Prácticas de Sistemas Operativos

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

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

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

Sesión 1. Llamadas al sistema para el SA (Parte I)

1. Objetivos del Módulo II

El primer objetivo de módulo 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 (lo habéis practicado en la asignatura Fundamentos del Software y en el primer módulo de esta asignatura) 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 Linux es conveniente disponer de la siguiente documentación:

25. 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: **`libc.info`** o **`libc.html`** (**`glibc.html`**).

26. Manual básico de C para consultar las diferencias con C++. En internet podéis encontrar mucha información sobre programación básica con C. Repasad los conceptos vistos en la asignatura de programación.

27. Manual en línea del sistema: **`man`** o **`info`**

La siguiente tabla muestra los números de sección del manual y los tipos de páginas que contienen. Se puede acceder a una determinada sección utilizando la siguiente orden:

```
$> man <numero_sección> <orden>
```

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***)

Nota 1: Os suministraremos los programas de ejemplo que están referenciados en el guion de prácticas. Estos programas están implementados en C, por tanto, no debéis olvidar compilarlos con el compilador de C que es **`gcc`** (**NO** **`g++`**). Vosotros también tenéis que trabajar en C y crear vuestros programas con este lenguaje.

Nota 2: 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 el mensaje de error del sistema, 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.

A continuación os mostramos el prototipo de otras funciones que permiten incluir variables en la salida del error.

```
#include <err.h>

      void err(int eval, const char *fmt, ...);

#include <stdarg.h>

       void verr(int eval, const char *fmt, va_list args);
```

Nota 3: Una orden que es útil para rastrear la ejecución de un programa es **`strace`**. Ejecuta el programa que se pasa como argumento y proporciona información de las llamadas al sistema que han sido invocadas (junto con el valor de los argumentos y el valor de retorno) y de las señales recibidas.

2. Objetivos principales

Esta sesión está pensada para trabajar con el sistema de archivos pero solicitando los servicios al sistema operativo utilizando las llamadas al sistema. Veremos cómo abrir un archivo, cerrarlo, leer o escribir en él.

- Conocer y saber usar las órdenes para poder trabajar (leer, escribir, cambiar el puntero de lectura/escritura, abrir y cerrar un archivo) con archivos regulares desde un programa implementado en un lenguaje de alto nivel como C.
- Conocer los atributos o metadatos que guarda Linux para un archivo.
- Saber usar las llamadas al sistema que nos permiten obtener los metadatos o atributos de un archivo.

3. 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 Linux 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 la salida de 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 3.1 *Trabajo con llamadas de gestión y procesamiento sobre archivos* regulares

Consulta 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)`**.

Mira la llamada al sistema **`close`** en el manual en línea.

Mira la llamada al sistema **lseek** fijándote en las posibilidades de especificación del nuevo **current_offset**.

Mira 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.

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

Ejercicio 1. ¿Qué hace el siguiente programa? Probad 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''
Probad 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);
}
close(fd);
return 0;
}
```

Ejercicio 2. Implementa un programa que realice la siguiente funcionalidad. El programa acepta como argumento el nombre de un archivo (*pathname*), lo abre y lo lee en bloques de tamaño 80 Bytes, y crea un nuevo archivo de salida, **salida.txt**, en el que debe aparecer la siguiente información:

```
Bloque 1
// Aquí van los primeros 80 Bytes del archivo pasado como argumento.
Bloque 2
// Aquí van los siguientes 80 Bytes del archivo pasado como argumento.
...
```

```
Bloque n
// Aquí van los siguientes 80 Bytes del archivo pasado como argumento.
```

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 <n°_bloques>
Bloque 1
// Aquí van los primeros 80 Bytes del archivo pasado como argumento.
Bloque 2
// Aquí van los siguientes 80 Bytes del archivo pasado como argumento.
...
```

4. Metadatos de un Archivo

En el punto 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.

4.1 Tipos de archivos

Linux soporta los siguientes tipos de archivos:

● Archivo regular. Contiene datos de cualquier tipo. No existe distinción para el núcleo de Linux 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).

4.2 Estructura **stat**

Los metadatos de un archivo, se pueden obtener con la llamada al sistema `stat` que utiliza una estructura de datos llamada `stat` para almacenar dicha información. 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 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 */
};
```

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)**.

Normalmente, **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) 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
```

Se definen las siguientes banderas (flags) para trabajar con el campo **st_mode:**

S_IFMT	0170000	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	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

4.3 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 3.2. Trabajo con llamadas al sistema de la familia `stat`.

Consulta las llamadas al sistema **stat** y **lstat** para entender sus diferencias.

Ejercicio 3. ¿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<stdlib.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;
}
```

Ejercicio 4. Define una macro en lenguaje C que tenga la misma funcionalidad que 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. Consulta en un libro de C o en internet cómo se especifica una macro con argumento en C.

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

Nota: Puede ser interesante para depurar la ejecución de un programa en C que utilice llamadas al sistema usar la orden `strace`. Esta orden, en el caso más simple, ejecuta un programa hasta que finalice e intercepta y muestra las llamadas al sistema que realiza el proceso junto con sus argumentos y devuelve los valores devueltos en la salida de error estándar o en un archivo si se especifica la opción **`-o`**. Obtén más información con `man`.

Prácticas de Sistemas Operativos

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

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

Sesión 2. Llamadas al sistema para el SA (Parte II)

1. Objetivos principales

Esta sesión está pensada para trabajar con las llamadas al sistema que modifican los permisos de un archivo y con los directorios.

- Conocer y saber usar las órdenes para poder modificar y controlar los permisos de los archivos que crea un proceso basándose en la máscara que tiene asociado el proceso.
- Conocer las funciones y estructuras de datos que nos permiten trabajar con los directorios.
- Comprender los conceptos e implementaciones que utiliza un sistema operativo UNIX para construir las abstracciones de archivos y directorios.

2. Llamadas al sistema relacionadas con los permisos de los archivos

En este punto trabajaremos ampliando la información que ya hemos visto en la sesión 1. Vamos a entender por qué cuando un proceso crea un archivo se le asignan a dicho archivo unos permisos concretos. También nos interesa controlar los permisos que queremos que tenga un archivo cuando se crea.

2.1 La llamada al sistema `umask`

La llamada al sistema **`umask`** fija la máscara de creación de permisos para el proceso y devuelve el valor previamente establecido. El argumento de la llamada puede formarse mediante 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

SINOPSIS

```
#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 del archivo que se va a crear. Específicamente, los permisos presentes en la máscara se desactivan del argumento **`mode`** de **`open`** (así pues, por ejemplo, si creamos un archivo con campo **`mode= 0666`** y tenemos el valor común por defecto de **`umask=022`**, este archivo se creará con permisos: **`0666 & ~022 = 0644 = rw-r--r--`**, que es el caso más normal).

VALOR DEVUELTO

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

2.2 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 archivo 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 los permisos del archivo dado mediante `path` o referido por `fildes`. Los permisos se pueden especificar mediante un `OR` lógico de los siguientes valores:

S_ISUID	04000	activar la asignación del UID del propietario al UID efectivo del proceso que ejecute el archivo.
S_ISGID	02000	activar la asignación del GID del propietario al GID efectivo del proceso que ejecute el archivo.
S_ISVTX	01000	activar *sticky* bit. En directorios significa un borrado restringido, es decir, un proceso no privilegiado no puede borrar o renombrar archivos del directorio salvo que tenga permiso de escritura y sea propietario. Por ejemplo se utiliza en el directorio **/tmp**.
S_IRWXU	00700	**user** (propietario del archivo) tiene permisos de lectura, escritura y ejecución
S_IRUSR	00400	lectura 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 root o 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 root 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 root).

Actividad 2.1 Trabajo con llamadas de cambio de permisos

Consulta el manual en línea para las llamadas al sistema `umask` y `chmod`.

Ejercicio 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<stdlib.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);
}
close(fd1);
close(fd2);

return 0;
}
```

3. Funciones de manejo de 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 **pathname** 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 (se tiene que usar en combinación con **telldir**).

· **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;


//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 */
      char d_name[256]; /* nombre del archivo */
};
```

Actividad 2.2 Trabajo con funciones estándar de manejo de directorios

Mirad 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.

Ejercicio 2. Realiza 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`**). Para convertir este argumento tipo cadena a un tipo numérico puedes utilizar la función **`strtol`**. Consulta el manual en línea para conocer sus argumentos.

El programa tiene que usar el número octal indicado en el segundo argumento para cambiar los permisos de todos los archivos que se encuentren en el directorio indicado en el primer argumento.
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>
```

Ejercicio 3. Programa 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 inodo y la suma total de espacio ocupado por dichos archivos. El formato de la nueva orden será:

```
$> ./buscar <pathname>
```

donde **<pathname>** especifica la ruta 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
```

Actividad 2.3 Trabajo con la llamada `nftw()` para recorrer un sistema de archivos

Si bien con las funciones anteriores podemos recorrer el sistema de archivos de forma “manual”, la función **`nftw()`** permite recorrer recursivamente un sub-árbol y realizar alguna operación sobre los archivos del mismo.

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

NOMBRE

nftw – permite recorrer recursivamente un sub-árbol del sistema de archivos

SINOPSIS

```
#define _XOPEN_SOURCE 500
#include <ftw.h>

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

DESCRIPCIÓN

La función recorre el árbol de directorios especificado por **`dirpath`** y llama a la función **`func`** definida por el programador para cada archivo del árbol. Por defecto, **`ntfw`** realiza un recorrido no ordenado en preorden del árbol, procesando primero cada directorio antes de procesar los archivos y subdirectorios dentro del directorio.

Mientras se recorre el árbol, la función **`nftw`** abre al menos un descriptor de archivo por nivel del árbol. El parámetro **`nopenfd`** especifica el máximo número de descriptores que puede usar. Si la profundidad del árbol es mayor que el número de descriptores, la función evita abrir más cerrando y reabriendo descriptores.

El argumento **`flags`** es creado mediante un OR (|) con cero o más constantes, que modifican la operación de la función:

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.

Para cada archivo, **nftw()** pasa cuatro argumentos al invocar a **func**. El primero, **pathname** indica el nombre del archivo, que puede ser absoluto o relativo dependiendo de si **dirpath** es de un tipo u otro. El argumento **statbuf** es un puntero a una estructura **stat** conteniendo información del archivo. El tercer argumento, **typeflag**, suministra información adicional sobre el archivo, y tiene uno de los siguientes nombres simbólicos:

FTW_D	Es un directorio
FTW_DNR	Es un directorio que no puede leerse (no se lee sus descendientes).
FTW_DP	Estamos haciendo un recorrido posorden de un directorio, y el ítem actual es un directorio cuyos archivos y subdirectorios han sido recorridos.
FTW_F	Es un archivo de cualquier tipo diferente de un directorio o enlace simbólico.
FTW_NS	**stat** ha fallado sobre este archivo, probablemente debido a restricciones de permisos. El valor **statbuf** es indefinido.
FTW_SL	Es un enlace simbólico. Este valor se retorno solo si **nftw** se invoca con FTW_PHYS
FTW_SLN	El ítem es un enlace simbólico perdido. Este se da cuando no se especifica FTW_PHYS.

El cuarto elemento de **func** es **ftwbuf**, es decir, un puntero a una estructura que se define de la forma:

```
struct FTW {
       int base;     /* Desplazamiento de la parte base del pathname */
       int level;    /*Profundidad del archivo dentro recorrido del arbol */
};
```

El campo **base** es un desplazamiento entero de la parte del nombre del archivo (el componente después del último /) del **pathname** pasado a **func**.

Cada vez que es invocada, **func** debe retornar un valor entero que es interpretado por **nftw**. Si retorna 0, indica a **nftw** que continúe el recorrido del árbol. Si todas las invocaciones a **func** retornan cero, **nftw** retornará 0. Si retorna distinto de cero, **nftw** para inmediatamente, en cuyo caso **nftw** retorna este valor como su valor de retorno.

VALOR DEVUELTO

Esta llamada al sistema devuelve 0 si recorre completamente el árbol; -1 si error o el primer valor no cero devuelto por **func**.

El siguiente programa muestra un ejemplo de uso de la función. En este caso, utilizamos **nftw** para recorrer el directorio pasado como argumento, salvo que no se especifique, en cuyo caso, actuamos sobre el directorio actual. Para cada elemento atravesado se invoca a la función **visitar()** que imprime el pathname y el modo en octal. Observa que para que el recorrido sea completo la función **visitar** debe devolver un 0, sino se detendría la búsqueda.

```
/* Programa que recorre un sub-árbol con la función nftw */

#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>

#include <ftw.h>

int visitar(const char* path, const struct stat* stat, int flags, struct FTW*
ftw) {
    printf ("Path: %s Modo: %o",path, stat->st_mode);
    return 0;
}

int main(int argc, char** argv) {
   if (nftw(argc >= 2 ? argv[1] : ".", visitar, 10, 0) != 0) {
   perror("nftw");
   }
}
```

Ejercicio 4. Implementa de nuevo el programa **`buscar`** del ejercicio 3 utilizando la llamada al sistema **`nftw`**.

Prácticas de Sistemas Operativos

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

Sesión 3. Llamadas al sistema para el Control de Procesos

Sesión 3. Llamadas al sistema para el Control de Procesos

1. Objetivos principales

Esta sesión trabajaremos con las llamadas al sistema relacionadas con el control y la gestión 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. En concreto, los objetivos son:

- Conocer y saber usar las órdenes para crear un nuevo proceso, finalizar un proceso, esperar a la terminación de un proceso y para que un proceso ejecute un programa concreto.
- Conocer las funciones y estructuras de datos que me permiten trabajar con los procesos.
- Comprender los conceptos e implementaciones que utiliza UNIX para construir las abstracciones de procesos, la jerarquía de procesos y el entorno de ejecución de éstos.

2. Creación 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 el **`init`** (PID=1). 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 y no es un proceso de sistema sino uno normal aunque se ejecute con privilegios de superusuario (root). El proceso **`init`** es *el proceso raíz de la jerarquía de procesos del sistema*, que se genera debido a las relaciones entre proceso creador (padre) y proceso creado (hijo).

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 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.
```

El siguiente programa utiliza las funciones para obtener el pid de un proceso y el pid de su proceso padre. Pruébalo y observa sus resultados.

