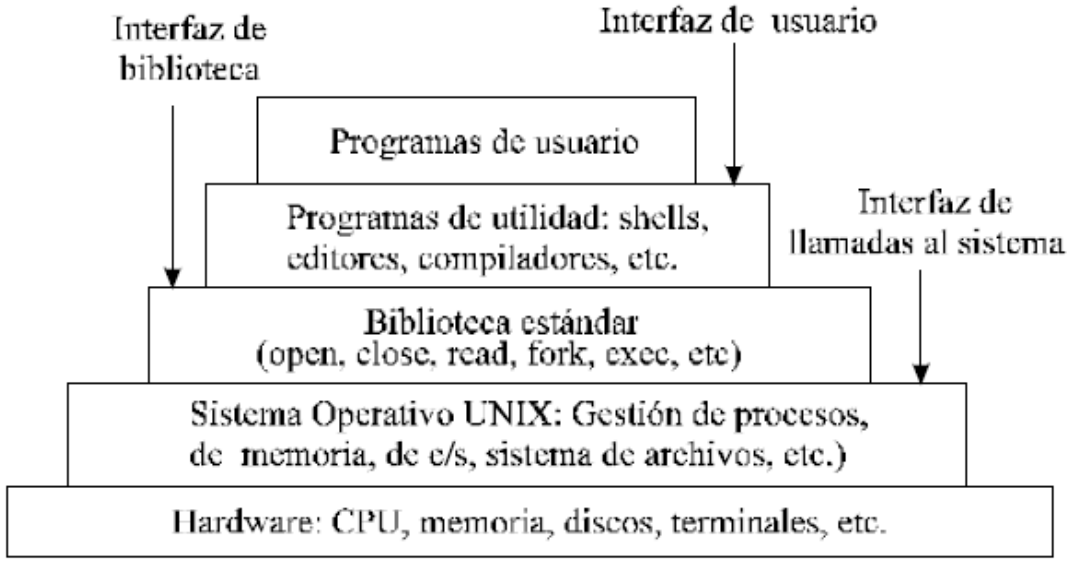
Sistemas Operativos

UNIDAD 1: INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS

INTRODUCCIÓN A LAS LLAMADAS AL SISTEMA

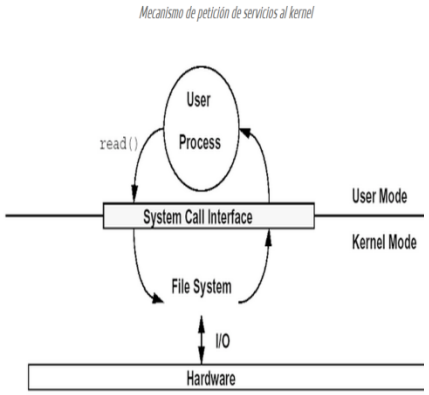
Llamadas al sistema

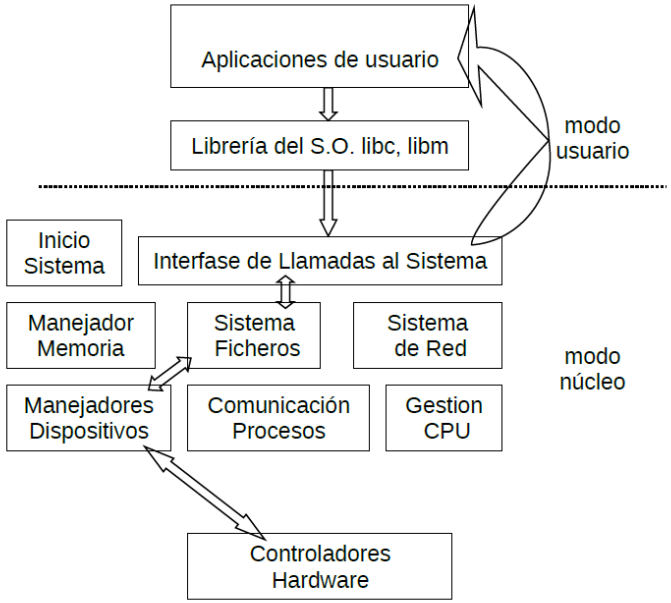


La llamada al sistema es la forma en la cual un proceso requiere un servicio específico de núcleo.

