T4-Entrada/Salida

Licencia

Índice

- Conceptos básicos de E/S
- Dispositivos: Virtual, lógico y físico
- Gestión de la E/S en Linux
  - Estructuras de datos del kernel
  - Llamadas a sistema básicas
  - Ejemplos
- Sistema de Ficheros
  - Linux: Relación entre llamadas a sistema / estructuras de datos

CONCEPTOS BÁSICOS DE E/S

¿Qué es la E/S?

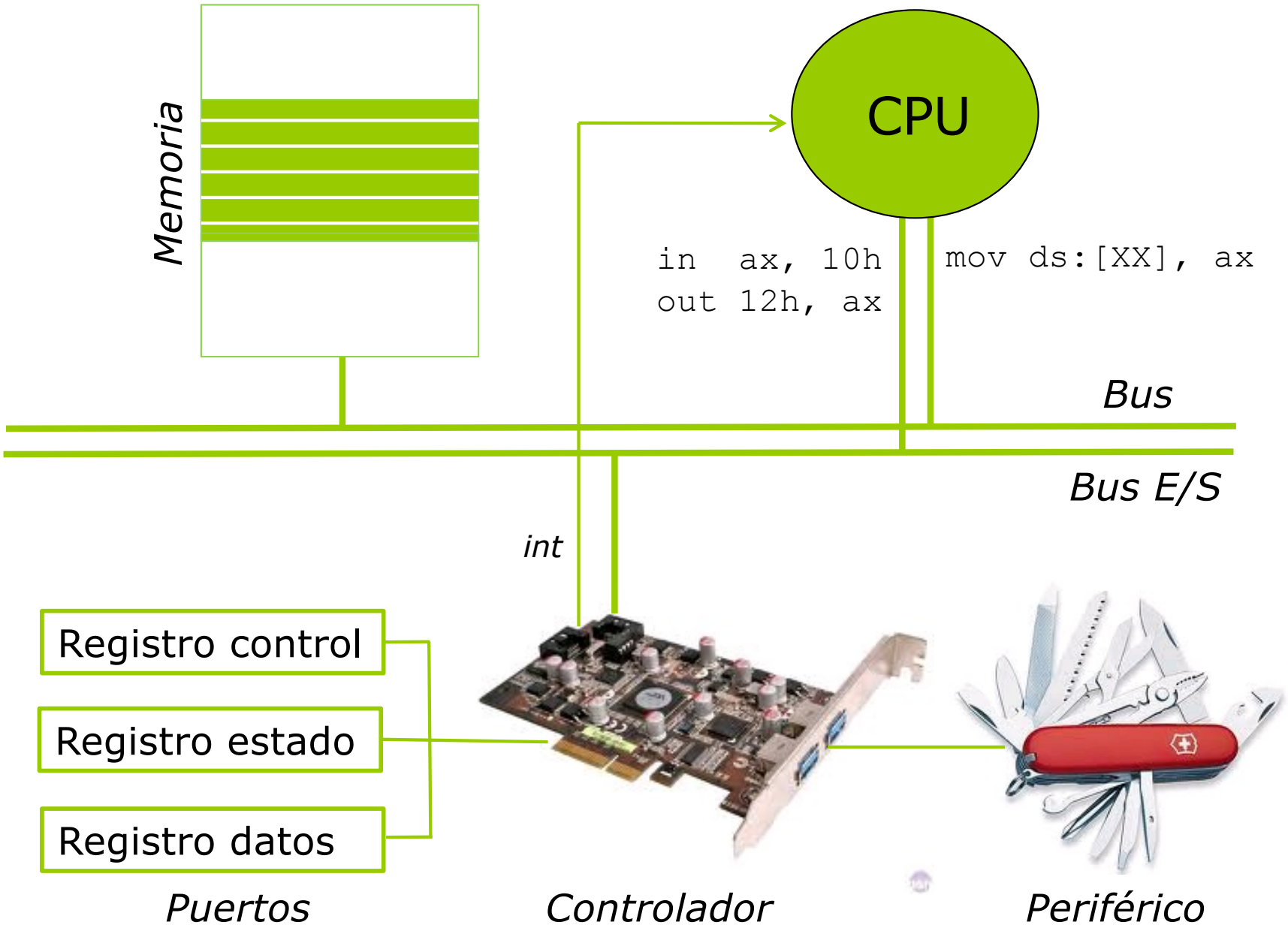
- **Definición**: transferencia de información entre un proceso i el exterior
  - Entrada de datos: del exterior al proceso
  - Salida de datos: del proceso al exterior

  (siempre desde el punto de vista del proceso)