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

void main(void)
{
      pid_t id_proceso;
      pid_t id_padre;

      id_proceso = getpid();
      id_padre = getppid();

      printf("Identificador de proceso: %d\n", id_proceso);
      printf("Identificador del proceso padre: %d\n", id_padre);
}
```

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 principal* del usuario. El grupo principal es el que aparece en la línea correspondiente al login del usuario en el archivo de passwords. El siguiente programa obtiene esta información sobre el proceso que las ejecuta. Pruébalo y observa sus resultados.

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

```
#include <stdio.h>
#include <unistd.h>

void main(void)
{
      printf("Identificador de usuario: %d\n", getuid());
      printf("Identificador de usuario efectivo: %d\n", geteuid());
      printf("Identificador de grupo: %d\n", getgid());
      printf("Identificador de grupo efectivo: %d\n", getegid());
}
```

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 el punto 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 un valor distinto en cada uno de los procesos (padre e hijo). La única diferencia entre los valores devueltos es que en el proceso hijo el valor es 0 y en el proceso padre (el 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 hecho de que se devuelva un valor distinto en el proceso padre y en el hijo nos va a ser muy útil, de cara a poder ejecutar distintas partes de código una vez finalizada la llamada **`fork`**. Para esto podremos hacer uso de instrucciones de bifurcación (ver tarea4.c).

Tanto el padre como el hijo continuarán ejecutando la instrucción siguiente al **`fork`** y el hijo será una copia idéntica del padre. Ambos procesos se ejecutarán a partir de este momento de forma concurrente.

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).

Actividad 3.1 *Trabajo con la llamada al sistema* `fork`

Consulta con **`man`** la llamada al sistema **`fork`**.

Ejercicio 1. Implementa un programa en C que tenga como argumento un número entero. Este programa debe crear un proceso hijo que se encargará de comprobar si dicho número es un número par o impar e informará al usuario con un mensaje que se enviará por la salida estándar. A su vez, el proceso padre comprobará si dicho número es divisible por 4, e informará si lo es o no usando igualmente la salida estándar.

Ejercicio 2. ¿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.

```
/*
tarea4.c
Trabajo con llamadas al sistema de Control 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 búfer 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 Linux 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***!

Ejercicio 3. Indica qué tipo de jerarquías de procesos se generan mediante la ejecución de cada uno de los siguientes fragmentos de código. Comprueba tu solución implementando un código para generar 20 procesos en cada caso, en donde cada proceso imprima su PID y el del padre, PPID.

```
/*
Jerarquía de procesos tipo 1
*/

for (i=1; i < nprocs; i++) {
  if ((childpid= fork()) == -1) {
    fprintf(stderr, "Could not create child %d: %s\n",i,strerror(errno));
    exit(-1);
  }

  if (childpid)
    break;
}

/*
Jerarquía de procesos tipo 2
*/

for (i=1; i < nprocs; i++) {
  if ((childpid= fork()) == -1) {
    fprintf(stderr, "Could not create child %d: %s\n",i,strerror(errno));
    exit(-1);
  }

  if (!childpid)
    break;
}
```

Actividad 3.2 Trabajo con las llamadas al sistema `wait`, `waitpid` y `exit`

Consulta en el manual en línea las llamadas **wait**, **waitpid** y **exit** para ver sus posibilidades de sincronización entre el proceso padre y su(s) proceso(s) hijo(s) y realiza los siguientes ejercicios:

Ejercicio 4. 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
```

Ejercicio 5. 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).

3. 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 los 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** actuarán de forma similar al shell si la ruta especificada no es un nombre de camino absoluto o relativo. La lista de búsqueda es la especificada en el entorno por la variable **PATH**. Si esta variable no está especificada, se emplea la ruta predeterminada "**:/bin:/usr/bin**".

- Si a un archivo se le deniega el permiso (**execve** devuelve **EACCES**), estas funciones continuarán buscando en el resto de la lista 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 3.3 Trabajo con la familia de llamadas al sistema `exec`

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.

Ejercicio 6. ¿Qué hace el siguiente programa?

```
/*
tarea5.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","./tarea5",NULL)<0)) {
        perror("\nError en el execl");
```

```
        exit(-1);
    }
}
wait(&estado);
/*
<estado> mantiene información codificada a nivel de bit sobre el motivo de
finalización del proceso hijo que puede ser el número de señal o 0 si alcanzó su
finalización normalmente.
Mediante la variable estado de wait(), el proceso padre recupera el valor
especificado por el proceso hijo como argumento de la llamada exit(), pero
desplazado 1 byte porque el sistema incluye en el byte menos significativo el
código de la señal que puede estar asociada a la terminación del hijo. Por eso se
utiliza estado>>8 de forma que obtenemos el valor del argumento del exit() del
hijo.
*/

printf("\nMi hijo %d ha finalizado con el estado %d\n",pid,estado>>8);
exit(0);

}
```

Ejercicio 7. Escribe un programa que acepte como argumentos el nombre de un programa, sus argumentos si los tiene, y opcionalmente la cadena “bg”. Nuesto programa deberá ejecutar el programa pasado como primer argumento en `foreground` si no se especifica la cadena “bg” y en `background` en caso contrario. Si el programa tiene argumentos hay que ejecutarlo con éstos.

4. La llamada `clone`

A partir de este momento consideramos que **`tid`** es el identificador de hebra de un proceso. Como hemos visto en teoría, en Linux, **`fork()`** se implementa a través de la llamada **`clone()`** que permite crear procesos e hilos con un grado mayor en el control de sus propiedades. La sintaxis para esta función es

```
#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 */ );

                  Retorna: si éxito, el PID del hijo; -1, si error
```

Cuando invocamos a **`clone`**, se crea un nuevo proceso hijo que comienza ejecutando la función dada por **`func`** (no la siguiente instrucción, como ocurre en **`fork`**), a la que se le pasa el argumento indicado en **`func_arg`**. El hijo finaliza cuando la función indicada retorna o cuando haga un **`exit`** o **`_exit`**. También, debemos pasar como argumento un puntero a la pila que debe utilizar el hijo, en el argumento **`child_stack`**, y que previamente hemos reservado.

Uno de los argumentos más interesantes de **`clone()`** son los ***indicadores de clonación*** (argumento **`flags`**). Estos tienen dos usos, el byte de orden menor sirve para especificar la *señal*

de terminación del hijo, que normalmente suele ser **SIGCHLD**. Si esta a cero, no se envía señal. Una diferencia con **fork()**, es que en ésta no podemos seleccionar la señal, que siempre es **SIGCHLD**. Los restantes bytes de **flags** tienen el significado que aparece en **Tabla 1**.

Tabla 1.- *Indicadores de clonación*.

Indicador	Significado
CLONE_CHILD_CLEARTID	Limpia **tid** (***t**hread **id**entifier*) 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 esta 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[9]
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 `exec()`
CLONE_VM	Padre e hijo comparten el espacio de memoria virtual

Vamos a ver un programa ejemplo, para entender como se comporta la creación de procesos con la llamada **clone** dependiendo de los indicadores que utilicemos. No veremos todos por razones de tiempo/espacio.

El **Programa 3** nos muestra el uso de **clone()** en el que hemos utilizado los indicadores **CLONE_VM|CLONE_FILES|CLONE_FS|CLONE_THREAD|CLONE_SIGHAND**. De la Tabla 1, concluimos que el hijo comparte con su padre la memoria virtual, los archivos abiertos, el directorio raíz, el directorio de trabajo y la máscara de creación de archivos, pone al hijo en el mismo grupo del padre, y comparten los manejadores de señales. Es decir, creamos un hilo.

Programa 3.- Ejemplo de `clone()`.

9 El kernel mantiene una lista para deshacer las operaciones de un semáforo cuando un proceso termina sin liberar los semáforos con el objetivo de minimizar la posibilidad de interbloqueo. Podeis ver http://eduunix.ccut.edu.cn/index2/html/linux/Interprocess%20Communications%20in%20Linux/ch07lev1sec2.htm.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <linux/unistd.h>
#include <sys/syscall.h>
#include <errno.h>
#include <linux/sched.h>
#include <malloc.h>

#define _GNU_SOURCE             /* See feature_test_macros(7) */

int variable=3;

int thread(void *p) {
      int tid;

      printf("\nsoy el hijo\n");
      sleep(5);
      variable++;
      tid = syscall(SYS_gettid);
      printf("\nPID y TID del hijo:%d %d\n",getpid(),tid);
      printf("\nEn el hijo la variable vale:%d\n",variable);
}

int main() {

      void **stack;
        int i, tid;

      stack = (void **)malloc(15000);
      if (!stack)
            return -1;

          i = clone(thread, (char*) stack + 15000, CLONE_VM|CLONE_FILES|CLONE_FS|
CLONE_THREAD|CLONE_SIGHAND, NULL);
      sleep(5);
        if (i == -1)
                perror("clone");
      tid = syscall(SYS_gettid);
      printf("\nPID y TID del padre:%d %d\n ",getpid(),tid);
      printf("\nEn el padre la variable vale:%d\n",variable);

return 0;
}
```

En el programa anterior, hemos utilizado la llamada de Linux que citamos anteriormente (no es general de UNIX) **`gettid()`** que devuelve el identificador de una hebra, **`tid`**.

Si ejecutamos este proceso podemos observar como padre e hijo están en el mismo grupo puesto que su PID es igual, tal como establece el indicador **`CLONE_THREAD`**. Además, como hemos

establecido **CLONE_VM**, el espacio de memoria es el mismo, por lo que **variable** es la misma en ambos.

```
$>./clone
PID y TID del hijo: 10549 10550
En el hijo la variable vale 4
PID y TID del padre: 10549 10549
En el padre la variable vale 4
```

Además, si durante la ejecución del programa lo paramos con **<Ctrl-Z>** podemos ver como efectivamente en el sistema hay dos hebras a través de **ps** (**NLWP** es el número de hilos) y como estas comparten la memoria ya que el tamaño de RSS (Resident Set Size – conjunto residente de memoria) es el mimo:

```
UID        PID  PPID   LWP  C NLWP    SZ   RSS PSR STIME TTY          TIME CMD
jagomez   6301  6297  6301  0    1  1364  2584   0 12:02 pts/1    00:00:00 /bin/bash
jagomez  11505  6301 11505  0    2   750   560   1 17:13 pts/1    00:00:00 ./c
jagomez  11505  6301 11506  0    2   750   560   1 17:13 pts/1    00:00:00 ./c
jagomez  11510  6301 11510  0    1   655   880   0 17:13 pts/1    00:00:00 ps -LfF
```

Si modificamos el programa anterior y eliminamos **CLONE_VM**, **CLONE_SIGHAND**, y **CLONE_THREAD**, los procesos padre e hijo no comparte la misma memoria virtual, ni el mismo grupo, como podemos ver si lo compilamos y ejecutamos.

```
$> ./clone
PID y TID del hijo: 10721 10721
En el hijo la variable vale 4
PID y TID del padre: 10720 10720
En el padre la variable vale 3
```

En este caso se trata de dos procesos, cada uno con una hebra y el RSS en diferente:

```
ps -LfF
UID        PID  PPID   LWP  C NLWP    SZ   RSS PSR STIME TTY          TIME CMD
jagomez   6301  6297  6301  0    1  1364  2584   1 12:02 pts/1    00:00:00 /bin/bash
jagomez  11408  6301 11408  0    1   750   804   1 17:09 pts/1    00:00:00 ./c
jagomez  11409 11408 11409  0    1   750   116   0 17:09 pts/1    00:00:00 ./c
jagomez  11413  6301 11413  0    1   655   884   0 17:09 pts/1    00:00:00 ps -LfF
```

Prácticas de Sistemas Operativos

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

Sesión 4. Comunicación entre procesos utilizando cauces

Sesión 4. Comunicación entre procesos utilizando cauces

1. Objetivos principales

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

En particular como objetivos más específicos abordaremos:

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

2. Concepto y tipos de cauce

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

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

La sincronización básica que ocurre se debe a que los datos no pueden ser consumidos mientras no sean enviados al cauce. Así pues, un proceso que intenta leer datos de un cauce se bloquea si actualmente no existen dichos datos, es decir, no se han escrito aún en el cauce por parte de alguno de los procesos productores, o si previamente los ha tomado ya alguno de los otros procesos consumidores. Los cauces proporcionan un método de comunicación entre procesos en un sólo sentido (unidireccional, semi-dúplex), es decir, si deseamos comunicar en el otro sentido es necesario utilizar otro cauce diferente.

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

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

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

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

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

- Los cauces con nombre se crean (llamadas al sistema **`mknod`** y **`mkfifo`**) en el sistema de archivos en disco como un archivo especial, es decir, consta de un nombre que ayuda a denominarlo exactamente igual que a cualquier otro archivo en el sistema de archivos, y por tanto aparecen contenidos/asociados de forma permanente a los directorios donde se crearon.
- Los procesos abren y cierran un archivo FIFO usando su nombre mediante las ya conocidas llamadas al sistema **`open`** y **`close`**, con el fin de hacer uso de él.
- Cualesquiera procesos pueden compartir datos utilizando las ya conocidas llamadas al sistema **`read`** y **`write`** sobre el cauce con nombre previamente abierto. Es decir, los cauces

con nombre permiten comunicar a procesos que no tienen un antecesor común en la jerarquía de procesos de UNIX.