Mecanismo de las llamadas al sistema

El código de la llamada al sistema se ejecuta en:





Modo Privilegiado
(Privileged Mode – kernel space)

Tiene acceso completo a los recursos hardware

Gracias a esto el kernel mantiene el control del sistema

Modo Usuario (User Mode – user space)

Acceso limitado a los recursos hardware

Si el código intenta ejecutar una instrucción privilegiada el microprocesador avisa al kernel quien normalmente mata al proceso.

Tiene que pedirle al kernel que ejecute la operación por el, por ejemplo para:

- Acceso a ficheros y a la red
- Crear y destruir procesos
- Apropiarse de más memoria (allocating memory)

Llamadas al sistema

- Casi todos los sistemas operativos Unix-Like tienen las mismas llamadas al sistemas aunque varíen detalles.
- El mecanismo real de la llamada al sistema depende de la máquina y se realiza en ensamblador.
- Se ofrecen funciones C (funciones de librería) que encapsulan los detalles de la implementación de la llamada al sistema.
  Estandar POSIX define muchos APIs para Linux y sistemas Unix-Like .
- Antes de poder escribir Software para un sistema es necesario conocer los APIs que ofrece.
  En el núcleo de todos ellos reside el API de las llamadas al sistema.

Tipos de llamadas al sistema

Control y gestión de procesos:

Fin, abortar, cargar, ejecutar, crear, finalizar, obtener y establecer atributos, espera, asignar y liberar memoria.

Manipulación de archivos:

Crear y eliminar archivo, abrir y cerrar, leer, escribir, reposicionar, obtener y establecer atributos.

Manipulación de dispositivos:

Solicitar y liberar, leer, escribir, reposicionar, obtener y establecer atributos, conectar y desconectar dispositivos.

Mantenimiento de información:

Obtener y establecer hora, fecha, datos del sistema, atributos de un proceso, archivo o dispositivo.

Comunicaciones, señales:

Crear, eliminar conexiones; enviar y recibir mensajes, ...

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Tipos de llamadas al sistema

Control y gestión de procesos:

Fin, abortar, cargar, ejecutar, crear, finalizar, obtener y establecer atributos, espera, asignar y liberar memoria.

Manipulación de archivos:

Crear y eliminar archivo, abrir y cerrar, leer, escribir, reposicionar, obtener y establecer atributos.

Manipulación de dispositivos:

Solicitar y liberar, leer, escribir, reposicionar, obtener y establecer atributos, conectar y desconectar dispositivos.

Mantenimiento de información:

Obtener y establecer hora, fecha, datos del sistema, atributos de un proceso, archivo o dispositivo.

Comunicaciones, señales:

Crear, eliminar conexiones; enviar y recibir mensajes, ...

Funcionalidad	Windows	Unix/Linux
Control de procesos	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
Gestión de ficheros	CreateFile() ReadFile() WriteFile() ClsoeHandle()	open() read() write() close()
Gestión de dispositivos	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Información y mantenimiento	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Comunicaciones	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protección	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Pasos de una llamada al sistema

