf("\nError por desbordamiento inferior LONG_MIN %d",pid);
		else
		printf("\nError por desbordamiento superior LONG_MAX %d",pid);
		perror("\nError en strtol");
		exit(-1);
```

```
}

signal=atoi(argv[1]);

switch(signal) {
        case 0: //SIGTERM
                kill(pid,SIGTERM);
                break;
        case 1: //SIGUSR1
                kill(pid,SIGUSR1);
                break;
        case 2: //SIGUSR2
                kill(pid,SIGUSR2);
                break;
        default : // not in [012]
                printf("\nNo puedo enviar ese tipo de señal");
}

}
```

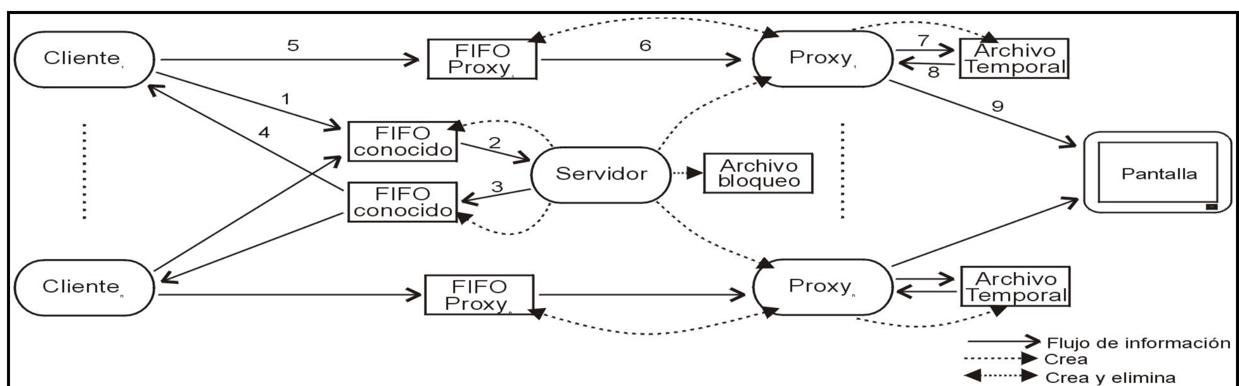
Actividad 6. Trabajo con llamadas al sistema del subsistema de procesos.

Revisa de nuevo en el manual en línea el uso las funciones relacionadas con la comunicación entre procesos: `pipe`, `mknod`, `mkfifo`.

Revisa también las funciones relacionadas con el manejo de señales: `sigaction` y `kill`.

1. Este ejercicio tratará aspectos relacionados con procesos, señales y cauces con nombre (archivos FIFO).

Se pretende construir un ***spool*** concurrente de impresión en pantalla. Su funcionamiento general consiste en controlar el acceso de procesos clientes al recurso compartido, en este caso la pantalla, garantizando la utilización en exclusión mutua de dicho recurso. Recuerde que un sistema *spool* para impresión imprime un documento sólo cuando éste se ha generado por completo. De esta forma, se consigue que un proceso no pueda apropiarse indefinidamente, o durante mucho tiempo si el proceso es lento, del recurso compartido. Como mecanismo de comunicación/sincronización entre procesos se van a utilizar ***cauces con nombre*** (archivos FIFO) y en algún caso señales. En la siguiente figura se muestra el esquema general a seguir para la implementación:



Siguiendo los mensajes numerados obtendremos las interacciones entre procesos para poder llevar a cabo la impresión de un archivo, según se explica a continuación:

- Un cliente solicita la impresión de un archivo enviando un mensaje al servidor a través de un FIFO cuyo nombre es conocido. Este mensaje es el PID del proceso cliente, es decir, los 4 bytes de los que consta un dato de tipo entero.
- El servidor lee esta petición delegando la recepción e impresión del documento en un proceso (*proxy*) que crea específicamente para atender a dicho cliente. Una vez servida dicha petición, el *proxy* terminará.
- El servidor (o el *proxy*) responde al cliente a través de otro FIFO de nombre conocido, informando de la identidad (PID) del *proxy*. Este dato es la base para poder comunicar al cliente con el *proxy*, ya que éste creará un nuevo archivo FIFO especifico para esta comunicación, cuyo nombre es *fifo.<PID>*, donde *<PID>* es el PID del proceso *proxy*. El *proxy* se encargará de eliminar dicho FIFO cuando ya no sea necesario.
- El cliente lee esta información que le envía el servidor, de manera que así sabrá donde enviar los datos a imprimir.
- Probablemente el cliente necesitará enviar varios mensajes como éste, tantos como sean necesarios para transmitir toda la información a imprimir. El final de la transmisión de la información lo indicará con un fin de archivo.
- El *proxy* obtendrá la información a imprimir llevando a cabo probablemente varias lecturas como ésta del FIFO
- Por cada lectura anterior, tendrá lugar una escritura de dicha información en un archivo temporal, creado específicamente por el *proxy* para almacenar completamente el documento a imprimir.
- Una vez recogido todo el documento, volverá a leerlo del archivo temporal justo después de comprobar que puede disponer de la pantalla para iniciar la impresión en exclusión mutua.
- Cada lectura de datos realizada en el paso anterior implicará su escritura en pantalla.

Ten en cuenta las siguientes consideraciones:

- Utilice un tamaño de 1024 bytes para las operaciones de lectura/escritura (mediante las llamadas al sistema *read*/*write*) de los datos del archivo a imprimir.
- Recuerde que cuando todos los procesos que tienen abierto un FIFO para escritura lo cierran, o dichos procesos terminan, entonces se genera automáticamente un fin de archivo que producirá el desbloqueo del proceso que esté bloqueado esperando leer de dicho FIFO, devolviendo en este caso la llamada al sistema *read* la cantidad de 0 Bytes leídos. Si en algún caso, como por ejemplo en el servidor que lee del FIFO conocido no interesa este comportamiento, entonces la solución más directa es abrir el FIFO en modo lectura/escritura (*O_RDWR*) por parte del proceso servidor. Así nos aseguramos que siempre al menos un proceso va a tener el FIFO abierto en escritura, y por tanto se evitan la generación de varios fin de archivo.
- Siempre que cree un archivo, basta con que especifique en el campo de modo la constante *S_IRWXU* para que el propietario tenga todos los permisos (lectura, escritura, ejecución) sobre el archivo. Por supuesto, si el sistema se utilizara en una situación real habría que ampliar estos permisos a otros usuarios.
- Utilice la función `tmpfile` incluida en la biblioteca estándar para el archivo temporal que crea cada *proxy*, así su eliminación será automática. Tenga en cuenta que esta función devuelve un puntero a un `STREAM` (`FILE *`), y por tanto las lecturas y escrituras se realizarán con las funciones de la biblioteca estándar `fread` y `fwrite` respectivamente.
- No deben quedar procesos *zombis* en el sistema. Podemos evitarlo atrapando en el servidor las señales `SIGCHLD` que envían los procesos *proxy* (ya que son hijos del servidor) cuando terminan. Por defecto la acción asignada a esta señal es ignorarla, pero mediante la llamada al sistema *signal* podemos especificar un manejador que ejecute la llamada *wait* impidiendo que los procesos *proxy* queden como *zombis*.

- El programa *proxy* por cuestiones de reusabilidad, leerá de su entrada estándar (macro `STDIN_FILENO`) y escribirá en su salida estándar (macro `STDOUT_FILENO`). Por tanto, hay que redireccionar su entrada estándar al archivo FIFO correspondiente, esto se puede llevar a cabo mediante la llamada al sistema *dup2*.
- Para conseguir el bloqueo/desbloqueo de pantalla a la hora de imprimir, utilice la función que se muestra a continuación. Esta recibe como parámetros un descriptor del archivo que se utiliza para bloqueo (por tanto ya se ha abierto previamente) y la orden a ejecutar, es decir, bloquear (*F_WRLCK*) o desbloquear (*F_UNLCK*):