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

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

3. Cauces con nombre

3.1 Creación de archivos FIFO

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2 Utilización de un cauce FIFO

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

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

Actividad 4.1 Trabajo con cauces con nombre

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

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

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

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

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

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

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

return 0;
}

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

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

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

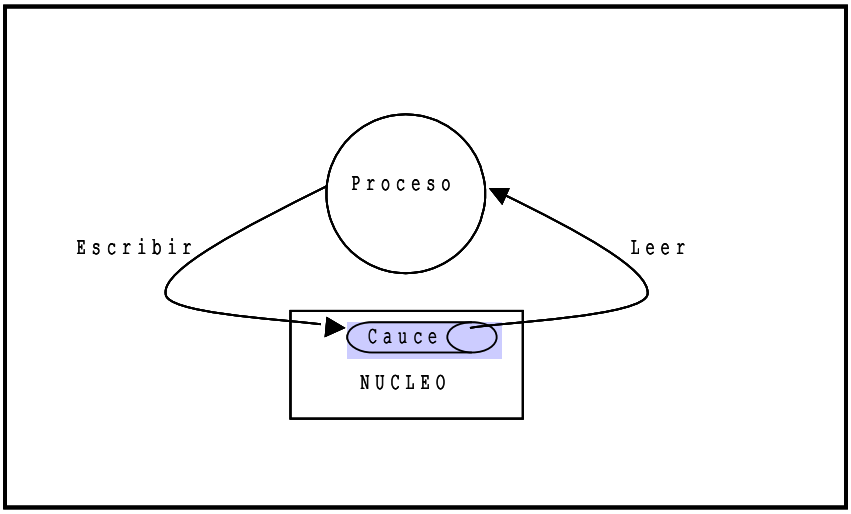
```
{
int fd;

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

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

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

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

4. Cauces sin nombre

4.1 Esquema de funcionamiento

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

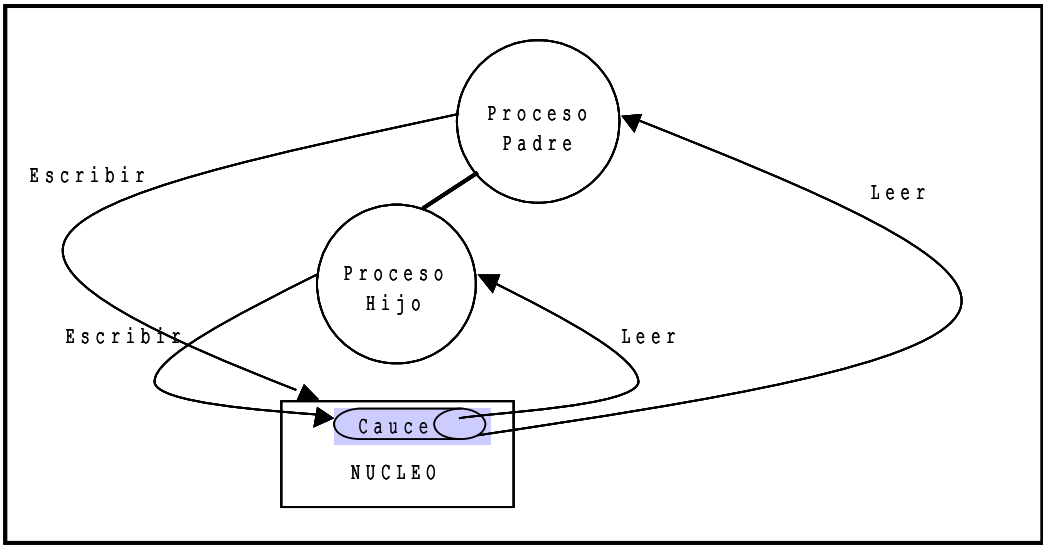
Al ejecutar la llamada al sistema **`pipe`** para crear un cauce sin nombre, el núcleo automáticamente instala dos descriptores de archivo para que los use dicho cauce. Un descriptor se usa para permitir un camino de envío de datos (**`write`**) al cauce, mientras que el otro descriptor se usa para obtener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



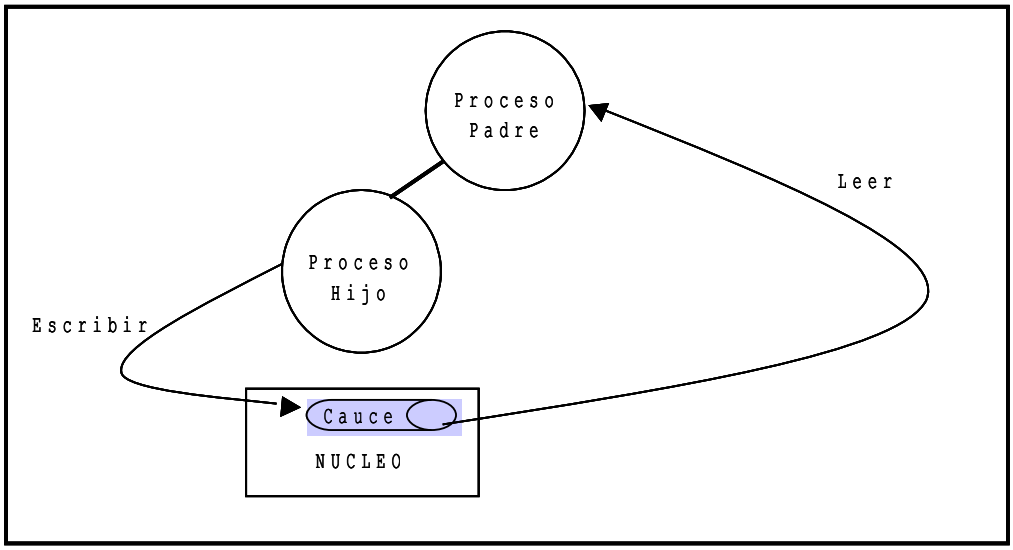
En el esquema anterior se puede observar cómo el proceso puede usar los descriptores de archivo para enviar datos (llamada al sistema **`write`**) al cauce, y leerlos desde éste con la llamada al sistema **`read`**. Sin embargo, este esquema se puede ampliar ya que carece de utilidad práctica; el mecanismo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

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

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



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



4.2 Creación de cauces

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

Una vez creado el cauce, creamos un proceso hijo (que heredará los descriptores de archivos del padre) y establecemos el sentido del flujo de datos (hijo->padre o padre->hijo). Como los descriptores son compartidos por el proceso padre y el hijo, debemos estar seguros siempre de

cerrar con la llamada al sistema **close** el extremo del cauce que no nos interese en cada uno de los procesos, para evitar confusiones que podrían derivar en errores al usar el mecanismo. Si el padre quiere recibir datos del hijo, debe cerrar el descriptor usado para escritura (**fd[1]**) y el hijo debe cerrar el descriptor usado para lectura (**fd[0]**). Si por el contrario el padre quiere enviarle datos al hijo, debe cerrar el descriptor usado para lectura (**fd[0]**) y el hijo debe cerrar el descriptor usado para escritura (**fd[1]**).

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

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

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

4.3 Notas finales sobre cauces con y sin nombre

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

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

- La sincronización entre procesos productores y consumidores es atómica.

Actividad 4.2 Trabajo con cauces sin nombre

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

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

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

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

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

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

return(0);
}
```

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

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

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

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

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

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

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

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

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

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

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

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

return(0);
}
```

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

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

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

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

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

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

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

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

return(0);
}
```

Ejercicio 5.

Este ejercicio se basa en la idea de utilizar varios procesos para realizar partes de una computación en paralelo. Para ello, deberá construir un programa que siga el esquema de computación maestro-esclavo, en el cual existen varios procesos trabajadores (esclavos) idénticos y un único proceso que reparte trabajo y reúne resultados (maestro). Cada esclavo es capaz de realizar una computación que le asigne el maestro y enviar a este último los resultados para que sean mostrados en pantalla por el maestro.

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

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

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

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

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

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

1. Objetivos principales

En esta sesión trabajaremos con las llamadas al sistema relacionadas con la gestión y el control de señales. El control de señales en Linux incluye las llamadas al sistema necesarias para cambiar el comportamiento de un proceso cuando recibe una determinada señal, examinar y cambiar la máscara de señales y el conjunto de señales bloqueadas, y suspender un proceso, así como comunicar procesos.

- Conocer las llamadas al sistema para el control de señales.
- Conocer las funciones y las estructuras de datos que me permiten trabajar con señales.
- Aprender a utilizar las señales como mecanismo de comunicación entre procesos.

2. Señales

Las señales constituyen un mecanismo básico de sincronización que utiliza el núcleo de Linux para indicar a los procesos la ocurrencia de determinados eventos síncronos/asíncronos con su ejecución. Aparte del uso de señales por parte del núcleo, los procesos pueden enviarse señales para la notificación de cierto evento (la señal es generada cuando ocurre este evento) y, lo que es más importante, pueden determinar qué acción realizarán como respuesta a la recepción de una señal determinada.

Un manejador de señal es una función definida en el programa que se invoca cuando se entrega una señal al proceso. La invocación del manejador de la señal puede interrumpir el flujo de control del proceso en cualquier instante. Cuando se entrega la señal, el kernel invoca al manejador, y cuando el manejador retorna, la ejecución del proceso sigue por donde fue interrumpida, como muestra la figura 1.

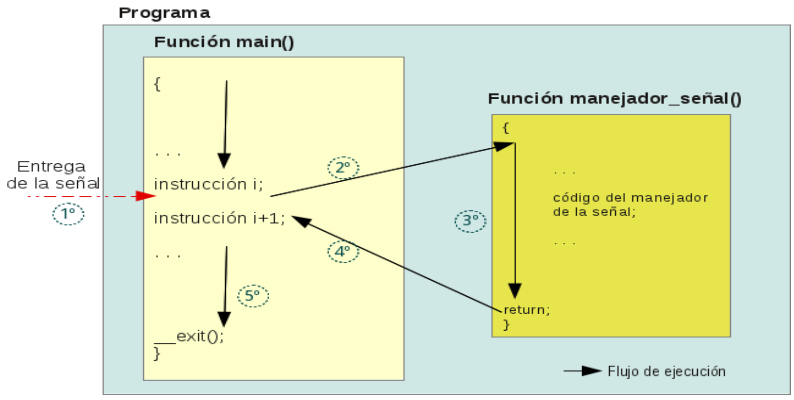


Figura 1. Ejecución del manejador tras la entrega de la señal

Se dice que una señal es *depositada* cuando el proceso inicia una acción en base a ella, y se dice que una señal está *pendiente* si ha sido generada pero todavía no ha sido depositada. Además un proceso puede bloquear la recepción de una o varias señales a la vez.

Las señales bloqueadas de un proceso se almacenan en un conjunto de señales llamado *máscara de bloqueo de señales*. No se debe confundir una señal *bloqueada* con una señal *ignorada*, ya que una señal ignorada es desechada por el proceso, mientras que una señal bloqueada permanece pendiente y será depositada cuando el proceso la desenmascare (la desbloquee). Si una señal es recibida varias veces mientras está bloqueada, se maneja como si se hubiese recibido una sola vez.

La lista de señales y su tratamiento por defecto se puede consultar con **`man 7 signal`** (o en **`signal.h`**). En la tabla 1 se muestran las señales posibles en POSIX.1. Cada señal posee un nombre que comienza por **`SIG`**, mientras que el resto de los caracteres se relacionan con el tipo de evento que representa. Realmente, cada señal lleva asociado un número entero positivo, que es el que se entrega al proceso cuando éste recibe la señal. Se puede usar indistintamente el número o la constante que representa a la señal.

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 (**<Ctrl+C>**)
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 **abort(3)**
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 **alarm(2)**
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

Tabla1: Lista de señales en **POSIX.1**

Las entradas en la columna "Acción" de la tabla anterior especifican la acción por defecto para la señal usando la siguiente nomenclatura:

- *Term* La acción por defecto es terminar el proceso.
- *Ign* La 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 que el proceso continúe su ejecución si está parado.

Las llamadas al sistema que podemos utilizar en Linux para trabajar con señales son principalmente:

- **kill**, se utiliza para enviar una señal a un proceso o conjunto de procesos.
- **sigaction**, permite establecer la acción que realizará un proceso como respuesta a la recepción de una señal. Las únicas señales que no pueden cambiar su acción por defecto son: **SIGKILL** y **SIGSTOP**.
- **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 el argumento `mask` y luego suspende el proceso hasta que se recibe una señal.

Sinopsis

```
#include <signal.h>
int kill(pid_t pid, int sig)
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);
```

2.1 La llamada `kill`

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

Sinopsis

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

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

Argumentos

- 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 `sigaction`

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

Sinopsis

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

Argumentos

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`.

Valor de retorno

`0` en caso de éxito y `-1` en caso de error

Estructuras de datos

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);
                    }
```

- **`sa_handler`** especifica la acción que se va a asociar con la señal *`signum`* pudiendo ser:
  - `SIG_DFL` para la acción predeterminada,
  - `SIG_IGN` para ignorar 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 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`, se usan las siguientes funciones:

- *`int`* ***`sigemptyset`****`(sigset_t *set);`*

  inicializa a vacío un conjunto de señales (devuelve 0 si tiene éxito y –1 en caso contrario).

- *int* ***sigfillset****(sigset_t *set);*

  inicializa un conjunto con todas las señales (devuelve 0 si tiene éxito y –1 en caso contrario).

- *int* ***sigismember****(const sigset_t *set, int senyal);*

  determina si una señal *senyal* pertenece a un conjunto de señales *set*

  (devuelve 1 si la señal se encuentra dentro del conjunto, y 0 en caso contrario).

- int **sigaddset**(sigset_t *set, int signo);

  añade una señal a un conjunto de señales *set* previamente inicializado

  (devuelve 0 si tiene éxito y –1 en caso contrario).

- *int* ***sigdelset****(sigset_t *set, int signo);*

  elimina una señal *signo* de un conjunto de señales *set*

  (devuelve 0 si tiene éxito y –1 en caso contrario).

- **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: 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 que ciertas llamadas al sistema reinicien su ejecución cuando son interrumpidas por la recepción de una señal.

  - SA_NOMASK o SA_NODEFER

    Se pide al núcleo que no impida la recepción de la señal desde el propio manejador de la señal.

- 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 */
}
```

Los posibles valores para cualquier señal se pueden consultar con **`man sigaction`**.

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

Los siguientes ejemplos ilustran el uso de la llamada al sistema **`sigaction`** para establecer un manejador para la señal **`SIGINT`** que se genera cuando se pulsa **`<CTRL+C>`**.

```
// tarea9.c
#include <stdio.h>
#include <signal.h>

int main(){
   struct sigaction sa;
   sa.sa_handler = SIG_IGN; // ignora la señal
   sigemptyset(&sa.sa_mask);

   //Reiniciar las funciones que hayan sido interrumpidas por un manejador
   sa.sa_flags = SA_RESTART;

   if (sigaction(SIGINT, &sa, NULL) == -1){
      printf("error en el manejador");}
      while(1);
 }
```

```
// tarea10.c
#include <stdio.h>
#include <signal.h>


   static int s_recibida=0;
   static void handler (int signum){
   printf("\n Nueva acción del manejador \n");
   s_recibida++;}



int main()
{
   struct sigaction sa;
   sa.sa_handler = handler; // establece el manejador a handler
   sigemptyset(&sa.sa_mask);


   //Reiniciar las funciones que hayan sido interrumpidas por un manejador
   sa.sa_flags = SA_RESTART;
```

```
   if (sigaction(SIGINT, &sa, NULL) == -1){
   printf("error en el manejador");}
   while(s_recibida<3);
 }
```

Actividad 5.1. Trabajo con las llamadas al sistema `sigaction` y `kill`.