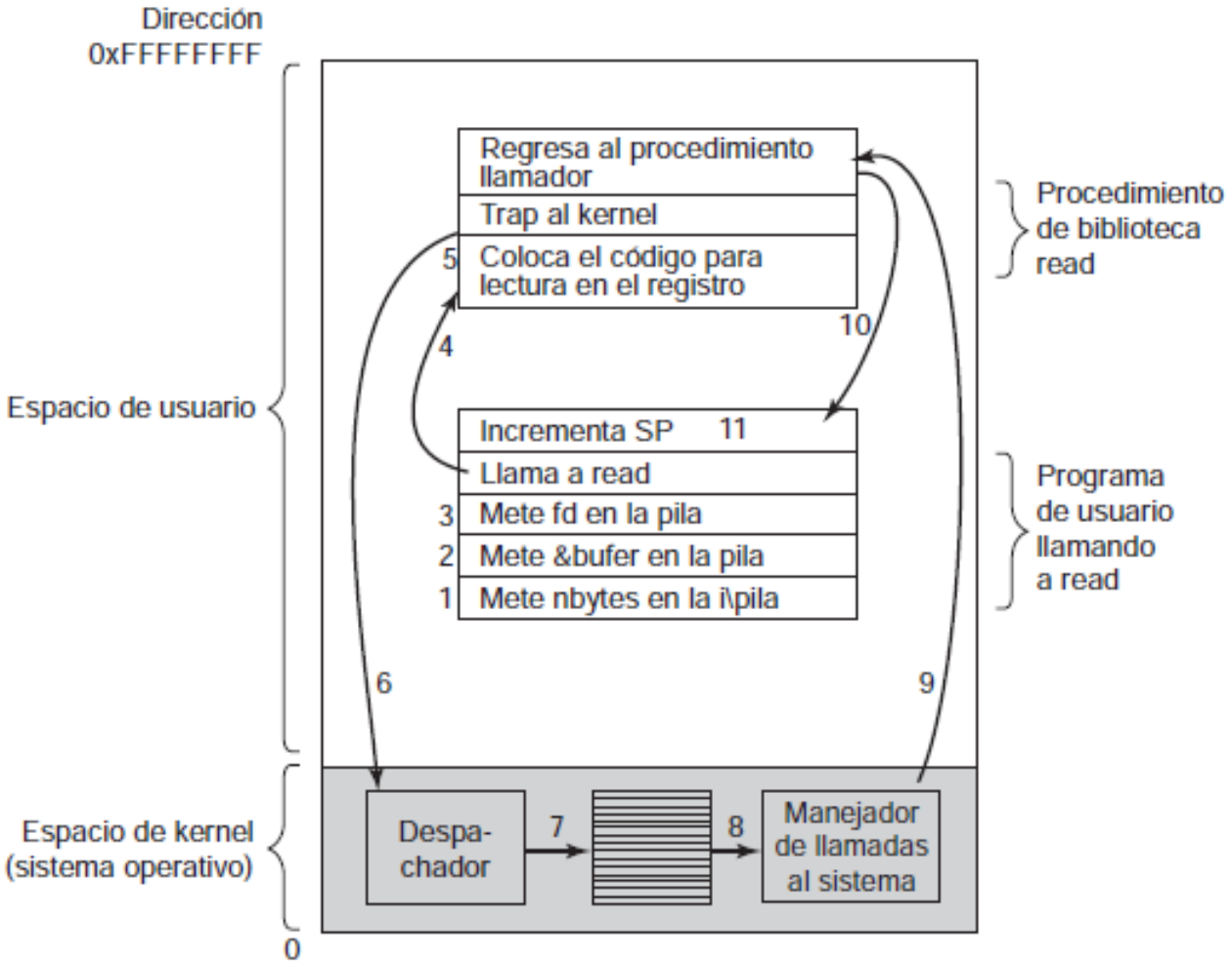


Figura 1-17. Los 11 pasos para realizar la llamada al sistema `read(fd, bufer, nbytes)`.

Llamadas al sistema

Las llamadas al sistema permiten a los programas de usuario pedir servicios del kernel.

En C parecen funciones normales, no solo se transfiere el control de ejecución a la función sino que se pasa al procesador en kernel mode.

Cualquier proceso linux se puede ver como un bucle que:

1. Realiza algún cálculo
2. Hace una llamada al sistema
3. Vuelve al paso 1

Simplificación:

Cualquier programa se limita a realizar llamadas al sistema
y cálculos que entre otras cosas deciden que nueva llamada al sistema realizar.

Los programas se definen por la secuencia de llamadas al sistema que realizan.