```
void bloqueodesbloqueo (int dbloqueo, int orden) {

struct flock cerrojo;

// Inicializamos el cerrojo para bloquear todo el archivo
cerrojo.l_type= orden;
cerrojo.l_whence= SEEK_SET; //desde el origen del archivo
cerrojo.l_start= 0; //situandose en el byte 0 (offset)
cerrojo.l_len = 0; //bloquea hasta el fin del archivo

//Si vamos a bloquearlo y ya lo esta, entonces el proceso
//duerme
if (fcntl(dbloqueo, F_SETLKW, &cerrojo)<0) {
  perror ("Proxy: problemas al bloquear para impresión");
  exit(-1);
}
}
```

También se proporciona un programa denominado *clientes* capaz de lanzar hasta 10 clientes solicitando la impresión de datos. Para simplificar y facilitar la comprobación del funcionamiento de todo el sistema, cada uno de los clientes pretende imprimir un archivo de tamaño desconocido pero con todos los caracteres idénticos, es decir, un cliente imprimirá sólo caracteres *a*, otro sólo caracteres *b*, y así sucesivamente. El formato de ejecución de este programa es:

```
prompt> clientes <nombre_fifos_conocidos> <número_clientes>
```

El argumento <nombre_fifos_conocidos> es un único nombre, de forma que los clientes suponen que el nombre del FIFO conocido de entrada al servidor es dicho nombre concatenado con el carácter 'e'. En el caso del FIFO de salida, se concatena dicho nombre con el carácter 's'.

Implementa el resto de programas (servidor y proxy) según lo descrito, para ello necesitarás además de las funciones y llamadas al sistema anteriormente comentadas, otras como: `mkfifo` (crea un archivo FIFO), `creat` (crea un archivo normal), `open` (abre un archivo, opcionalmente lo puede crear si no existe), `close` (cierra un archivo especificando su descriptor), `fork` (crea un proceso), `exec` (pasa a ejecutar un programa en un archivo ejecutable), `getpid` (devuelve el PID del proceso) y `unlink` (borra un archivo de cualquier tipo).

Ten en cuenta que el servidor debe ser un proceso demonio que está permanentemente ejecutándose, por tanto, tendrás que ejecutarlo en *background* y siempre antes de lanzar los clientes. Asegúrate también de que el servidor cree los archivos FIFO conocidos antes de que los clientes intenten comunicarse con él.