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.

Ejercicio 1. Compila y ejecuta los siguientes programas y trata de entender su funcionamiento.

```
/*
 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> //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("\n No puedo enviar ese tipo de señal");

              }

}
```

```
/*
 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(;;)
   {
   }
}
```

Ejercicio 2. Escribe un programa en C llamado **`contador`**, tal que cada vez que reciba una señal que se pueda manejar, muestre por pantalla la señal y el número de veces que se ha recibido ese tipo de señal, y un mensaje inicial indicando las señales que no puede manejar. En el cuadro siguiente se muestra un ejemplo de ejecución del programa.

```
kawtar@kawtar-VirtualBox:~$ ./contador &

[2] 1899

kawtar@kawtar-VirtualBox:~$

No puedo manejar la señal 9

No puedo manejar la señal 19

Esperando el envío de señales...

kill -SIGINT 1899

kawtar@kawtar-VirtualBox:~$ La señal 2 se ha recibido 1 veces

kill -SIGINT 1899

La señal 2 se ha recibido 2 veces

kill -15 1899

kawtar@kawtar-VirtualBox:~$ La señal 15 se ha recibido 1 veces

kill -111 1899

bash: kill: 111: especificación de señal inválida

kawtar@kawtar-VirtualBox:~$ kill -15 1899 // el programa no puede capturar la
señal 15

[2]+ Detenido            ./contador

kawtar@kawtar-VirtualBox:~$ kill -cont 1899

La señal 18 se ha recibido 1 veces
```

```
kawtar@kawtar-VirtualBox:~$ kill -KILL 1899

[2]+  Terminado (killed)    ./contador
```

2.3 La llamada `sigprocmask`

La llamada **`sigprocmask`** se emplea para examinar y cambiar la máscara de señales.

Sinopsis

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

Argumentos

- El argumento `how` indica el tipo de cambio. Los valores que puede tomar son los siguientes:
  - `SIG_BLOCK`: El conjunto de señales bloqueadas es la unión del conjunto actual y el argumento `set`.
  - `SIG_UNBLOCK`: Las señales que hay en `set` se eliminan del conjunto actual de señales bloqueadas. Es posible intentar el desbloqueo de una señal que no está bloqueada.
  - `SIG_SETMASK`: El conjunto de señales bloqueadas se pone según el argumento `set`.

- `set` representa el puntero al nuevo conjunto de señales enmascaradas. Si `set` es diferente de NULL, apunta a un conjunto de señales, en caso contrario `sigprocmask` se utiliza para consulta.
- `oldset` representa el conjunto anterior de señales enmascaradas. Si `oldset` no es `NULL`, el valor anterior de la máscara de señal se guarda en `oldset`. En caso contrario no se retorna la máscara la anterior.

Valor de retorno

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

2.4 La llamada `sigpending`

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

Sinopsis

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

Argumento

`set` representa un puntero al conjunto de señales pendientes

Valor de retorno

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

2.5 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 se recibe una señal.

Sinopsis

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

Argumento

`mask` representa el puntero al nuevo conjunto de señales enmascaradas

Valor de retorno

-1 si `sigsuspend` es interrumpida por una señal capturada (no está definida la terminación correcta)

Ejemplo de uso: En el siguiente ejemplo se suspende la ejecución del proceso actual hasta que reciba una señal distinta de **SIGUSR1**.

```
//tarea11.c


#include <stdio.h>
#include <signal.h>


int main(){
   sigset_t new_mask;


   /* inicializar la nueva mascara de señales */
   sigemptyset(&new_mask);


   sigaddset(&new_mask, SIGUSR1);


   /*esperar a cualquier señal excepto SIGUSR1 */
   sigsuspend(&new_mask);


 }
```

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** (llamada al sistema que permite a un proceso mandarse a sí mismo una señal). La división entera entre 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 conocer el manejador de señal en curso. También puede emplearse para comprobar si una señal dada es válida para la máquina donde 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.

Actividad 5.2. Trabajo con las llamadas al sistema sigsuspend y sigprocmask

Ejercicio 3. Escribe un programa que suspenda la ejecución del proceso actual hasta que se reciba la señal SIGUSR1. Consulta en el manual en línea sigemptyset para conocer las distintas operaciones que permiten configurar el conjunto de señales de un proceso.

Ejercicio 4. Compila y ejecuta el siguiente programa y trata de entender su funcionamiento.

```
//tarea12.c

#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

```
static int signal_recibida = 0;


{
          signal_recibida = 1;
}


int main (int argc, char *argv[])
{
    sigset_t conjunto_mascaras;
    sigset_t conj_mascaras_original;
    struct sigaction act;

    //Iniciamos a 0 todos los elementos de la estructura act
    memset (&act, 0, sizeof(act));

    act.sa_handler = manejador;

    if (sigaction(SIGTERM, &act, 0)) {
        perror ("sigaction");
        return 1;
    }

    //Iniciamos un nuevo conjunto de mascaras
    sigemptyset (&conjunto_mascaras);
    //Añadimos SIGTERM al conjunto de mascaras
    sigaddset (&conjunto_mascaras, SIGTERM);

    //Bloqueamos SIGTERM
     if (sigprocmask(SIG_BLOCK, &conjunto_mascaras, &conj_mascaras_original) < 0)
```

```
{
       perror ("primer sigprocmask");
       return 1;
    }

    sleep (10);

    //Restauramos la señal – desbloqueamos SIGTERM
    if (sigprocmask(SIG_SETMASK, &conj_mascaras_original, NULL) < 0) {
       perror ("segundo sigprocmask");
       return 1;
       }

    sleep (1);

   if (signal_recibida)
       printf ("\nSenal recibida\n");
   return 0;
}
```

Prácticas de Sistemas Operativos

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

Sesión 6. Control de archivos y archivos proyectados en memoria

Sesión 6. Control de archivos y archivos proyectados a memoria

1. Objetivos principales

Esta nueva sesión nos va a permitir:

- Estudiar las órdenes de la función **`fcntl`** que nos permiten cambiar o consultar las banderas de control de acceso de un archivo abierto.
- Manejar la orden de la función **`fcntl`** que nos permite duplicar un descriptor de archivo para implementar redireccionamiento de archivos.
- Manejar las órdenes de la función **`fcntl`** que nos permiten consultar o implementar el bloqueo de una región de un archivo.
- Interfaz de programación para crear y manipular proyecciones de archivos en memoria, y uso de **`mmap()`** como IPC (InterProcess Communication).

2. La función `fcntl`

La llamada al sistema **`fcntl`** (*file control*) es una función multipropósito que, de forma general, permite consultar o ajustar las banderas de control de acceso de un descriptor, es decir, de un archivo abierto. Además, permite realizar la duplicación de descriptores de archivos y bloqueo de un archivo para acceso exclusivo. Tiene como prototipo:

```
#include <unistd.h>

#include <fcntl.h>

int fcntl(int fd, int orden, /* argumento_orden */);

                    Retorna: si Ok, depende de orden; -1 si error.
```

El argumento **`orden`** admite un rango muy diferente de operaciones a realizar sobre el descriptor de archivo que se especifica en **`fd`**. El tercer argumento, que es opcional, va a depender de la orden indicada. A continuación, mostramos las órdenes admitidas y que describiremos en los apartados siguientes:

F_GETFL Retorna las banderas de control de acceso asociadas al descriptor de archivo.

F_SETFL Ajusta o limpia las banderas de acceso que se especifican como tercer argumento.

F_GETFD Devuelve la bandera *close-on-exec*[10] del archivo indicado. Si devuelve un 0, la bandera está desactivada, en caso contrario devuelve un valor distinto de cero. La bandera *close-on-exec* de un archivo recién abierto esta desactivada por defecto.

F_SETFD Activa o desactiva la bandera *close-on-exec* del descriptor especificado. En este caso, el tercer argumento de la función es un valor entero que es 0 para limpiar la bandera, y 1 para activarlo.

F_DUPFD Duplica el descriptor de archivo especificado por **fd** en otro descriptor. El tercer argumento es un valor entero que especifica que el descriptor duplicado debe ser mayor o igual que dicho valor entero. En este caso, el valor devuelto por la llamada es el descriptor de archivo duplicado (**nuevoFD = fcntl(viejoFD, F_DUPFD, inicialFD**).

F_SETLK Establece un cerrojo sobre un archivo. No bloquea si no tiene éxito inmediatamente.

F_SETLKW Establece un cerrojo y bloquea al proceso llamador hasta que se adquiere el cerrojo.

F_GETLK Consulta si existe un bloqueo sobre una región del archivo.

2.1 Banderas de estado de un archivo abierto

Uno de los usos de la función permite recuperar o modificar el modo de acceso y las banderas de estado (las especificadas en **open**) de un archivo abierto. Para recuperar los valores, utilizamos la orden **F_GETFL**:

```
int banderas, ModoAcceso;
banderas=fcntl(fd, F_GETFL);
if (banderas == -1)
      perror("fcntl error");
```

Tras lo cual, podemos comprobar si el archivo fue abierto para escrituras sincronizadas[11] como se indica:

```
if (banderas & O_SYNC)
      printf ("Las escrituras son sincronizadas \n");
```

Comprobar el modo de acceso es algo más complicado ya que las constantes **O_RDONLY** (0), **O_WRONLY** (1) y **O_RDWR** (2) no se corresponden con un único bit de la bandera de estado. Por ello, utilizamos la máscara **O_ACCMODE** y comparamos la igualdad con una de las constantes:

```
ModoAcceso=banderas & O_ACCMODE;
if (ModoAcceso == O_WRONLY || ModoAcceso == O_RDWR)
      printf ("El archivo permite la escritura \n);
```

10 Si la bandera *close-on-exec* está activa en un descriptor, al ejecutar la llamada **exec()** el proceso hijo no heredará este descriptor.

11 Consulte la bandera **O_SYNC** de la llamada **open()**.

Podemos utilizar la orden **F_SETFL** de **fcntl()** para modificar algunas de las banderas de estado del archivo abierto. Estas banderas son **O_APPEND**, **O_NONBLOCK**, **O_NOATIME**, **O_ASYNC**, y **O_DIRECT**. Se ignorará cualquier intento de modificar alguna otra bandera.

El uso de la función **fcntl** para modificar las banderas de estado es útil en los siguientes casos:

- El archivo no fue abierto por el programa llamador, de forma que no tiene control sobre las banderas utilizadas por **open**. Por ejemplo, la entrada, salida y error estándares se abren antes de invocar al programa.
- Se obtuvo el descriptor del archivo a través de una llamada al sistema que no es **open**. Por ejemplo, se obtuvo con **pipe()** o **socket()**.

Para modificar las banderas, primero invocamos a **fcntl** para obtener una copia de la bandera existente. A continuación, modificamos el bit correspondiente, y finalmente hacemos una nueva invocación de **fcntl** para modificarla. Por ejemplo, para habilitar la bandera **O_APPEND** podemos escribir el siguiente código:

```
int bandera;
bandera = fcntl(fd, F_GETFL);
if (bandera == -1)
      perror("fcntl");
bandera |= O_APPEND;
if (fcntl(fd, F_SETFL, bandera) == -1)
      perror("fcntl");
```

2.2 La función `fcntl` utilizada para duplicar descriptores de archivos

La orden **F_DUPFD** de **fcntl** permite duplicar un descriptor, es decir, que cuando tiene éxito, tendremos en nuestro proceso dos descriptores apuntando al mismo archivo abierto con el mismo modo de acceso y compartiendo el mismo puntero de lectura-escritura, es decir, compartiendo la misma sesión de trabajo, como vimos en una sesión anterior (ordenes **dup** y **dup2**).

Veamos un fragmento de código que nos permite redireccionar la salida estándar de un proceso hacia un archivo (tal como hacíamos con **dup**, **dup2** o **dup3**):

```
int fd = open ("temporal", O_WRONLY);
close (1);
if (fcntl(fd, F_DUPFD, 1) == -1 ) perror ("Fallo en fcntl");
char bufer[256];
int cont = write (1, bufer, 256);
```

La primera línea abre el archivo al que queremos redireccionar la salida estándar, en nuestro ejemplo, temporal. A continuación, cerramos la salida estándar asignada al proceso llamador. De esta forma, nos aseguramos que el descriptor donde se va a duplicar está libre. Ya podemos realizar la duplicación de **fd** en el descriptor 1. Recordad que el tercer argumento de **fcntl** especifica que el descriptor duplicado es mayor o igual que el valor especificado. En nuestro ejemplo, como hemos realizado un **close(1)**, el descriptor duplicado es el 1. Tras definir el búfer de escritura, realizamos una operación **write(1, …)** que escribe en el archivo temporal, ya que este descriptor apunta ahora al archivo abierto por **open**. Podéis esbozar cómo se redireccionaría la entrada estándar.

Actividad 6.1 Trabajo con la llamada al sistema `fcntl`

Ejercicio 1. Implementa un programa que admita t argumentos. El primer argumento será una orden de Linux; el segundo, uno de los siguientes caracteres "<" o ">", y el tercero el nombre de un archivo (que puede existir o no). El programa ejecutará la orden que se especifica como argumento primero e implementará la redirección especificada por el segundo argumento hacia el archivo indicado en el tercer argumento. Por ejemplo, si deseamos redireccionar la entrada estándar de **`sort`** desde un archivo **`temporal`**, ejecutaríamos:

```
$> ./mi_programa sort "<" temporal
```

Nota. El carácter redirección (**<**) aparece entre comillas dobles para que no los interprete el shell sino que sea aceptado como un argumento del programa **`mi_programa`**.

Ejercicio 2. Reescribir el programa que implemente un encauzamiento de dos órdenes pero utilizando **`fcntl`**. Este programa admitirá tres argumentos. El primer argumento y el tercero serán dos órdenes de Linux. El segundo argumento será el carácter "|". El programa deberá ahora hacer la redirección de la salida de la orden indicada por el primer argumento hacia el cauce, y redireccionar la entrada estándar de la segunda orden desde el cauce. Por ejemplo, para simular el encauzamiento **`ls|sort`**, ejecutaríamos nuestro programa como:

```
$> ./mi_programa2  ls "|" sort
```

2.3 La función `fcntl()` y el bloqueo de archivos

Es evidente que el acceso de varios procesos a un archivo para leer/escribir puede producir condiciones de carrera. Para evitarlas debemos sincronizar las acciones de éstos. Si bien podríamos pensar en utilizar semáforos, el uso de cerrojos de archivos es más corriente debido a que el kernel asocia automáticamente los cerrojos con archivos. Tenemos dos APIs para manejar cerrojos de archivos:

3. **`flock()`** que utiliza un cerrojo para bloquear el archivo completo.

4. **`fcntl()`** que utiliza cerrojos para bloquear regiones de un archivo.

El método general para utilizarlas tiene los siguientes pasos:

1. Posicionar un cerrojo sobre el archivo.
2. Realizar las entradas/salidas sobre el archivo.
3. Desbloquear el archivo de forma que otro proceso pueda bloquearlo.

Si bien, el bloqueo de archivos se utiliza en conjunción con E/S de archivos, se puede usar como técnica general de sincronización. Por ejemplo, varios procesos cooperantes que bloquean un archivo para indicar que un proceso está accediendo a un recurso compartido, como una región de memoria compartida, que no tiene por qué ser el propio archivo.

Como consideración inicial y dado que la biblioteca **`stdio`** utiliza búfering en espacio de usuario, debemos tener cuidado cuando utilizamos funciones de *stdio* con las técnicas que vamos a describir a continuación. El problema proviene de que el búfer de entrada puede llenarse antes de situar un cerrojo, o un búfer de salida puede limpiarse después de eliminar un cerrojo. Algunos medios para evitar estos problemas podemos:

- Realizar las E/S utilizando **`read()`** y **`write()`** y llamadas relacionadas en lugar de utilizar la biblioteca **`stdio`**.

- Limpiar el flujo (stream) **stdio** inmediatamente después de situar un cerrojo sobre un archivo, y limpiarlo una vez más inmediatamente antes de liberar el cerrojo.

- Si bien a coste de eficiencia, deshabilitar el búfering de **stdio** con **setbuf()** o similar.

Debemos distinguir dos tipos de bloqueo de archivos: consultivo y obligatorio. Por defecto, el tipo de bloqueo es **bloqueo consultivo**, esto significa que un proceso puede ignorar un cerrojo situado por otro proceso. Para que el bloqueo consultivo funcione, cada proceso que accede a un archivo debe cooperar situando un cerrojo antes de realizar una operación de E/S. Por contra, en un **bloqueo obligatorio** se fuerza a que un proceso que realiza E/S respete el cerrojo impuesto por otro proceso.

2.3.1 Bloqueo de registros con `fcntl`

Como hemos indicado, **fcntl()** puede situar un cerrojo en cualquier parte de un archivo, desde un único byte hasta el archivo completo. Esta forma de bloqueo se denomina normalmente bloqueo de registros[12].

Para utilizar **fcntl** en el bloqueo de archivos, debemos usar una estructura **flock** que define el cerrojo que deseamos adquirir o liberar. Se define como sigue:

```
struct flock {
      short l_type; /* Tipo de cerrojo: F_RDLCK, F_WRLCK, F_UNLCK */
      short l_whence; /*Interpretar l_start: SEEK_SET,SEEK_CURR,SEEK_END*/
      off_t l_start; /* Desplazamiento donde se inicia el bloqueo */
      off_t l_len; /* Numero bytes bloqueados: 0 significa "hasta EOF" */
      pid_t l_pid; /* Proceso que previene nuestro bloqueo(solo F_GETLK)*/
};
```

El elemento **l_type** indica el tipo de bloqueo que queremos utilizar y puede tomar uno de los siguientes valores: **F_RDLCK** para un cerrojo de lectura, **F_WRLCK** para un cerrojo de escritura, y **F_UNLCK** que elimina un cerrojo.

A la vista de lo indicado, si fd es el descriptor de un archivo previamente abierto donde queremos situar el bloqueo, la forma general de utilizar **fcntl** para el bloqueo tiene el aspecto siguiente:

```
struct flock mi_bloqueo;

. . . /* ajustar campos de mi_bloqueo para describir el cerrojo a usar */

fcntl(fd, orden, &mi_bloqueo);
```

La Figura 1 muestra cómo podemos utilizar el bloqueo de archivos para sincronizar el acceso de dos procesos a la misma región de un archivo. En esta figura, asumimos que todas las peticiones a los cerrojos son bloqueantes, de forma que la espera fallará si el cerrojo está cogido por otro proceso.

---

12 Si bien, el nombre es técnicamente incorrecto, ya que en sistemas tipo Unix los archivos son un flujo de bytes sin estructura de registros. Cualquier noción de registro dentro de un archivo es definida completamente por la aplicación que lo usa. En realidad, sería más correcto utilizar los términos rango de bytes, región de archivo o segmento de archivo.

Si queremos situar un cerrojo de lectura en un archivo, el archivo debe abrirse en modo lectura. De forma similar, si el cerrojo es de escritura, el archivo se abrirá en modo escritura. Para ambos tipos de cerrojos, el archivo debe abrirse en modo lectura-escritura (**O_RDWR**). Si usamos un cerrojo que sea incompatible con el modo de apertura del archivo se producirá un error **EBADF**.

El conjunto de campos **l_whence**, **l_start** y **l_len** especifica el rango de bytes que deseamos bloquear. Los primeros dos valores son similares a los campos **whence** y **offset** de **lseek()**, es decir, **l_start** especifica un desplazamiento dentro del archivo relativo a valor **l_whence** (**SEEK_SET** para el inicio del archivo, **SEEK_CUR** para la posición actual y **SEEK_END** para el final del archivo). El valor de **l_start** puede ser negativo siempre que la posición definida no caiga por delante del inicio del archivo.

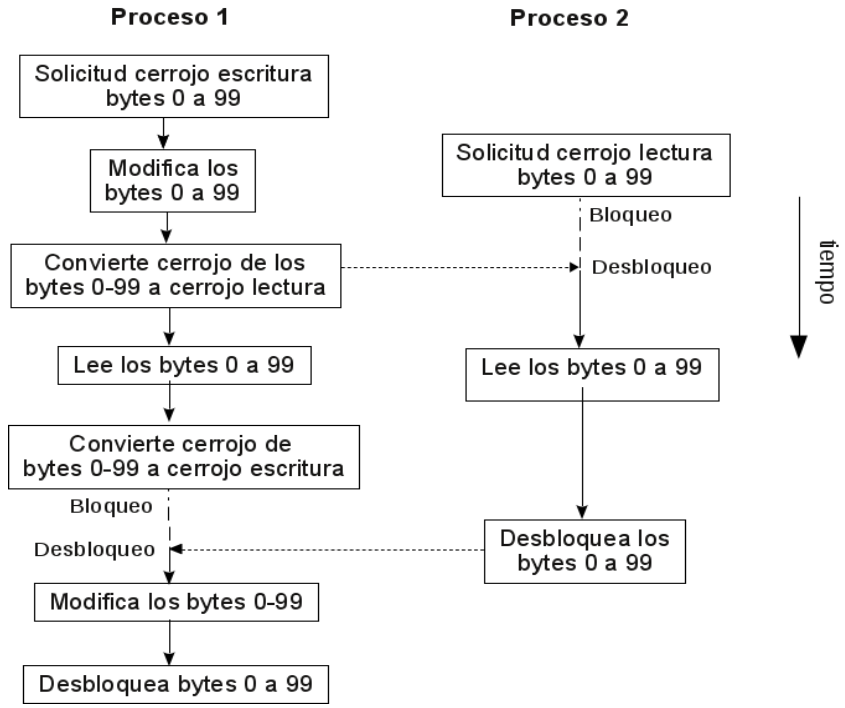


Figura 1.-Uso del bloqueo de registros para sincronizar el acceso a un archivo.

El campo **l_len** es un entero que especifica el número de bytes a bloquear a partir de la posición definida por los otros dos campos. Es posible bloquear bytes no existentes posteriores al fin de archivo, pero no es posible bloquearlos antes del inicio del archivo. A partir del kernel de Linux 2.4.21 es posible especificar un valor de **l_len** negativo. Esta solicitud se aplica al rango **(l_start – abs(l_len), l_start -1)**.

De forma general, lo adecuado sería bloquear el rango mínimo de bytes que necesitemos. Esto permite más concurrencia entre todos los procesos que desean bloquear diferentes regiones de un mismo archivo.

El valor 0 de **l_len** tiene un significado especial “bloquear todos los bytes del archivo, desde la posición especificada por **l_whence** y **l_start** hasta el fin de archivo sin importar cuanto crezca el archivo”. Esto es conveniente si no conocemos de antemano cuantos bytes vamos a añadir al archivo. Para bloquear el archivo completo, podemos especificar **l_whence** como **SEEK_SET** y **l_start** y **l_len** a 0.

Como vimos anteriormente, son tres las órdenes que admite **fcntl()** relativas al bloqueo de archivo. En este apartado, vamos a detallar cada una de ellas:

F_SETLK Adquiere (**l_type** es **F_RDLCK** o **F_WRLCK**) o libera (**l_type** es **F_UNLCK**) un cerrojo sobre los bytes especificados por **flockstr**. Si hay un proceso que tiene un cerrojo incompatible sobre alguna parte de la región a bloquear, la llamada **fcntl** falla con el error **EAGAIN**.

F_SETLKW Igual que la anterior, excepto que si otro proceso mantiene un cerrojo incompatible sobre una parte de la región a bloquear, el llamador se bloqueará hasta que su bloqueo sobre el cerrojo tenga éxito. Si estamos manejando señales y no hemos especificado **SA_RESTART**, una operación **F_SETLKW** puede verse interrumpida, es decir, falla con error **EINTR**. Esto se puede utilizar para establecer un temporizador asociado a la solicitud de bloquea a través de **alarm()** o **setitimer()**.

F_GETLK Comprueba si es posible adquirir un cerrojo especificado en **flockstr**, pero realmente no lo adquiere. El campo **l_type** debe ser **F_RDLCK** o **F_WRLCK**. En este caso la estructura **flockstr** se trata como valor-resultado. Al retornar de la llamada, la estructura contiene información sobre si podemos establecer o no el bloqueo. Si el bloqueo se permite, el campo **l_type** contiene **F_UNLCK**, y los restantes campos no se tocan. Si hay uno o más bloqueos incompatibles sobre la región, la estructura retorna información sobre uno de los bloqueos, sin determinar cual, incluyendo su tipo (**l_type**), y rango de bytes (**l_start**, **l_len**; **l_whence** siempre se retorna como **SEEK_SET**) y el identificador del proceso que lo tiene (**l_pid**).

Existe una condición de carrera potencial al combinar **F_GETLK** con un posterior **F_SETLK** o **F_SETLKW**, ya que cuando realicemos una de estas dos últimas operaciones, la información devuelta por **F_GETLK** puede estar obsoleta. Por tanto, **F_GETLK** es menos útil de lo que parece a primera vista. Incluso si nos indica que es posible bloquear un archivo, debemos estar preparados para que **fcntl** retorne con error al hacer un **F_SETLK** o **F_SETLKW**.

Tenemos que observar los siguientes puntos en la adquisición/liberación de un cerrojo:

- Desbloquear una región siempre tiene éxito. No se considera error desbloquear una región en la cual no tenemos actualmente un cerrojo.

- En cualquier instante, un proceso solo puede tener un tipo de cerrojo sobre una región concreta de un archivo. Situar un nuevo cerrojo en una región que tenemos bloqueada no modifica nada si el cerrojo es del mismo tipo del que teníamos, o se cambia automáticamente el cerrojo actual al nuevo modo. En este último caso, cuando convertimos un cerrojo de lectura en uno de escritura, debemos contemplar la posibilidad de que la llamada devuelva error (**F_SETLK**) o bloquee (**F_SETLKW**).

- Un proceso nunca puede bloquearse el mismo en una región de archivo, incluso aunque sitúe cerrojos mediante múltiples descriptores del mismo archivo.

- Situar un cerrojo de modo diferente en medio de un cerrojo que ya tenemos produce tres cerrojos: dos cerrojos más pequeños en el modo anterior, uno a cada lado del nuevo cerrojo (ver Figura 2). De la misma forma, adquirir un segundo cerrojo adyacente o que solape con un cerrojo existente del mismo modo produce un único cerrojo que cubre el área combinada de los dos cerrojos. Se permiten otras permutaciones. Por ejemplo, desbloquear una región situada dentro de otra región mayor existente deja dos regiones más pequeñas alrededor de la región desbloqueada. Si un nuevo cerrojo solapa un cerrojo existente con un modo diferente, el cerrojo existente es encogido, debido a que los bytes que se solapan se incorporan al nuevo cerrojo.

- Cerrar un descriptor de archivo tiene una semántica inusual respecto del bloqueo de archivos, que veremos en breve

Otro aspecto a tener en cuenta, es el interbloqueo entre procesos que se puede producir cuando utilizamos **`F_SETLKW`**. El escenario es el típico, dos procesos intentan bloquear una región previamente bloqueada por el otro. Para evitar esta posibilidad, el kernel comprueba en cada nueva solicitud de bloqueo realizada con **`F_SETLKW`** si puede dar lugar a un interbloqueo. Si fuese así, el kernel selecciona uno de los procesos bloqueados y provoca que su llamada **`fcntl()`** falle con el error **`EDEADLK`**. En los kernel de Linux actuales, se selecciona al último proceso que hizo la llamada, pero no tiene porque ser así en futuras versiones, ni en otros sistemas Unix. Por tanto, cualquier proceso que utilice **`F_SETLKW`** debe estar preparado para manejar el error **`EDEADLK`**. Las situaciones de interbloqueo son detectadas incluso cuando bloqueamos múltiples archivos diferentes, como ocurre en un interbloqueo circular que involucra a varios procesos.

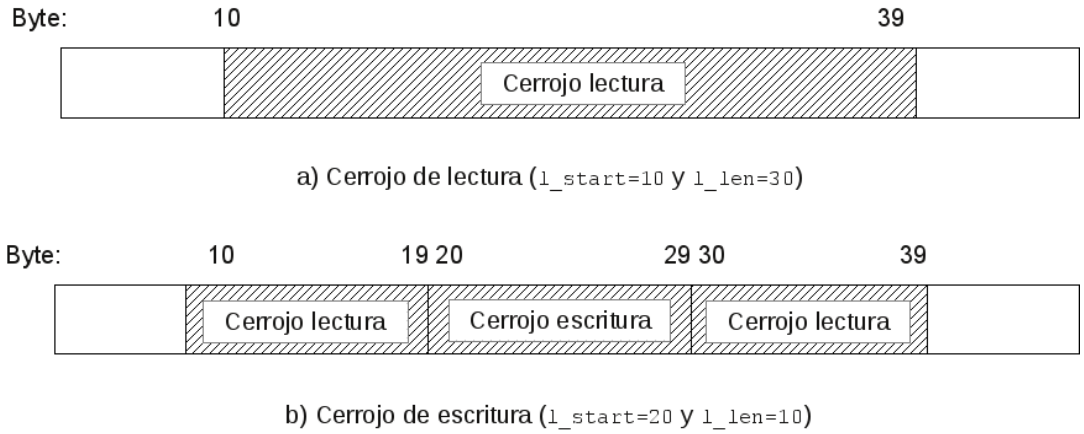


Figura 2.- Partiendo un cerrojo de lectura mediante un cerrojo de escritura.

Respecto a la inanición que puede producirse cuando un proceso intente realizar un cerrojo de escritura cuando otros procesos intentan establecer cerrojos de lectura, en Linux debemos indicar que se siguen las siguientes reglas (que no tienen que cumplirse en otros sistemas Unix):

- El orden en el que se sirven las solicitudes de cerrojos no esta determinado. Si varios procesos esperan para obtener un cerrojo, el orden en el que se satisfacen las peticiones depende de como se planifican los procesos.

- Los escritores no tienen prioridad sobre los lectores, ni viceversa.

El programa **Programa 1** toma uno o varios pathnames como argumento y, para cada uno de los archivos especificados, se intenta un cerrojo consultivo del archivo completo. Si el bloqueo falla, el programa escanea el archivo para mostrar la información sobre los cerrojos existentes: el nombre del archivo, el PID del proceso que tiene bloqueada la región, el inicio y longitud de la región bloqueada, y si es exclusivo (“w”) o compartido (“r”). El programa itera hasta obtener el cerrojo, momento en el cual, se procesaría el archivo (esto se ha omitido) y, finalmente, se libera el cerrojo.

Programa 1. Bloqueo de múltiples archivos.