La mayoría de las distribuciones linux tienen una utilidad que permite ver las llamadas al sistema que realizan los procesos. *Strace*

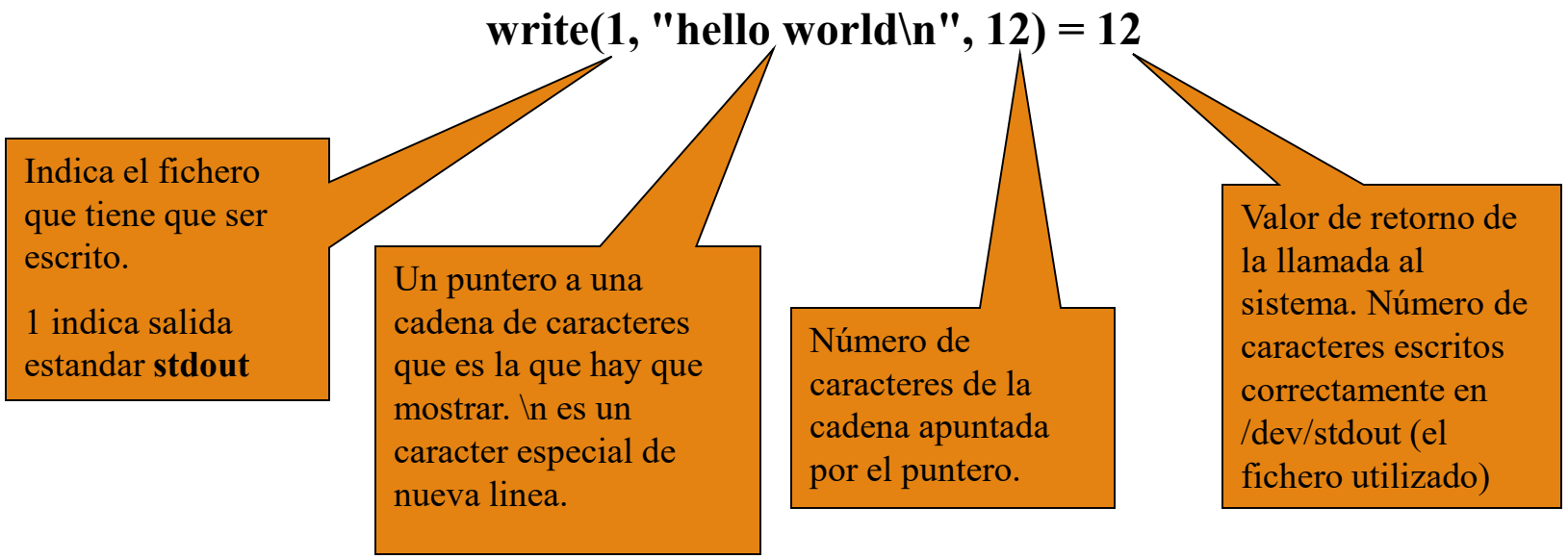
strace

$ strace echo hello world

```
mmartinez@ideafix:~$ strace echo hello world
execve("/bin/echo", ["echo", "hello", "world"], [/* 13 vars */]) = 0
uname({sys="Linux", node="ideafix", ...}) = 0
brk(0)                                  = 0x804b724
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=19890, ...}) = 0
old_mmap(NULL, 19890, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40012000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\275Z\1"..., 1024) = 1024
fstat64(3, {st_mode=S_IFREG|0755, st_size=1103880, ...}) = 0
old_mmap(NULL, 1113636, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x40017000
mprotect(0x4011f000, 32292, PROT_NONE)  = 0
old_mmap(0x4011f000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x107000) = 0x4011f000
old_mmap(0x40125000, 7716, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40125000
close(3)                                = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40127000
munmap(0x40012000, 19890)               = 0
brk(0)                                  = 0x804b724
brk(0x804c724)                          = 0x804c724
brk(0)                                  = 0x804c724
brk(0x804d000)                          = 0x804d000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40012000
write(1, "hello world\n", 12hello world
)            = 12
munmap(0x40012000, 4096)                = 0
semget(IPC_PRIVATE, 4096, IPC_EXCL|IPC_NOWAIT|0x40120040|0) = -1 ENOSYS (Function not implemented)
_exit(0)                                = ?
mmartinez@ideafix:~$
```

Llamadas al sistema

La mayoría de las llamadas tienen que ver con la carga del programa en si y de la inicialización de la librería C.



