

# 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)

---

## 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:

- 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`).
- 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.
- 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>
```

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

**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[]="ABCDEFGH IJ";
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; /* n° de dispositivo (filesystem) */
    dev_t st_rdev; /* n° de dispositivo para archivos especiales */
    ino_t st_ino; /* n° 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 formato 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`:

<code>S_IFMT</code>	0170000	máscara de bits para los campos de bit del tipo de archivo (no POSIX)
<code>S_IFSOCK</code>	0140000	socket (no POSIX)
<code>S_IFLNK</code>	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	<b>user</b> (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	<b>group</b> 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	<b>other</b> 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 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`.