

Branch: master ▾

SO-FIB / Utilidades / resum2.md

Find file

Copy path

MrRobb Update resum2.md

e869f21 11 days ago

2 contributors 

715 lines (523 sloc) 19.5 KB

## Entrada / Salida

- Entrada: exterior -> proceso
- Salida: proceso -> exterior

Lo que queremos es abstraer toda la complejidad del acceso a los datos de entrada y salida a una interfaz simple y sencilla.

Pasar de puertos (registros de control, de estado...) a *cin* y *cout*

Desafortunadamente, en medio de estas cosas está:

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
ssize_t read(int fildes, void *buf, size_t nbyte);
```

Básicamente, leer y escribir.

### Tipos de dispositivos

Se organizan según:

- Tipo:
  - Lógico
  - Físico
  - de Red
- Velocidad de acceso
- Flujo de acceso
- Exclusividad de acceso
- ...

### ¿Qué queremos?

- Operaciones **uniformes**: usar las mismas llamadas al sistema para todos los dispositivos.
- Utilizar **dispositivos virtuales**: utilizar identificadores que se traducen luego
- **Redireccionamiento**: pipes (lo veremos luego)

### Principios de diseño

#### Virtual

Aporta:

Independencia, trabaja con dispositivos virtuales

Trabajamos con canales, que son un número que representa un dispositivo:

- 0 ---> stdin (input) // teclado
- 1 ---> stdout (output) // pantalla
- 2 ---> stderr (errores) // pantalla

Permiten redireccionamiento, vamos que cuando haces:

```
./mi_programa > fichero //Redireccionas el canal 1 al fichero
```

```
./mi_programa < fichero //Redireccionas el canal 0 al fichero
```

Por eso en el write ponemos:

```
write(1, ..., ...); // escribir en stdout
```

Quien dice qué números corresponden a qué dispositivos, lo maneja el nivel lógico.

Los dispositivos son ficheros, bueno, no exactamente, los dispositivos se tratan como ficheros, para poder leer o escribir de/en ellos.

## Lógico

Aporta:

- Compartición de dispositivos (accesos concurrentes al mismo dispositivos, da orden)

Enlaza dispositivo virtual y dispositivo (físico)

Manipula bloques de datos de medida la que quieras.

Si haces el siguiente código, ...

```
int canal = open("path_a_la_impresora");
write(canal, "hola\n", 5);
```

... en canal se te guarda el número del cual tu puedes hacer write y se mandaría a la impresora.

Se tienen en cuenta los permisos, ya que se accede como si fuese un fichero

## Físico

Aporta:

Implementación a bajo nivel, traduce los parámetros lógicos a parámetros concretos.

Funcionamiento:

1. Inicializa el dispositivo
2. Mira si está libre
3. Hace la programación de la operación que se le pide
4. Espera o no a que termine
5. Devuelve los resultados o informa de algún error

Los dispositivos se identifican con:

- Tipo: **block/character** ---> medida de escritura/lectura por defecto (puede ser un byte(char) o un bloque(medida que tu pones))
- **Major** ---> (int) indica el tipo de dispositivo
- **Minor** ---> (int) instancia concreta según el major

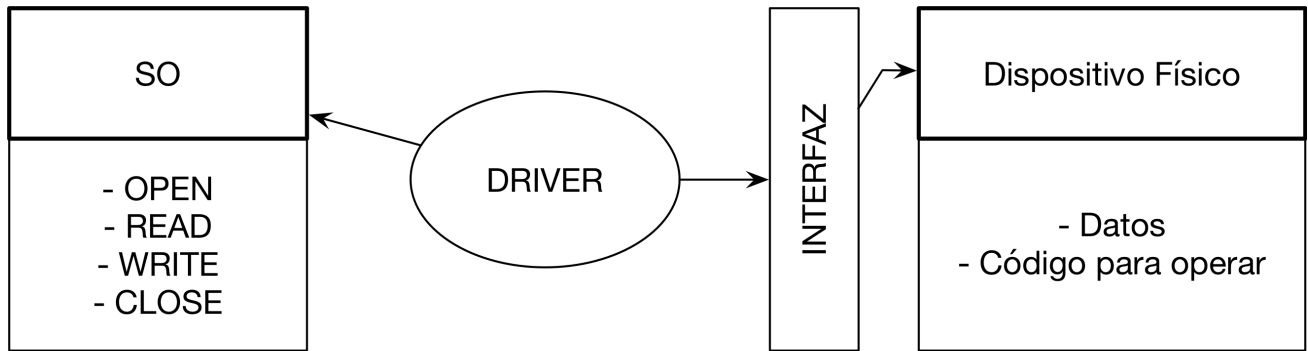
## Drivers

Def Es un controlador; controla un dispositivo.

## Nivel Físico

Aumentan la funcionalidad del sistema operativo.

Con el MAJOR y el MINOR (son int) identifican el dispositivo dentro del kernel.



Si el driver cumple con las funciones que ofrece el SO el driver se puede instalar

## MODO PRIVILEGIADO

## Nivel Lógico