Llamadas al sistema

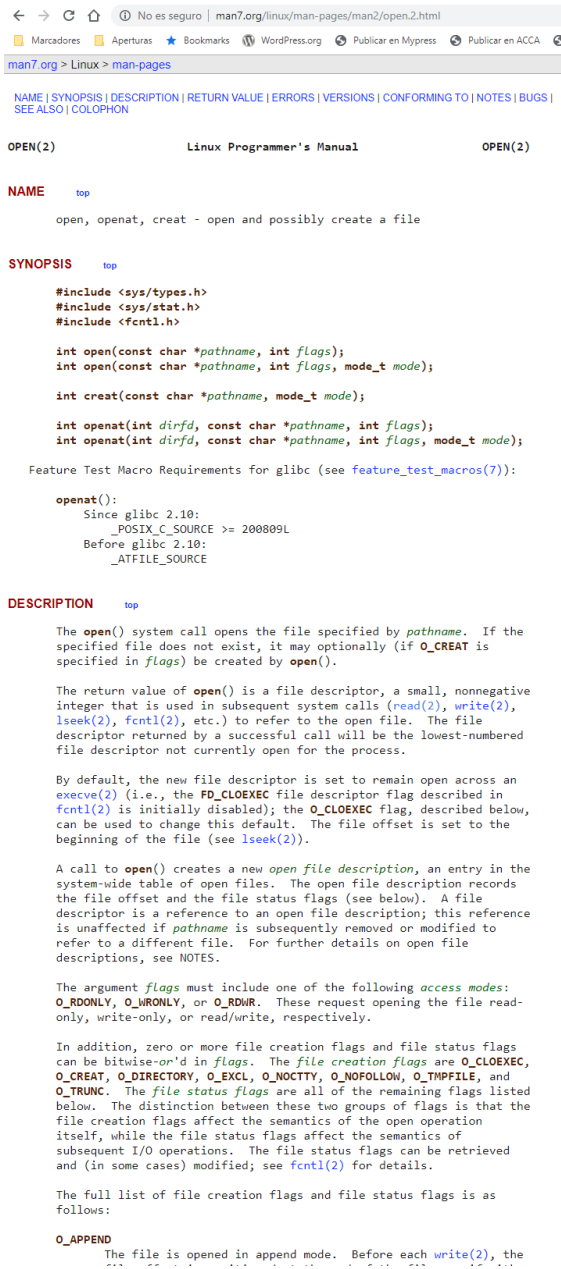
Ejemplo de llamada:

count = read(file, buffer, nbyte);

- read -> nombre de la llamada al sistema
- file -> fichero de donde leer
- buffer -> zona de memoria donde colocar los dato
- nbytes -> nº de bytes a leer
- count -> nº de bytes leídos, si count es - 1, hay error, que se coloca en la variable global errno

Veamos otro ejemplo con **open** para abrir el fichero especial "/dev/tty" vinculado con la consola asociada al proceso

```
# include <fcntl.h>
# include <stdlib.h>
int main (int argc, char **argv) {
    // abrimos la consola
    int fd = open("/dev/tty", O_RDWR);
    /* el descriptor de fichero "fd" devuelto por open es utilizado
       por write para escribir */
    write(fd, "Hola joven!\n", 12);
    // cerramos el fichero
    close(fd);
    exit(0);
}
```



No es seguro | man7.org/linux/man-pages/man2/open.2.html

Marcadores Aperturas Bookmarks WordPress.org Publicar en Mypress Publicar en ACCA

man7.org > Linux > man-pages

NAME | SYNOPSIS | DESCRIPTION | RETURN VALUE | ERRORS | VERSIONS | CONFORMING TO | NOTES | BUGS | SEE ALSO | COLOPHON

OPEN(2) Linux Programmer's Manual OPEN(2)