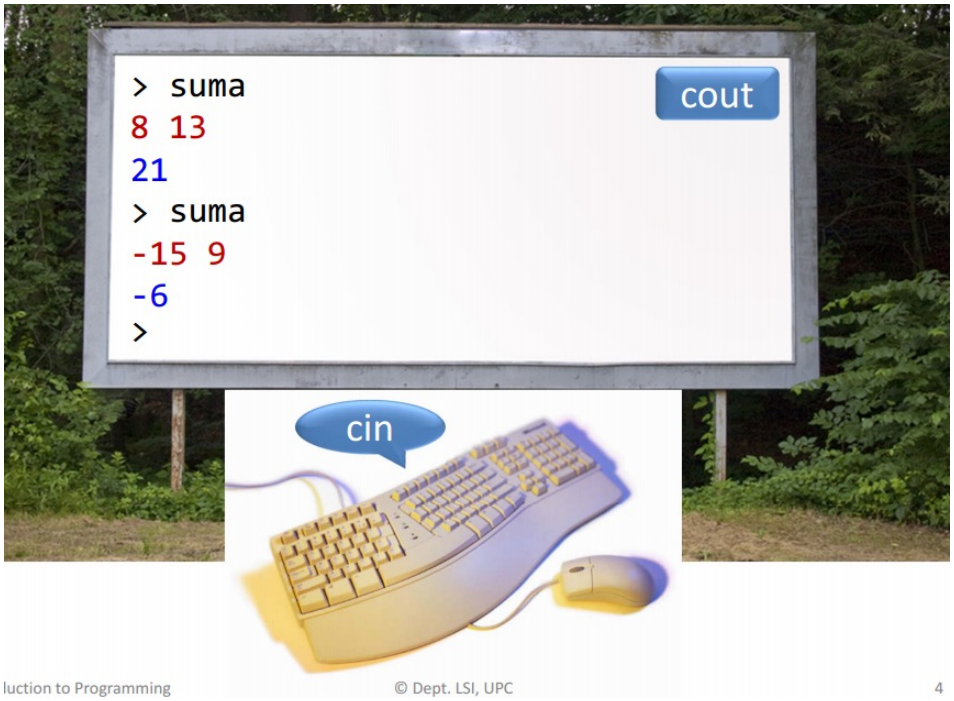
- De hecho, básicamente, los procesos realizan cálculo y E/S
- Muchas veces, incluso, la E/S es la tarea principal de un proceso: p.ej. navegación web, intérprete de comandos, procesador de textos
- **Gestión del E/S:** administrar el funcionamiento de los dispositivos de E/S (periféricos) para un uso **correcto**, **compartido** y **eficiente** de los recursos

Dispositivos de E/S

Visión HW: Acceso a los dispositivios de E/S

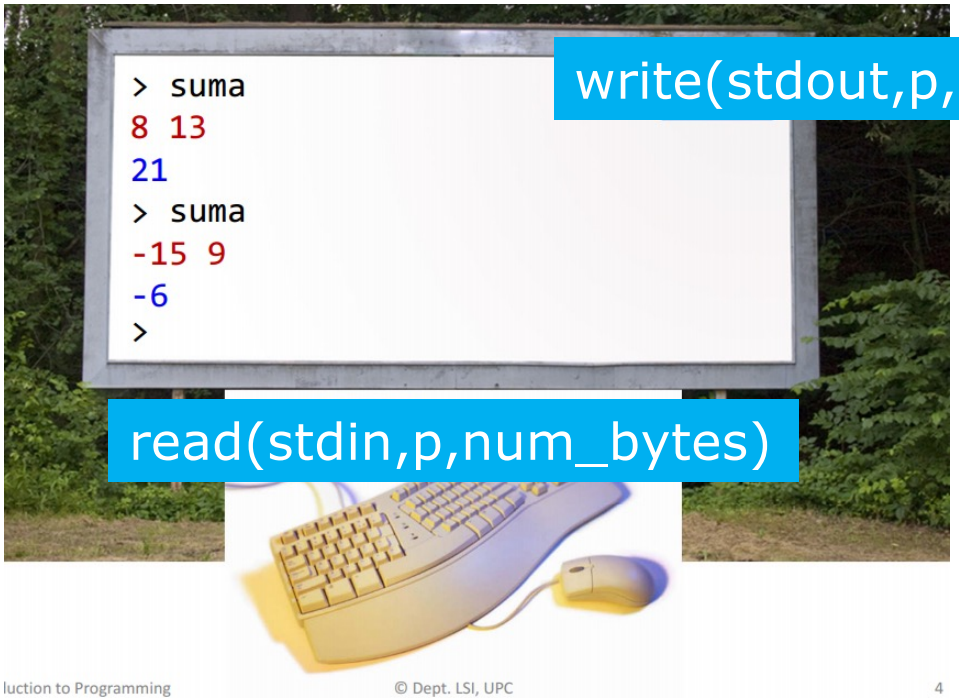


Visión usuario (hasta ahora) (PRO1)