```
open(nombre);
```

El nombre del open, se lo va a dar el sistema lógico, ya que este asocia el nombre al dispositivo (con la syscall mknod).

comandos2.md ---> mknod

**mknod** genera el **file descriptor**: Es el número que le pones al write.

Normalmente se hace:

```
int fd = open("path_al_fichero");
write(fd, "hola\n", 5);
close(fd);
```

Para añadir un dispositivo nuevo puedes:

- Recompilar el kernel
- Añadir dispositivos sin recompilar (solo si es sistema lo permite)

## Estructura

Un fichero con un device driver para un dispositivo "inventado". Para no tener que recompilar el kernel, insertamos el driver usando un módulo.

```
// Estructura que define las operaciones que soporta el driver
struct file_operations fops_drive_1 = {
    owner: THIS_MODULE,
    read: read_drive_1
};

//En este caso, solo operacion lectura (read)
int read_drive_1(struct file* file, char user* buffer, size_t s, loff_t *off) {
    ...
    return size;
}

//Operaciones para cargar/eliminar el driver del kernel
static int _init driver1_init(void) {
    ...
}
```

```
static void _exit driver1_exit(void){
    ...
}

//Operacions del mòdul de carregar/eliminar el driver del kernel
module_init(driver1_init);
module_exit(driver1_exit);
```

## Contenido del DD (Device Driver)

- **Información general** del DD (nombre, autor...)
- Implementación de las **funciones genéricas**
  - open
  - read
  - write
  - ...
- Funciones de **inicialización**
- Funciones de **desinstalación**

## Pasos a seguir

A nivel **Físico**:

- Compilar el DD en .ko
- **Instalar las rutinas** del driver (se decide el mayor y menor)

A nivel **Lógico**:

- Crear un dispositivo lógico (con mknod)

A nivel **Virtual**:

- Crear el dispositivo virtual (con open)

## Llamadas al sistema

- Escribir por pantalla ---> **no** bloqueante
- Escribir una pipe ---> **no** bloqueante
- Escribir al disco ---> **bloqueante**

Crida a sistema	Descripció
open	Donat un nom de fitxer i un mode d'accés, retorna el nombre de canal per poder accedir
read	<b>Llegeix N bytes d'un dispositiu</b> (identificat amb un nombre de canal) i ho guarda a memòria
write	Llegeix N bytes de memòria i els <b>escriu al dispositiu</b> indicat (amb un nombre de canal)
close	Allibera el canal indicat i el deixa lliure per ser reutilitzat.
dup/dup2	Duplica un canal. El nou canal fa referència a la mateixa entrada de la TFO que el canal duplicat.
pipe	Crea un dispositiu tipus pipe llesta per ser utilitzada per comunicar processos
lseek	Mou la posició actual de L/E (per fitxers de dades)

## OPEN

```
int fd = open("path_al_fichero", modo_de_acceso, [flags]);
/*
Path -> dirección al fichero
Modo de acceso:
    - O_RDONLY -> solo lectura
    - O_WRONLY -> solo escritura
    - O_RDWR -> lectura y escritura
*/
```

### Abrir un fichero:

```
ret = open("ruta/nombre", modo, flags);
//flags = 0 aquí
//Devuelve:
    //ret = -1 error
    //ret = canal
```

### Crear un fichero:

```
ret = open("ruta/nombre", O_CREAT | O_WRONLY | O_RDWR | O_TRUNC, permisos);
//2o parámetro:
    //o_trunc -> (si existe)se sobrescribe
    //o_creat -> creat("ruta/nombre", permisos);
//3r parámetro:
    //user, grupo, otros
    //rwx, rwx, rwx
    //mejor empexar por 0
```

### Preguntar existencia fichero:

```
ret = open("ruta/nombre", O_CREAT | O_EXCL, permisos);
//si existe, ret = 1 i errno = EEXIST
```

### Creación (diapositivas):

- ficheros especiales tienen que existir antes de acceder a ellos
- especificar los `permission_flags` (or de `S_IRWXU`, `S_IRUSR...`)
- crear ---> añadir `O_CREAT` (con or de bits) en `acces_mode`
- no hay llamada para eliminar algunos datos de un fichero, solo podemos borrarlos todos.
- cortar el contenido de un fichero: `O_TRUNC`

```
//si no existia, lo crea
open("X", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR)

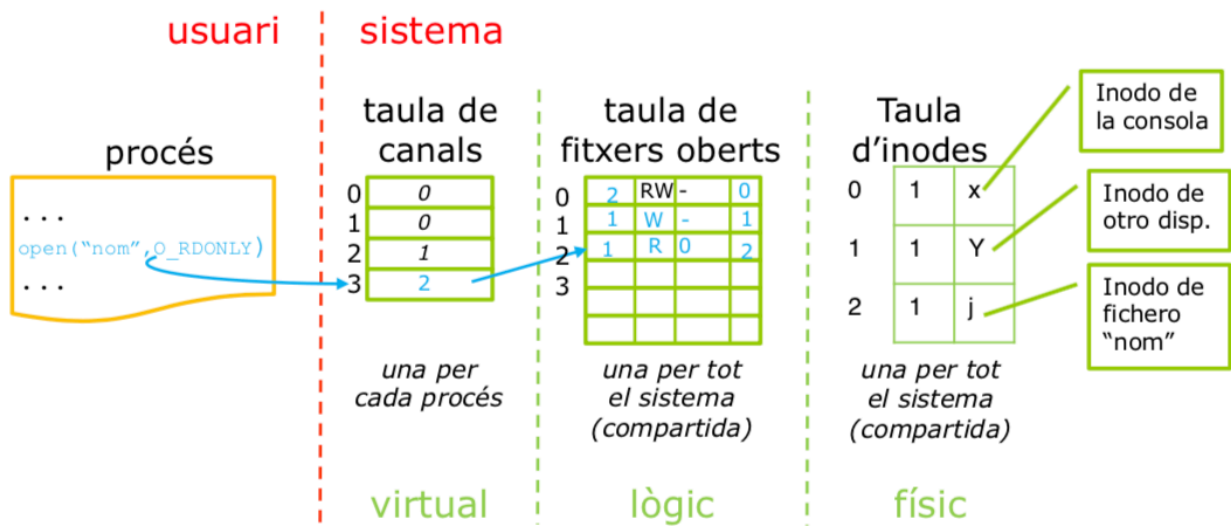
//si no existia, lo crea. else, libera datos y tamaño = 0;
open("X", O_RDWR | O_CREAT | O_TRUNC, S_IRWXU)
```

### Estructuras de datos:

`Open` (cont);

Def Efecto sobre estructuras de datos internas del sistema.

- Ocupa un canal libre de TC. (1º disponible)
- Ocupa una nueva entrada de TFO: (Pos L/E=0)
- Asocia las estructuras al DD que le toca (major del nombre simbólico).
- diferentes TFO pueden apuntar al mismo DD



## READ

```
int fd = open(...);
char buffer[64] // del tamaño que quieras
int num_bytes_leídos = read(fd, buffer, num_bytes_a_leer); // bytes a leer que quieras
```

Pide la lectura de num\_bytes\_a\_leer:

- Si hay suficientes ---> los lee.
- Si hay menos de los que pide ---> lee los que hay.
- Si no hay nada que se puede leer ---> depende del dispositivo (se bloquea, o devuelve 0)

Devuelve:

- -1 ---> error
  - 0 ---> nada que leer
  - >0 ---> bytes leídos
- La posición por la cual debería seguir leyendo después, se avanza automáticamente (lo hace el kernel) tantas posiciones como bytes haya leído

## WRITE