```
// tarea13.c
```

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

int main(int argc, char* argv[])
{
       struct flock cerrojo;
       int fd;
       while (--argc >0 ) {
              if ((fd=open(*++argv, O_RDWR)) == -1) {
              perror("open fallo"); continue;
       }
       cerrojo.l_type =F_WRLCK;
       cerrojo.l_whence =SEEK_SET;
       cerrojo.l_start =0;
       cerrojo.l_len =0;
       /* intentamos un bloqueo de escritura del archivo completo */

       while (fcntl(fd, F_SETLK, &cerrojo) == -1) {
       /*si el cerrojo falla, vemos quien lo bloquea*/
              while(fcntl(fd, F_GETLK, &cerrojo) != -1 && cerrojo.l_type != F_UNLCK ) {
                     printf("%s bloqueado por %d desde %d hasta %d para %c", *argv,
                     cerrojo.l_pid, cerrojo.l_start, cerrojo.l_len,
                     cerrojo.l_type==F_WRLCK ? 'w':'r'));
                     if (!cerrojo.l_len) break;
              cerrojo.l_start +=cerrojo.l_len;
              cerrojo.l_len=0;
              } /*mientras existan cerrojos de otros procesos*/
       } /*mientras el bloqueo no tenga exito */

       /* Ahora el bloqueo tiene exito y podemos procesar el archivo */
       . . .
       /* Una vez finalizado el trabajo, desbloqueamos el archivo entero */

       cerrojo.l_type =F_UNLCK;
       cerrojo.l_whence =SEEK_SET;
       cerrojo.l_start =0;
       cerrojo.l_len =0;
       if (fcntl(fd, F_SETLKW, &cerrojo) == -1) perror("Desbloqueo");
       }
       return 0;
}
```

En sistemas Linux, no existe límite teórico al número de cerrojos de archivos. El limite viene impuesto por la cantidad de memoria disponible.

Respecto a la eficiencia de adquirir/liberar un cerrojo, indicar que no hay una respuesta fija ya que depende de las estructuras de datos del kernel utilizadas para mantener el registro de cerrojos y la localización de un cerrojo particular dentro de la estructura de datos. Si suponemos un gran número de cerrojos distribuidos aleatoriamente entre muchos procesos, podemos decir que el tiempo necesario para añadir o eliminar un cerrojo crece aproximadamente de forma lineal con el numero de cerrojos ya utilizados sobre un archivo.

Al manejar el bloqueo de registros con **`fcntl()`** debemos tener en cuenta las siguientes consideraciones relativas a la semántica de herencia y liberación:

- Los cerrojos de registros no son heredados con **`fork()`** por el hijo.

- Los bloqueos se mantienen a través de **`exec()`**.

- Todos los hilos de una tarea comparten los mismos bloqueos.

– Los cerrojos de registros están asociados tanto a procesos como a inodos. Una consecuencia esperada de esta asociación es que cuando un proceso termina, todos los cerrojos que poseía son liberados. Menos esperado es que cuando un proceso cierra un descriptor, se liberan todos los cerrojos que ese proceso tuviese sobre ese archivo, sin importar el descriptor sobre el que se obtuvo el cerrojo ni como se obtuvo el descriptor (**dup**, o **fcntl**).

Los elementos vistos hasta ahora sobre cerrojos se refieren a bloqueos consultivos. Como comentamos, esto significa que un proceso es libre de ignorar el uso de **fcntl()** y realizar directamente la operación de E/S sobre el archivo. El kernel no lo evitará. Cuando usamos bloqueo consultivo se deja al diseñador de la aplicación que:

- Ajuste la propiedad y permisos adecuados sobre el archivo, para evitar que los procesos no cooperantes realicen operaciones de E/S sobre él.

- Se asegure que los procesos que componen la aplicación cooperan para obtener el cerrojo apropiado sobre el archivo antes de realizar las E/S.

Linux, como otros Unix, permite que el bloqueo obligatorio de archivos con **fcntl()**. Es decir, cada operación de E/S sobre el archivo se comprueba para ver si es compatible con cualquier cerrojo que posean otros procesos sobre la región del archivo que deseamos manipular.

Para usar bloqueo obligatorio, debemos habilitarlo en el sistema de archivos que contiene a los archivos que deseamos bloquear y en cada archivo que vaya a ser bloqueado. En Linux, habilitamos el bloqueo obligatorio sobre un sistema de archivos montándolo con la opción **-o mand**:

```
$> mount -o mand /dev/sda1 /pruebafs
```

Desde un programa, podemos obtener el mismo resultado especificando la bandera **MS_MANDLOCK** cuando invocamos la llamada **mount(2)**.

El bloqueo obligatorio sobre un archivo se habilita combinado la activación del bit *setgroupid* y desactivando el bit *group-execute*. Esta combinación de bits de permisos no tiene otro uso y por eso se le asigna este significado, lo que permite no cambiar los programas ni añadir nuevas llamadas. Desde el shell, podemos habilitar el bloqueo obligatorio como sigue:

```
$> chmod g+s,g-x /pruebafs/archivo
```

Desde un programa, podemos habilitarlo ajustando los permisos adecuadamente con **chmod()** o **fchmod()**. Así, cuando mostramos los permisos de un archivo que permite el bloqueo obligatorio, veremos una S en el permiso ejecución-de-grupo:

```
%> ls -l /pruebafs/archivo
-rw-r-Sr-- 1 jagomez jagomes 0 12 Dec 14:00 /pruebafs/archivo
```

El bloqueo obligatorio es soportado por todos los sistemas de archivos nativos Linux y Unix, pero puede no ser soportado en sistemas de archivos de red o sistemas de archivos no-Unix. Por ejemplo, VFAT no soporta el bloqueo obligatorio al no disponer de bit de permiso set-group-ID.

La cuestión que se plantea ahora es qué ocurre cuando tenemos activado el bloqueo obligatorio y al realizar una operación de E/S encontramos un conflicto de cerrojos como, por ejemplo, intentar escribir en una región que tiene actualmente un cerrojo de lectura o escritura, o intentar leer de una región que tiene un cerrojo de lectura. La respuesta depende de si el archivo esta abierto de forma bloqueante o no bloqueante. Si el archivo se abrió en modo bloqueo, la llamada al sistema bloqueará al proceso. Si el archivo se abrió con la bandera **O_NONBLOCK**, la llamada fallará inmediatamente con el error **EAGAIN**. Reglas similares se aplican

para **truncate()** y **ftruncate()**, si intentamos añadir o eliminar en una región que solapa con la región bloqueada. Si hemos abierto el archivo en modo bloqueo, no hemos especificado **O_NONBLOCK**, las llamadas de lectura o escritura pueden provocar una situación de interbloqueo. En una situación como la descrita antes, el kernel resuelve la situación aplicando el mismo criterio, selecciona a uno de los procesos involucrados en el interbloqueo y provoca que la llamada al sistema **write()** falle con el error **EDEADLK**.

Los intentos de abrir un archivo con la bandera **O_TRUNC** fallan siempre de forma inmediata, con error **EAGAIN**, si los otros procesos tienen un cerrojo de lectura o escritura sobre cualquier parte del archivo.

Los cerrojos obligatorios hacen menos por nosotros de lo que cabría esperar, y tienen algunas deficiencias:

- Mantener un cerrojo sobre un archivo no evita que otro proceso pueda borrarlo, ya que solo necesita los permisos necesarios para eliminar el enlace del directorio.

- Para habilitar cerrojos obligatorios en un archivo públicamente accesible debemos tener cierto cuidado ya que incluso procesos privilegiados no podrían sobrepasar el cerrojo. Un usuario malintencionado podría obtener continuamente un cerrojo sobre el archivo de forma que provoque un ataque de denegación de servicio.

- Existe un coste de rendimiento asociado a los cerrojos obligatorios debido a que el kernel debe comprobar en cada acceso los posibles conflictos. Si el archivo tiene numerosos cerrojos la penalización puede ser significativa.

- Los cerrojos obligatorios también incurren en un coste en el diseño de la aplicación debido a la necesidad de comprobar si cada llamada de E/S produce un error **EAGAIN** para operaciones no bloqueantes, o **EDEADLK** para operaciones bloqueantes.

En resumen, debemos evitar cerrojos obligatorios salvo que sean estrictamente necesarios.

2.3.2 El archivo `/proc/locks`

En Linux, podemos ver los cerrojos actualmente en uso en el sistema examinando el archivo específico de /proc/locks. Un ejemplo de su contenido se puede ver a continuación:

```
jose@linux:~> cat /proc/locks
1: POSIX ADVISORY READ 8581 08:08:2100091 128 128
2: POSIX ADVISORY READ 8581 08:08:2097547 1073741826 1073742335
3: POSIX ADVISORY READ 8581 08:08:2100089 128 128
4: POSIX ADVISORY READ 8581 08:08:2097538 1073741826 1073742335
5: POSIX ADVISORY WRITE 8581 08:08:2097524 0 EOF
6: POSIX ADVISORY WRITE 3937 08:08:2359475 0 EOF
. . .
```

Este archivo contiene información de los cerrojos creados tanto por **flock()** como por **fcntl()**. Cada línea muestra información de un cerrojo y tiene ocho campos que indican:

- Número ordinal del cerrojo dentro del conjunto de todos los cerrojo del archivo.

- Tipo de cerrojo, donde POSIX indica que el cerrojo se creó con **`fcntl()`** y **`FLOCK`** el que se creó con **`flock()`**.

- Modo del cerrojo, bien consultivo (**`ADVISORY`**) bien obligatorio (**`MANDATORY`**).

- El tipo de cerrojo, ya sea **`WRITE`** o **`READ`** (correspondiente a cerrojos compartidos o exclusivos por **`fcntl()`**).

- El **`PID`** del proceso que mantiene el cerrojo.

- Tres números separados por ":" que identifican el archivo sobre el que se mantiene el cerrojo: el número principal y secundario del dispositivo donde reside el sistema de archivos que contiene el archivo, seguido del número de inodo del archivo.

- El byte de inicio del bloqueo. Para **`flock()`**, siempre es 0.

- El byte final del bloqueo. Donde **`EOF`** indica que el cerrojo se extiende hasta el final del archivo. Para **`flock()`** esta columna siempre es **`EOF`**.

2.3.3 Ejecutar una única instancia de un programa

Algunos programas y, en especial, algunos demonios, deben asegurarse de que solo se ejecuta una instancia del mismo en un instante dado. El método común es que el programa cree un archivo en un directorio estándar y establezca un cerrojo de escritura sobre él.

El programa tiene el bloqueo del archivo durante toda su ejecución y lo libera justo antes de terminar. Si se inicia otra instancia del programa, fallará al obtener el cerrojo de escritura. Por tanto, supone que existe ya otra instancia del programa ejecutándose y termina. Una ubicación usual para tales archivos de bloqueos es /var/run. Alternativamente, la ubicación del archivo puede especificarse en una línea en el archivo de configuración del programa.

Por convención, los demonios escriben su propio identificador en el archivo de bloqueo, y en ocasiones el archivo se nombra con la extensión "**`.pid`**". Por ejemplo, **`syslogd`** crea un archivo **`/var/run/syslogd.pid`**. Esto es útil en algunas aplicaciones para poder encontrar el PID del demonio.

También permite hacer comprobaciones extras de validez, ya que podemos comprobar que el proceso con ese indicador existe utilizando **`kill(pid,0)`**.

Actividad 6.2: Bloqueo de archivos con la llamada al sistema `fcntl`

Ejercicio 3. Construir un programa que verifique que, efectivamente, el kernel comprueba que puede darse una situación de interbloqueo en el bloqueo de archivos.