NAME top

open, openat, creat - open and possibly create a file

SYNOPSIS top

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

int creat(const char *pathname, mode_t mode);

int openat(int dirfd, const char *pathname, int flags);
int openat(int dirfd, const char *pathname, int flags, mode_t mode);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```
openat():
    Since glibc 2.10:
        _POSIX_C_SOURCE >= 200809L
    Before glibc 2.10:
        _ATFILE_SOURCE
```

DESCRIPTION top

The **open**() system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if **O_CREAT** is specified in *flags*) be created by **open**().

The return value of **open**() is a file descriptor, a small, nonnegative integer that is used in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.) to refer to the open file. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

By default, the new file descriptor is set to remain open across an execve(2) (i.e., the **FD_CLOEXEC** file descriptor flag described in fcntl(2) is initially disabled); the **O_CLOEXEC** flag, described below, can be used to change this default. The file offset is set to the beginning of the file (see lseek(2)).

A call to **open**() creates a new *open file description*, an entry in the system-wide table of open files. The open file description records the file offset and the file status flags (see below). A file descriptor is a reference to an open file description; this reference is unaffected if *pathname* is subsequently removed or modified to refer to a different file. For further details on open file descriptions, see NOTES.

The argument *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

In addition, zero or more file creation flags and file status flags can be bitwise-or'd in *flags*. The *file creation flags* are **O_CLOEXEC**, **O_CREAT**, **O_DIRECTORY**, **O_EXCL**, **O_NOCTTY**, **O_NOFOLLOW**, **O_TMPFILE**, and **O_TRUNC**. The *file status flags* are all of the remaining flags listed below. The distinction between these two groups of flags is that the file creation flags affect the semantics of the open operation itself, while the file status flags affect the semantics of subsequent I/O operations. The file status flags can be retrieved and (in some cases) modified; see fcntl(2) for details.

The full list of file creation flags and file status flags is as follows:

O_APPEND
The file is opened in append mode. Before each write(2), the

Llamadas al sistema

¿Que pasa cuando una llamada al sistema falla?

$ cat /ABC Si en el directorio root no hay ningún fichero nombrado ABC la llamada falla

```
brk(0x804d000)                          = 0x804d000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
open("/ABC", O_RDONLY|O_LARGEFILE)      = -1 ENOENT (No such file or directory)
write(2, "cat: ", 5cat: )                        = 5
```

Strace informa que echo falla y da el error ENOENT que indica que el fichero no fue abierto porque no existe en el sistema.

La llamada al sistema devuelve –1 que es una forma estándar del kernel de indicar ERROR, pero el código exacto de error es un entero que se guarda en la variable *errno*. Cualquier programa que quiera acceder deberá incluir algo asi:

```
returnCode = open(somefile, O_RDONLY);
if (returnCode < 0) {
    /* Handle the error */
    printf("Error %d occured\n",
            errno);
   }
```

En /usr/include/asm/errno.h se encuentra la relación completa de constantes de error y sus significado.
Usar los simbólicos pues las constantes pueden no ser portables

Llamadas al sistema

Los programas deben siempre comprobar después de una llamada si todo es correcto, para ello Linux proporciona una variable **errno** y una función **perror**(), ya definidas por el sistema.

Ejemplo de programa, del uso de la variable global **errno** y del procedimiento **perror**()