```
n = write(fd, buffer, count);
```

- **n**: num de bytes escritos
- **fd**: num del canal
- **buffer**: puntero de memoria de donde copiar los datos
- **count**: num de bytes a escribir

Funcionamiento:

Pide la escritura de count chars al dispositivo asociado a fd.

- Si hay espacio para count ---> escribe count.
- Si hay menos que count ---> escribe los que quepan.
- Si no hay espacio ---> depende de dispositivo (bloqueo, return 0, ...).

La posición l/e avanza automáticamente (kernel en write) tantas posiciones como bytes escritos

- Posición\_l/e = posición\_l/e + num\_bytes\_escritos

Devuelve:

- -1 ---> si hay error
- **>= 0** ---> Los bytes que ha escrito realmente

- La posición por la cual debería seguir escribiendo después, se avanza automáticamente (lo hace el kernel) tantas posiciones como bytes haya escrito.

## DUP

```
newfd = dup(fd);
```

**Def** Hace una copia del canal que le pases como parámetro.

Explicación:

- Hace la copia en la primera entrada libre de la Tabla de Canales. Se comparten los punteros de lectura y escritura.
- Es decir, si yo escribo algo en fd, si luego sigo escribiendo en newfd continuará por donde lo he dejado con el fd antiguo (no se ha sobrescrito).

Devuelve:

- -1 ---> error
- **> 0** ---> canal nuevo

## DUP2

```
newfd = dup2(fd, newfd);
```

**Def** Cierra el canal `newfd` si estaba abierto y se duplica el canal `fd` en `newfd`.

Lo hace de forma atómica (en una sola instrucción).

```
//lo que significa que...
dup2(viejo, nuevo);

//es diferente de...
close(nuevo);
nuevo = dup(viejo);
```

Ya que dup2 es una sola instrucción (siempre hace las dos cosas o no hace ninguna).

## CLOSE

```
close(fd);
```

Funcionamiento

- Se elimina el canal fd de los niveles inferiores
- Es posible que haya varias referencias al canal por lo tanto cuando haces close el contador de referencias se decrementa por 1. Lo mismo para la lista de inodos.
- Si el número de referencias es 0, se elimina de la Tabla de Ficheros Abiertos, lo mismo en la Tabla de Inodos.

## FORK

```
int ret = fork();
```

Funcionamiento:

- Crea un proceso hijo (nada nuevo, no?). El hijo hereda todo el PCB... Eso incluye la Tabla de Canales del padre.

- Permite compartir el acceso a dispositivos abiertos antes del fork.
- Los open que hagamos después ya serán independientes.

## EXEC

```
int ret = exec("path", arg0, arg1...);
```

Funcionamiento:

- Cambia el binario a ejecutar por el proceso, pero la Tabla de Canales se mantiene (y por lo tanto también la Tabla de Ficheros Abiertos).

## PIPE

Explicación:

Implementa un **buffer temporal** con funcionamiento FIFO. Los datos de la pipe se borran a medida que se leen.

---

---> METO DATOS ---> LEO DATOS

---

Sirven para **compartir información** entre procesos.

- Pipe **con nombre**, cualquiera que tenga permiso puede acceder al dispositivo

```
mknod p nombre
```

```
open(nombre, R);
open(nombre, W);

// Es importante abrir por separado
// para que se puedan cerrar por separado.

// Si haces:
open(nombre, RW);
// Enhorabuena, has suspendido.
```

- Pipe **sin nombre**, solo accesible via herencia (eso significa que solo funcionan entre padre - hijo).

```
int fd_vector[2];
int ret = pipe(fd_vector);
/*
    fd_vector[0] --> será el canal de lectura
    fd_vector[1] --> será de escritura
*/
fork();
```

La pipe crea un **inodo virtual** en la Tabla de Inodos y **dos filas** en la Tabla de Ficheros de Abiertos, **una de escritura y una de lectura**.

El hijo escribe en el canal de escritura y el padre lee del canal de lectura (o viceversa). Magic.

Pero que pasa: que el padre no sabe cuando el hijo ha escrito algo y por lo tanto no sabría cómo leer. Por eso, es mejor hacer 2 pipes:

---

PIPE PETICIONES ---->

---



---

## <---- PIPE RESPUESTAS

---

Si leo de la pipe:

```
read(pipe,...);
```

Devuelve:

- Si hay error ---> -1
- Si hay datos ---> retorna **bytes leídos** (tantos como ha podido, según el parámetro 3º del read).
- Si no hay datos
  - Si hay escritores (algún proceso con pipe[1]) ---> se **BLOQUEA** (hasta que haya datos o hasta que pipe[1] esté cerrado totalmente).
  - Si no hay escritores ---> 0

### CERRAR SIEMPRE LOS CANALES QUE NO UTILICEMOS.

Ejemplo: si el padre escribe y el hijo lee, que el hijo cierre el canal de escritura y el padre el canal de lectura.

Si escribo en la pipe:

```
write(pipe,...);
```

- Si hay error ---> -1
- Si hay espacio ---> **escribe** (con límite del 3º parámetro)
- Si no hay espacio ---> se **BLOQUEA** (hasta que haya espacio).
- Si no hay lectores ---> recibo el SIGPIPE (default: **acabar**)

Dependiendo del fabricante, se bloquea hasta que le cabe toda la consigna o se bloquea hasta que haya algún byte libre.

## LSEEK

Def La posición de l/e (lectura/escritura) se modifica manualmente por el usuario. Permite hacer accesos directos a posiciones concretas de ficheros de datos (o ficheros de dispositivos con accesos secuenciales).

Como:

- Se inicializa a 0 open
- Se incrementa automáticamente al hacer read/write
- Lo podemos mover manualmente con LSEEK