Ejercicio 4. Construir un programa que se asegure que solo hay una instancia de él en ejecución en un momento dado. El programa, una vez que ha establecido el mecanismo para asegurar que solo una instancia se ejecuta, entrará en un bucle infinito que nos permitirá comprobar que no podemos lanzar más ejecuciones del mismo. En la construcción del mismo, deberemos asegurarnos de que el archivo a bloquear no contiene inicialmente nada escrito en una ejecución anterior que pudo quedar por una caída del sistema.

3. Archivos proyectados en memoria con `mmap`

Un archivo proyectado en memoria es una técnica que utilizan los sistemas operativos actuales para acceder a archivos. En lugar de una lectura "tradicional" lo que se hace es crear una nueva región de memoria en el espacio de direcciones del proceso del tamaño de la zona a acceder del archivo (una parte o todo el archivo) y cargar en ella el contenido de esa parte del archivo. Las páginas de la proyección son (automáticamente) cargadas del archivo cuando sean necesarias.

La función **`mmap()`** proyecta bien un archivo bien un objeto memoria compartida en el espacio de direcciones del proceso. Podemos usarla para tres propósitos:

- Con un archivo regular para suministrar E/S proyectadas en memoria (este Apartado).
- Con archivo especiales para suministrar proyecciones anónimas (Apartado 3.2).
- Con **`shm_open`** para compartir memoria entre procesos no relacionados (no lo veremos dado que no hemos visto segmentos de memoria compartida, que es uno de los mecanismos denominados *System V* IPC).

```
#include <sys/mman.h>

void  *mmap(void *address, size_t length, int prot, int flags,

                  int fd, off_t offset);

      Retorna: dirección inicial de la proyección,si OK; MAP_FAILED, si error.
```

El primer argumento de la función, **`address`**, especifica la dirección de inicio dentro del proceso donde debe proyectarse (mapearse) el descriptor. Normalmente, especificaremos un nulo que le indica al kernel que elija la dirección de inicio. En este caso, la función retorna como valor la dirección de inicio asignada. El argumento **`len`** es el número de bytes a proyectar, empezando con un desplazamiento desde el inicio del archivo dado por **`offset`**. Normalmente, **`offset`** es 0. La Figura 3 muestra la proyección. El argumento **`fd`** indica el descriptor del archivo a proyectar, y que una vez creada la proyección, podemos cerrar.

Dado que la página es la unidad mínima de gestión de memoria, la llamada **`mmap()`** opera sobre páginas: el área proyectada es múltiplo del tamaño de página. Por tanto, **`address`** y **`offset`** deberían estar alineados a limite de páginas, es decir, deberían ser enteros múltiples del tamaño de página. Si el parámetro **`len`** no está alineado a página (porque, por ejemplo, el archivo subyacente no tiene un tamaño múltiplo de página), la proyección se redondea por exceso hasta completar última página. Los bytes añadidos para completar la página se rellenan a cero: cualquier lectura de los mismos retornará ceros, y cualquier escritura de ésta memoria no afectará al almacén subyacente, incluso si es **`MAP_SHARED`** (volveremos sobre esto n el apartado 4).

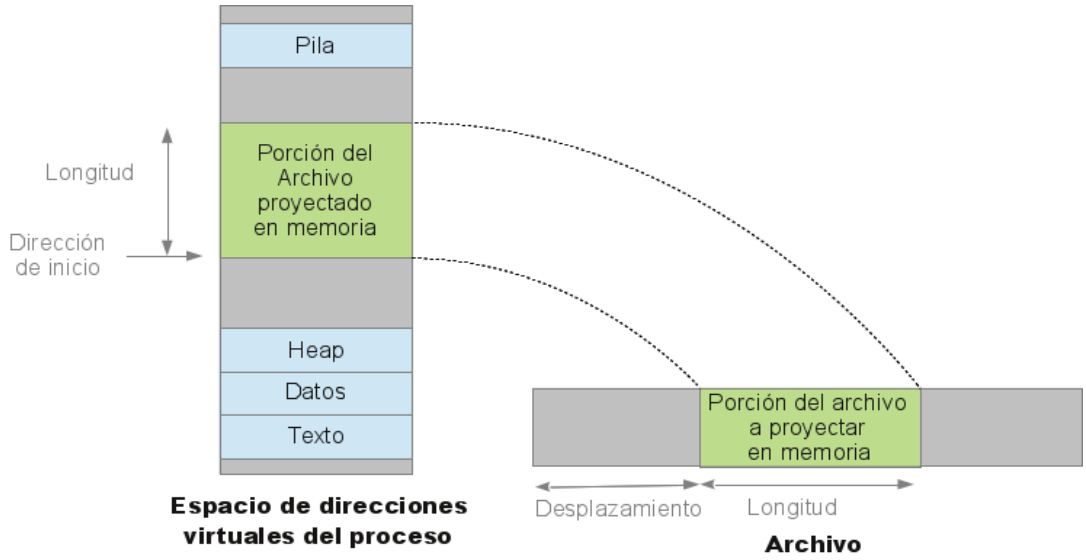


Figura 3.- Visión lógica de una proyección en memoria de un archivo.

El tipo de protección de la proyección viene dado por la máscara de bits **`prot`** y para ello usamos las constantes indicadas en la Tabla 1. Los valores comunes para la protección son **`PROT_READ | PROT_WRITE`**.

Tabla 1.- Valores de protección de una proyección.

Valor `prot`	Descripción
`PROT_READ`	Los datos se pueden leer.
`PROT_WRITE`	Los datos se pueden escribir.
`PROT_EXEC`	Podemos ejecutar los datos
`PROT_NONE`	No podemos acceder a los datos

Los valores del argumento **`flags`** se indican en la Tabla 2. Debemos especificar el indicador **`MAP_SHARED`** o **`MAP_PRIVATE`**. Opcionalmente se puede hacer un **`OR`** con **`MAP_FIXED`**. El significado detallado de algunos de estos indicadores es:

`MAP_PRIVATE` Las modificaciones de los datos proyectados por el proceso son visibles solo para ese proceso y no modifican el objeto subyacente de la proyección (sea un archivo o memoria compartida). El uso principal de este tipo de proyección es que múltiples procesos compartan el código/datos de un ejecutable o biblioteca[13], de forma que las modificaciones que realicen no se guarden en el archivo.

`MAP_SHARED` Las modificaciones de los datos de la proyección son visibles a todos los procesos que la comparten y estos cambios modifican el objeto subyacente. Los dos usos principales son bien realizar entradas/salidas proyectadas en memoria, bien compartir memoria entre procesos.

13 Los sistemas operativos actuales construyen los espacios de direcciones de los procesos proyectando en memoria las regiones de código y datos del archivo ejecutable, así como de las bibliotecas.

MAP_FIXED	Instruye a **mmap()** que la dirección **address** es un requisito, no un consejo. La llamada fallará si el kernel es incapaz de situar la proyección en la dirección indicada. Si la dirección solapa una proyección existente, las páginas solapadas se descartan y se sustituyen por la nueva proyección. No debería especificarse por razones de portabilidad ya que requiere un conocimiento interno del espacio de direcciones del proceso.

Tabla 2.- Indicadores de una proyección.

Valor `flags`	Descripción
MAP_SHARED	Los cambios son compartidos
MAP_PRIVATE	Los cambios son privados
MAP_FIXED	Interpreta exactamente el argumento **address**
MAP_ANONYMOUS	Crea un mapeo anónimo (Apartado 3.2)
MAP_LOCKED	Bloquea las páginas en memoria (al estilo **mlock**)
MAP_NORESERVE	Controla la reserva de espacio de intercambio
MAP_POPULATE	Realiza una lectura adelantada del contenido del archivo
MAP_UNITIALIZED	No limpia(poner a cero) las proyecciones anónimas

A continuación, veremos un ejemplo de cómo crear una proyección. El programa **Programa 2** crea un archivo denominado Archivo y los rellena con nulos. Tras lo cual, crea una proyección compartida del archivo para que los cambios se mantengan. Una vez establecida la proyección, copia en la memoria asignada a la misma el mensaje "**Hola Mundo\n**". Tras finalizar el programa, podemos visualizar el archivo para ver cual es el contenido: la cadena "**Hola Mundo\n**".

Programa 2. Ejemplo de creación de una proyección.

```
// tarea14.c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

const int MMAPSIZE=1024;
int main()
{
  int fd, num;
  char ch='\0';
  char *memoria;
  fd = open("Archivo", O_RDWR|O_CREAT|O_EXCL, S_IRWXU);
  if (fd == -1) {
    perror("El archivo existe");
    exit(1);
  }
 for (int i=0; i < MMAPSIZE; i++){
      num=write(fd, &ch, sizeof(char));
      if (num!=1) printf("Error escritura\n");
  }
  memoria = (char *)mmap(0, MMAPSIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

```
  if (memoria == MAP_FAILED) {
      perror("Fallo la proyeccion");
      exit(2);
  }
  close(fd);  /* no es necesario el descriptor*/
  strcpy(memoria, "Hola Mundo\n"); /* copia la cadena en la proyección */
  exit(0);

}
```

Como hemos podido ver en el ejemplo anterior, los dos pasos básicos para realizar la proyección son:

- Obtener el descriptor del archivo con los permisos apropiados dependientes del tipo de proyección a realizar, normalmente vía **`open()`**.

- Pasar este descriptor de archivo a la llamada **`mmap()`**.

En el programa **`Programa 3`**, ilustramos otro ejemplo de la función **`mmap()`** que utilizamos en este caso tanto para visualizar un archivo completo, que pasamos como primer argumento (similar a la orden **`cat`**), como mostrar el contenido del byte cuyo desplazamiento se pasa como segundo argumento. Mostramos estos dos aspectos para ilustrar cómo, una vez establecida la proyección, podemos acceder a los datos que contiene con cualquier mecanismo para leer de memoria.

Programa 3. Visualizar un archivo con mmap().

```
// tarea15.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
int main (int argc, char *argv[])
{
       struct stat sb;
       off_t len;
       char *p;
       int fd;
       if (argc < 2) {
              printf("Uso: %s archivo\n”, argv[0]);
              return 1;     }
       fd = open (argv[1], O_RDONLY);
       if (fd == -1) {
              printf("Error al abrir archivo\n”);
              return 1;     }
       if (fstat (fd, &sb) == -1) {
              printf("Error al hacer stat\n");
              return 1;     }
       if (!S_ISREG (sb.st_mode)) {
              printf(“%s no es un archivo regular\n",argv[1]);
              return 1;     }
       p = (char *) mmap (0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
       if (p == MAP_FAILED) {
              printf("Fallo el mapeo\n");
              return 1; }
       if (close (fd) == -1) {
              printf("Error al cerrar el archivo\n");
              return 1;
       }
              /* Mostramos el archivo completo */
              printf(%s\n”, p);
```

```
        /* Mostramos en byte con desplazamiento argv[2]*/
        printf("Byte con desplamiento %s es %d: ", argv[2], p[atoi(argv[2])]);
    if (munmap (p, sb.st_size) == -1) {
        printf("Error al cerrar la proyeccion\n");
        return 1;    }
    return 0;
}
```

Para eliminar una proyección del espacio de un proceso, invocaremos a la función:

```
#include <sys/mman.h>

int munmap(void *address, size_t length);

                                Retorna: 0,si OK; -1, si error.
```

El argumento **address** es la dirección que retornó la llamada **mmap()** para esa proyección, y **len** es el tamaño de la región mapeada. Una vez desmapeada, cualquier referencia a la proyección generará la señal **SIGSEGV**. Si una región se declaró **MAP_PRIVATE**, los cambios que se realizaron en ella son descartados.

Durante la proyección, el kernel mantiene sincronizada la región mapeada con el archivo (normalmente en disco) suponiendo que se declaró **MAP_SHARED**. Es decir, si modificamos un dato de la región mapeada, el kernel actualizará en algún instante posterior el archivo. Pero si deseamos que la sincronización sea inmediata debemos utilizar **msync()**.

Pero ¿por qué utilizar **mmap()**? La utilización del mecanismo de proyección de archivos tiene algunas ventajas:

- Desde el punto de vista del programador, simplificamos el código, ya que tras abrir el archivo y establecer el mapeo, no necesitamos realizar operaciones **read()** o **write()**.

- Es más eficiente, especialmente con archivos grandes, ya que no necesitamos copias extras de la información desde del kernel al espacio de usuario y a la inversa. Además, una vez creada la proyección no incurrimos en el coste de llamadas al sistema extras, ya que solo accedemos a memoria.

- Cuando varios procesos proyectan un mismo objeto en memoria, solo hay una copia de él compartida por todos ellos. Las proyecciones de solo-lectura o escritura-compartida son compartidas en cada proceso, las proyecciones de escritura comparten las páginas mediante el *mecanismo COW* (*copy-on-write*).

- La búsqueda de datos en la proyección involucra la manipulación de punteros, por lo que no hay necesidad de utilizar **lseek()**.

No obstante, hay que mantener presentes algunas consideraciones al utilizar la llamada:

- El mapeo de un archivo debe encajar en el espacio de direcciones de un proceso. Con un espacio de direcciones de 32 bits, si hacemos numerosos mapeos de diferentes tamaños, fragmentamos el espacio y podemos hacer que sea difícil encontrar una región libre continua.

- La atomicidad de las operaciones es diferente. La atomicidad de las operaciones en una proyección viene determinada por la atomicidad de la memoria, es decir, la celda de

memoria. Con las operaciones **read()** o **write()** la atomicidad en la modificación de datos viene determinada por el tamaño del búfer que especifiquemos en la operación.

- La visibilidad de los cambios es diferente. Si dos procesos comparten una proyección, la modificación de datos de la proyección por parte de un proceso es instantánemente vista por el otro proceso. Cuando utilizamos la interfaz clásica **read()**/**write()**, si un proceso lee un dato de un archivo y posteriormente otro proceso hace una escritura para modificarlo, el primer proceso solo verá la modificación si vuelve a realizar otra lectura.

- La diferencia entre el tamaño del archivo proyectado y el número de páginas utilizadas en la proyección es espacio "desperdiciado". Para archivos pequeños la porción de espacio desperdiciado es más significativa que para archivos grandes. En muchos casos, este posible desperdicio de espacio se compensa no debiendo tener múltiples copias de la información en memoria.

- No todos los archivos pueden ser proyectados. Si intentamos proyectar un descriptor que referencia a un terminal o a un socket, la llamada genera error. Estos descriptores deben accederse mediante **read()** o **write()** o variantes.

Otras funciones relacionadas con la protección de archivos son:

`mremap()`: se utiliza para extender una proyección existente.

`mprotect()`: cambia la protección de una proyección.

`madvise()`: establece consejos sobre el uso de memoria, es decir, como manejar las entradas/salidas de páginas de una proyección.

`remap_file_pages()`: permite crear mapeos no-lineales, es decir, mapeos donde las páginas del archivo aparecen en un orden diferente dentro de la memoria contigua.

`mlock()`: permite bloquear (anclar) páginas en memoria.

`mincore()`: informa sobre las páginas que están actualmente en RAM

3.1 Compartición de memoria

Una forma de compartir memoria entre un proceso padre y un hijo es invocar **mmap()** con **MAP_SHARED** en el padre antes de invocar a **fork()**. El estándar POSIX.1 garantiza que la proyección creada por el padre se mantiene en el hijo. Es más, los cambios realizados por un proceso son visibles en el otro.

Para mostrarlo, en el programa **Programa 4** construimos un ejemplo donde un proceso padre crea una proyección que se utiliza para almacenar un valor. El padre asignará valor a **cnt** y el hijo solo leerá el valor asignado por el padre (lo hacemos así -solo lectura en el hijo- para evitar condiciones de carrera y así evitarnos tener que introducir un mecanismo de sincronización para acceder al contador). También podemos ver cómo el archivo contiene el valor modificado del padre.

Programa 4. Compartición de memoria entre procesos padre-hijo con **`mmap()`**.