```
/*
* errores.c
* lista los 53 primeros errores de
llamadas al sistema
*/
# include <stdio.h>
main(int argc, char **argv){
int i;
extern int errno;
for (i=0;i<=53;i++){
  fprintf(stderr,"%3d",i);
  errno=i;
  perror(" ");
  }
exit(0);
}
```

Algunas llamadas simples:

void exit(int errcode); #Termina el proceso que la invoca devolviendo el errcode al padre

pid_t getpid(void); #devuelve el PID del proceso que la invoca. pid_t es un 32 bit integer

printf ("Soy el proceso %d\n",getpid());

pid_t fork(void); #Crea un proceso hijo con pid pid_t. El padre recibe el pid del hijo y el hijo recibe 0.

En linux kernel v.2.0.36 hay menos de 300 llamadas al sistema, pero son suficientes para permitir la ejecución de las aplicaciones actuales sobre Linux.

Conocerlas permite al programador saber que puede hacer el kernel por el.

Llamadas para manejo de procesos

pid = fork() - crea un proceso hijo idéntico al proceso padre.

pid = waitpid(pid, &statloc, opts) - espera a que un hijo determinado termine y coge su condición de salida.

s = wait(&status) - espera a que un proceso hijo termine y coge su condición de salida devuelve el pid del hijo que termina..

s = execve (name, argv, envp) - sustituye la imagen en memoria de un proceso.

exit(status) - pone fin a la ejecución del proceso y devuelve la condición de salida.

size = brk (addr) - fija el tamaño del segmento de datos a (addr).

pid = getpid() - devuelve el id del proceso solicitante.

pid = getppid() – devuelve el id del proceso padre del solicitante.

pid = getpgrp() - devuelve el id de grupo de procesos del proceso solicitante.

pid = setsid() - crea una nueva sesión y asigna el pid del invocador como identificador del grupo de procesos.

s = ptrace(req, pid, addr, data) - controla un proceso para depurar.

Llamadas para manejo de señales

s = signal(sig,&función) - define la acción a realizar cuando se recibe una señal.

s = sigaction(sig, act, oldact) - controla el estado y manejo de señales.

s = sigreturn(&context) - regresa de una señal.

s = sigprocmask(how, &set, &old) - examina o cambia la mascara de las señales.

s = sigpending(set) - obtiene el conjunto de señales bloqueadas.

s = sigsuspend(sigmask) - sustituye la mascara de señales y suspende el proceso.

s = kill (pid, sig) - envía una señal a un proceso.

residual = alarm(seconds) - planifica o programa una señal SIGALRM después de un cierto tiempo.

s = pause() - suspende el solicitante hasta la siguiente señal.

Llamadas para manejo de ficheros

fd =creat (name, mode) - crea un nuevo fichero o trunca uno existente.

fd = mknod (name, mode, addr) - crea un nodo i normal, especial, o de directorio.

fd = open (file, how, ...) - abre un fichero para lectura, escritura o ambos.

s = close (fd) - cierra un fichero abierto.

n = read (fd, buffer, nbytes) - lee datos de un fichero y los coloca en un buffer.

n = write (fd, buffer, nbytes) - escribe datos a un fichero desde un buffer.

pos = lseek (fd, offset, whence) - mueve el apuntador del fichero.

s = stat (name, &buf) - lee y devuelve el estado de un fichero de su nodo i.

s = fstat (fd, buf) - lee y devuelve el estado de un fichero a partir de su nodo i.

fd = dup (fd) - asigna otro descriptor de fichero para un fichero abierto.

s = pipe (&fd [0]) - crea una tubería.

s = ioctl(fd, request, argp) - realiza operaciones especiales en ficheros especiales.

s = access(name,amode) - verifica los accesos a un fichero.

s = rename(old, new) - cambia el nombre de un fichero.

s = fcntl(fd, cmd, ...) - bloqueo de un fichero y otras operaciones.

Llamadas para manejo directorios y sistema de ficheros

s = mkdir(name, mode) - crea un nuevo directorio.

s = rmdir(name) - elimina un directorio vacio.

s = link (name1, name2) - crea una entrada nueva name2 en el directorio, que apunta al fichero name1.

s = unlink (name) - elimina una entrada del directorio.

s = mount (special, name, rwflag) - monta un sistema de ficheros.

s = unmount (special) - desmonta un sistema de ficheros.

s = sync() - escribe todos los bloques de la memoria cache en el disco.

s = chdir (dirname) - cambia el directorio de trabajo.

s = chroot (dirname) - cambia al directorio raíz.

Llamadas para protección

s = chmod (name, mode) - cambia los bits de protección asociados con un fichero.

iud = getuid() - determina el uid del solicitante.

gid = getgid() - determina el gid del solicitante.

s = setuid(uid) - fija el uid del solicitante.

s = setgid(gid) - fija el gid del solicitante.

s = chown (name, owner, group) - cambia el propietario y grupo de un fichero.

oldmask = umask (complmode) - pone una máscara que se utiliza para fijar los bits de protección.

Fin

UNIDAD 1: INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS
INTRODUCCIÓN A LAS LLAMADAS AL SISTEMA