```
new_pos = lseek(fd, desplazamiento, SEEK_...)
```

SEEK\_...: (desplazamiento puede ser negativo)

- SEEK\_SET: desde inicio del fichero `posicio_l/e=desplaçament`
- SEEK\_CUR: desde posición actual P(l/e) `posicio_l/e=posicio_l/e+desplaçament`
- SEEK\_END: desde final fichero (end of file) `posicio_l/e=file_size+desplaçament`

## VIRTUAL FILE SYSTEM

### Disco inmanejable

1. Acceso a datos compartimentados (nombre)

## 2. Organización -> directorio

### Directorio

Fichero especial (no accesible mediante syscalls) Liga nombres -> contenido

ls -l :

tipo	permisos	nº links	usuario	grupo	...	nombre
d						
p						
s						
...						

Nombres e inodos:

nombre	inodo
.	ref a él mismo
..	ref al padre
fichero.txt	7
directorio	25

. y .. : generar jerarquía (árbol invertido)

Punto de entrada ---> RAÍZ ( / ) (inodos: tanto . como .. valen 0 en ubuntu)

- Acceso absoluto: empieza por / . /(....)/(....)/(....)/(....).txt
- Acceso relativo: a partir de CWD (current working directory) ---> nunca empieza por / . ./(....) , ../(....)

### Directorios en grafo

Def Árbol de directorios + nombre ---> GRAFO (cíclico)

- **Hardlinks:** acceso es a contenido (fichero) -> acíclico
- **Softlinks:** acceso es a otro nombre (acceso directo) -> cíclico

nombre	inodo	// Comentarios
.	ref a él mismo	---
..	ref al padre	---
A	7	Hardlink
B	7	Hardlink
C	25	Softlink

### DESCRIPTOR FICHERO: inodo

- Tipo: c/b/p/l... (soft)
- Tamaño
- Nº enlaces ---> nº Hardlinks
- Atributos (rwx)
- Acceso a ops dependiente

### BORRADO DE FICHERO, BORRADO DE HL

- Acceder al inodo a partir del nombre

- #enlaces (inodo): Si es 0, se borra inodo, contenido y nombre.

## DISCOS ORGANIZADOS EN PARTICIONES

/FAT32 pendrive  
/ISO9660 DVD/CD

- **Sector:** unidad de transferencia del fabricante. (512, ...) [foto ari 1]
- **Bloque:** unidad de transferencia del sistema operativo
- **Particiones:** VFS(virtual file system) [foto ari 2] Los discos se montan en puntos a partir de la raíz global

## PARTICIONES EN LINUX

- **SuperBloque:**
  - Metadatos del SF
    - formateado
    - joliet
    - gestión de espacio de la partición
      - lista inodos libres/ocupados
      - lista bloques de datos libres/ocupados
      - referencias inodos
    - tamaño
    - ...
  - Sección de Inodos
    - Lista de inodos reservados en la partición (ocupados o no)
    - Todos los inodos de SF (/ es el inodo 0)
  - Sección de datos
    - Bloques con contenido ficheros

Resumen:

- Metadatos ---> SuperBloque
- Descriptores inodos ---> disp, directorio, fichero...
- Datos
  - contenido de ficheros
  - contenido de directorios (espacios de nombres jerarquía)

---	Apunta al inodo
.	0
..	0
A	1
B	2
C	3

## ASIGNACIÓN DE ESPACIO A DATOS

Hay dos formas de hacerlo:

- **Contiguo:** (CD/DVD) ---> solo lectura (joliet, ISO 9660)

F1	F2	F3	F4	...
----	----	----	----	-----

- **NO Contiguo:** ---> con un fichero FREE. ????

FAT (File access table)	Fichero 1	Fichero 2	Fichero 3	...
-------------------------	-----------	-----------	-----------	-----

## LINUX (UNIX)

[foto diapositiva 1.91]

nombre – inodo (en lista de inodos) :

**tamaño inodo** = tamaño bloque (4kB) \* 10 enlaces directos a datos \* 1 enlace indirecto \* 1 enlace indirecto doble \* 1 enlace indirecto triple

[foto diapositiva 1.94]

## ESTRUCTURAS EN MEMORIA PARA REDUCIR ACCESOS A DISCO

- Tabla de ficheros abiertos ---> modo + Puntero l/e
- Tabla de inodos ---> caché de lista de inodos del disco ---> inodos en uso
- Buffer caché (cache de bloques) ---> almacena bloques (inodos y bloques)

## Sistema de ficheros

Jerarquía: DIRECTORIOS

Grafo (nombres):

- **Hardlinks:** si apuntan al mismo inodo (desde entradas de directorio).
- **Softlinks:** fichero marcado como "link" (marca en inodo) y su contenido (bloque de datos) es la ruta a otro fichero.

Boot	SuperBloque	Inodos	Bloques de datos
------	-------------	--------	------------------

+Ficheros/dispositivos

Raíz Directorio ---> / (inodo 0)

Directorio ---> 1 Inodo + 1 (al menos) Bloque de DATOS

Nombre	Inodos
.	0
..	0
A	1
B (hardlink)	1
SL (softlink)	8
...	...

Dispositivo ---> 1 Inodo

Ficheros "normales"

- 1 Inodo + 0 ó más BLOQUES DE DATOS

ESTRUCTURA INODO:

- Info Admin:
  - Tamaño
  - Fecha de creación
  - Propietario
  - ...
- Enlaces a llamadas DEP

- Si NORMAL o DIRECTORIO
- Enlaces a bloques de datos

#### Disco Sistema de Ficheros

SuperBloque	Inodos	Bloques de datos
-------------	--------	------------------

Proceso: (TC)

En el SO:

- TFA (open/close)
- T.inodos (caché inodos de ficheros en uso)
- Buffer-cache: cache de bloques (Inodos + bloques)

## Syscalls, estructuras de datos y accesos al disco

### OPEN

- Ruta: /home/alumne/S7/a.c
- Navegación (con el siguiente ejemplo): Inodo / + bloque / + inodo 8 + bloque / HOME

Nombre	Inodos
.	0
..	0
...	...
HOME	8

Open: acaba al traer a memoria el inodo de a.c .

**No carga bloques de datos.**

Mete el inodo en la TI (si no estaba).  
 + 1 entrada en la TFA (según parámetros)  
 + 1 entrada en la TC

### CREAR

- Ruta: /home/alumne/ ~~b.e~~
1. Crear un nuevo inodo
  2. Añadirlo al directorio
  3. Modificar el bloque de datos del directorio y el tamaño del directorio
  4. Modificar el inodo del directorio

**Actualizar salvo que esté en disco**

### READ

ret = read(\_\_\_, \_\_\_, long)

(tr. a bloques) + P(l/e) ----> P(l/e) agumenta en ret

// TO-DO: ejemplos (relación syscalls - estructuras de datos) diapo 1.102

### LSEEK

**No afecta a disco.**

**Lseek solo cambia el puntero de l/e en tabla de ficheros abiertos (TFA)**