```
// tarea16.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

#define MMAPSIZE 1

int main (int argc, char *argv[])
{
    off_t len;
    char bufer='a';
    char *cnt;
    int fd;

    fd = open("Archivo", O_RDWR|O_CREAT|O_TRUNC, S_IRWXU);
    if (fd == -1) {
    perror("El archivo existe");
    exit(1);
    }
    write(fd, &bufer, sizeof(char));

    cnt = (char *) mmap (0, MMAPSIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (cnt == MAP_FAILED) {
       printf("Fallo el mapeo\n");
       return 1;
    }
    if (close (fd) == -1) {
      printf("Error al cerrar el archivo\n");
      return 1;   }

if (fork() == 0) { /* hijo */
  sleep(2);
  printf("El valor de cnt es: %d", cnt);
  exit(0);
  }
  /* padre */
  strcpy(cnt, "b");
  exit(0);
}
```

La compartición de memoria no está restringida a procesos emparentados, cualesquiera procesos que mapeen la misma región de un archivo, comparten las mismas páginas de memoria física. El que estos procesos vean o no las modificaciones realizadas por otros dependerá de que el mapeo sea compartido o privado.

3.2 Proyecciones anónimas

Una *proyección anónima* es similar una proyección de archivo salvo que no existe el correspondiente archivo de respaldo. En su lugar, las páginas de la proyección son inicializadas a cero. La Tabla 4 resumen los propósitos de los diferentes tipos de proyecciones.

Tabla 4.- Propósitos de los diferentes tipos de proyecciones en memoria.

Visibilidad de las modificaciones	Tipo de proyección	
	Archivo	Anónimo
Privado	Inicializa memoria con el contenido del archivo	Asignación de memoria
Compartido	E/S proyectadas en memoria Compartición de memoria (IPC)	Compartición de memoria (IPC)

El ejemplo mostrado en el Programa 3 funciona correctamente pero nos ha obligado a crear un archivo en el sistema de archivos (si no existía ya) y a inicializarlo. Cuando utilizamos una proyección para ser compartida entre procesos padre e hijo, podemos simplificar el escenario de varias formas, dependiendo del sistema:

- Algunos sistemas suministran las proyecciones anónimas que nos evitan tener que crear y abrir un archivo. En su lugar, podemos utilizar el indicador **`MAP_ANON`** (o **`MAP_ANONYMOUS`**) en **`mmap()`** el valor -1 para **`fd`** (el **`offset`** se ignora). La memoria se inicializa a cero. Esta forma está en desuso (en Linux se implementa a través de */dev/zero*) y la podemos ver en el programa **`Programa 5`**.

- La mayoría de los sistemas suministran el seudo-dispositivo **`/dev/zero`** que tras abrirlo y leerlo suministra tantos ceros como le indiquemos, y cualquier cosa que escribamos en él es descartada. Su uso se ilustra en el programa **`Programa 6`**.

Programa 5. Proyección anónima con **`MAP_ANON`**.

```
// tarea17.c
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

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

    p = (char *)mmap (0, sizeof(char), PROT_READ , MAP_SHARED |MAP_ANON, -1, 0);
    if (p == MAP_FAILED) {
              printf("Fallo el mapeo\n");
              return 1;
       }
       close(fd);
      return 0;
}
```

Programa 6. Proyección anónima con **`/dev/zero`**.

```
// tarea18.c
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

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

  fd = open("/dev/zero", O_RDONLY);

  p = (char *) mmap (0, sizeof(int), PROT_READ , MAP_SHARED, fd, 0);
  if (p == MAP_FAILED) {
       printf("Fallo el mapeo\n");
       return 1;
  }
  close(fd);
  return 0;
}
```

Si unimos este hecho a los vistos en el Apartado 2, podemos deducir que una de las utilidades de las proyecciones anónimas es que varios procesos emparentados compartan memoria. También se puede utilizar como mecanismo de reserva y asignación de memoria en un proceso.

3.3 Tamaño de la proyección y del archivo proyectado

En muchos casos, el tamaño del mapeo es múltiplo del tamaño de página, y cae completamente dentro de los límites del archivo proyectado. Sin embargo, no es necesariamente así, por ello vamos a ver que ocurre cuando no se dan esas condiciones.

La Figura 4a describe el caso en el que la proyección está dentro de los límites del archivo proyectado pero el tamaño de la región no es múltiplo del tamaño de página del sistema. Suponemos en el sistema tiene páginas de 4KiB y que realizamos una proyección de 5999 B. En este caso, la región proyectada ocupará 2 páginas de memoria si bien la proyección del archivo es menor. Cualquier intento de acceder a la zona redondeada hasta el límite de página provocará la generación de la señal `SIGSEGV`.

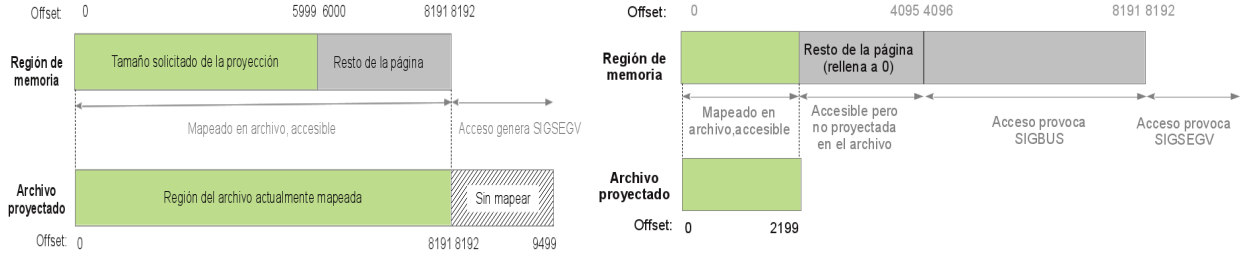


Figura 4.- Proyección con tamaño (a) no múltiplo del tamaño de página, (b) mayor que el archivo de respaldo.

Cuando la proyección se extiende más allá del fin de archivo (Figura 4b), la situación es algo más compleja. Como antes, si el tamaño de la proyección no es múltiplo del tamaño de página, ésta se redondea por exceso. En la Figura 4, creamos una proyección de 8192 B de un archivo que tiene 2199 B. En éste los bytes de redondeo de la proyección hasta completar una página (bytes 2199 al 4095, en el ejemplo), son accesibles pero no se mapean en el archivo de respaldo, y se inicializan a 0. Estos bytes nunca son compartidos con la proyección del archivo con otro proceso. Su modificación no se almacena en el archivo.

Si la proyección incluye páginas por encima de la página redondeada por exceso, cualquier intento de acceder esta zona generará la señal **`SIGBUS`**. En nuestro ejemplo, los bytes comprendidos entre la dirección 4096 y la 8192. Cualquier intento de acceder por encima de la dirección 8191, generará la señal **`SIGSEGV`**.

El programa **`Programa 7`** muestra la forma habitual de manejar un archivo que esta creciendo: especifica una proyección mayor que el archivo, tiene en cuenta del tamaño actual del archivo (asegurándonos no hacer referencias a posiciones posteriores al fin de archivo), y deja que que se incremente el tamaño del archivo conforme se escribe en él.

Programa 7. Proyección que deja crecer un archivo.

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

#define FILE "datos"
#define SIZE 32768

int main(int argc, char **argv)
{
       int    fd, i;
       char   *ptr;

       fd = open(FILE, O_RDWR|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
       ptr = (char*)mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

       for (i = 4096; i<= SIZE; i += 4096) {
         printf("Ajustando tamaño archivo a %d, i);
         ftruncate(fd, i);
         printf("ptr["<< i-1 <<"] = ", <<ptr[i - 1])endl;
       }
       exit(0);
}
```

En el Programa, hemos utilizado la función **ftruncate()** que trunca el archivo indicado por **fd** al tamaño indicado como segundo argumento. Si el tamaño indicado es mayor que el tamaño del archivo, su contenido se rellena con nulos.

La ejecución del programa se muestra a continuación. Además, podemos ver como el tamaño del archivo efectivamente se ha modificado.

```
~/tmp> ./m7
Ajustando tamaño archivo a 4096
ptr[4095] =
Ajustando tamaño archivo a 8192
ptr[8191] =
. . .
Ajustando tamaño archivo a 32768
ptr[32767] =
~/tmp> ls -l datos
-rw-r--r-- 1 jose users 32768 ene 14 18:31 datos
```

Actividad 6.3 Trabajo con archivos proyectados

Ejercicio 5: Escribir un programa, similar a la orden **cp**, que utilice para su implementación la llamada al sistema **mmap()** y una función de C que nos permite copiar memoria, como por ejemplo **memcpy()**. Para conocer el tamaño del archivo origen podemos utilizar **stat()** y para establecer el tamaño del archivo destino se puede usar **ftruncate()**.

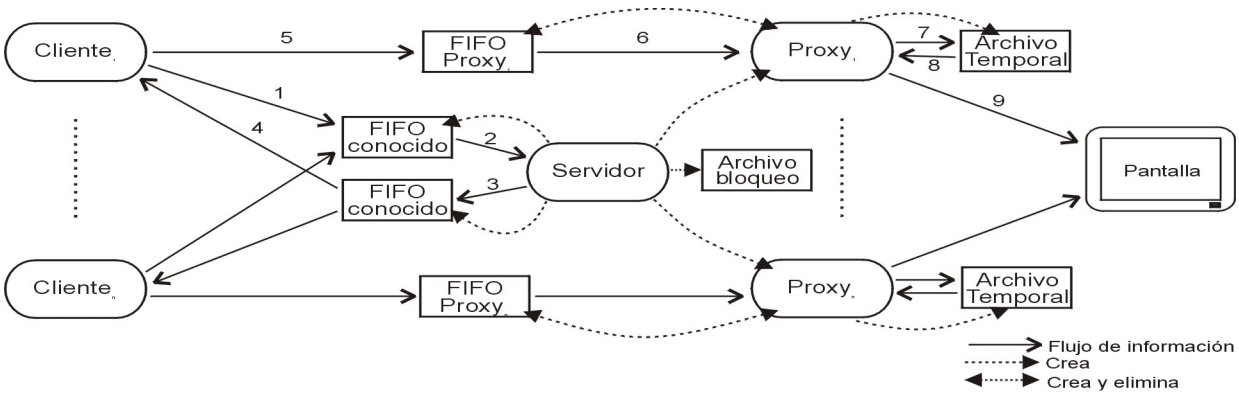
Módulo II. Uso de los Servicios del SO Linux mediante la API

Sesión 7. Construcción de un spool de impresión

Sesión 7. Construcción de un spool de impresión

1. Enunciado del ejercicio

Esta práctica tratará aspectos *relacionados* con procesos, señales y cauces con nombre (archivos FIFO). El ejercicio plantea la programación de 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. 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:

1. Un cliente solicita la impresión de un archivo enviando un mensaje (cuyo contenido no tiene importancia) al servidor a través de un FIFO cuyo nombre es conocido.

2. 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.
3. El servidor responde al cliente a través de otro FIFO de nombre conocido, informando de la identidad (PID) del *proxy* que va a atender su petición. 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 puede ser el propio PID del *proxy*. El *proxy* se encargará de eliminar dicho FIFO cuando ya no sea necesario (justo antes de terminar su ejecución).
4. El cliente lee esta información que le envía el servidor, de manera que así sabrá donde enviar los datos a imprimir.
5. 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.
6. El *proxy* obtendrá la información a imprimir llevando a cabo probablemente varias lecturas como ésta del FIFO.
7. 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.
8. 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.
9. Cada lectura de datos realizada en el paso anterior implicará su escritura en pantalla.

Ten en cuenta las siguientes consideraciones de cara a la implementación:

- Utiliza 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.
- Recuerda 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.
- Utiliza 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 la estructura **`FILE`** (**`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 manejando en el servidor las señales **SIGCHLD** que envían los procesos *proxy* (ya que son hijos del servidor) cuando terminan. Por omisión, 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 en estado zombi indefinidamente.
- Por cuestiones de reusabilidad, el programa *proxy* leerá de su entrada estándar (constante **STDIN_FILENO**) y escribirá en su salida estándar (constante **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 las funciones vistas en la sesión 6 para bloqueo de archivos sobre un archivo llamado **bloqueo** que será creado en el proceso servidor.

También se proporciona un programa denominado **clientes.c** 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:

```
$> 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”.

Implemente el resto de programas según lo descrito, para ello necesitará además de las funciones y llamadas al sistema comentadas anteriormente, otras como: **mkfifo** (crea un archivo FIFO), **creat** (crea un archivo normal), **unlink** (borra un archivo de cualquier tipo).

Ten en cuenta que el servidor debe ser un proceso 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.