Entrada: **cin**. Lee los datos y los procesa para adaptarlos al tipo de datos
Salida: **cout**. Procesa los datos y los escribe

En este curso veremos que hay en el medio



- No será tan fácil como en C o C++
- La conversión de tipo de datos es responsabilidad del usuario (es lo que hacen las librerias de C o C++)

DISPOSITIVOS:VIRTUAL, LÓGICO, FÍSICO

Tipos de dispositivos

- Dispositivos de interacción con el usuario (pantalla, teclado, mouse), almacenamiento (disco duro, DVD, pen drive), transmisión (módem, red con cable, red sin cable) o incluso otros más especializados (controlador de un avión, sensores, robots) … ¡muchísima variedad!
- Clasificación según:
  - Tipo de dispositivo: lógico, físico, de red
  - Velocidad de acceso: teclado vs disco duro
  - Flujo de acceso: mouse (byte) vs DVD (bloque)
  - Exclusividad de acceso: disco (compartido) vs impresora (dedicado)
  - Otras…
- Dilema: estandarización vs nuevos tipos de dispositivos

CONCEPTO: Independencia de los dispositivos

Independencia: principios de diseño

- El objetivo es que los procesos (código principalmente) sean independientes del dispositivo que se está accediendo
- Operaciones de E/S **uniformes**
  - Acceso a todos los dispositivos mediante las mismas llamadas a sistema
  - Aumenta la simplicidad y la portabilidad de los procesos de usuario
- Utilizando **dispositivos virtuales**
  - El proceso no especifica concretamente sobre qué dispositivo trabaja, sino que utiliza unos identificadores y existe una traducción posterior
- Con posibilidad de **redireccionamiento**
  - El sistema operativo permite a un proceso cambiar la asignación de sus dispositivos virtuales

```
% programa < disp1 > disp2
```

Independencia: principios de diseño

- Por eso, habitualmente, diseño en tres niveles: **virtual**, **lógico** y **físico**
- El primer nivel nos aporta independencia, **el proceso trabaja con dispositivos virtuales** y no necesita saber qué hay detrás
- **El segundo nivel nos aporta compartición de dispositivos**. Diferentes accesos concurrentes en el mismo dispositivo.
- De esta forma se pueden escribir programas que realicen E/S sobre dispositivos (virtuales) sin especificar cuáles (lógico).
- En el momento de ejecutar el programa se determina dinámicamente sobre qué dispositivos se trabaja
  - Puede ser un parámetro del programa, puede ser “heredado” de su padre,...
- El tercer nivel separa las operaciones (software) de la implementación. Este código es de muy bajo nivel, muchas veces en ensamblador

Dispositivo virtual

- **Nivel virtual**: Aísla al usuario de la complejidad de gestión de los dispositivos físicos
  - Establece correspondencia entre nombre simbólico (nombre de archivo) y la aplicación de usuario, a través de un dispositivo virtual
    - Un nombre simbólico es la representación en el sistema de un dispositivo
    - /dev/dispX o bien .../dispX un nombre de fichero
      - Un dispositivo virtual representa un dispositivo en uso de un proceso
      - Dispositivo virtual = canal = descriptor de fichero. Es un número entero
      - Los procesos tienen 3 canales :
        - Entrada canal 0 (stdin)
        - Salida canal 1 (stdout)
        - Salida de error canal 2 (stderr)

  - Las llamadas a sistema de transferencia de datos utilizan como identificador el dispositivo virtual

Dispositivo lógico

- **Nivel lógico**:
  - Establece correspondencia entre dispositivo virtual y dispositivo (¿físico?)
  - Gestiona dispositivos que pueden tener, o no, representación física
  - P.ej. Disco virtual (sobre memoria), dispositivo nulo
  - Manipula bloques de datos de tamaño independiente
  - Proporciona una interfaz uniforme al nivel físico
  - Ofrece compartición (acceso concurrente) a los dispositivos físicos que representan (si existen)
  - En este nivel se tienen en cuenta los permisos, etc
  - En Linux se identifican con un nombre de archivo

Dispositivo físico

- **Nivel físico**: Implementa a bajo nivel las operaciones de nivel lógico
  - Traducir parámetros del nivel lógico a parámetros concretos
    - P.ej. En un disco, traducir posición L/E a cilindro, cara, pista y sector
  - Inicializa dispositivo. Comprueba si libre; de lo contrario pone la petición en cola
  - Realiza la programación de la operación pedida
    - Puede incluir examinar el estado, poner motores en marcha (disco), …
  - Espera, o no, a la finalización de la operación
  - Devuelve los resultados o informa sobre algún error
  - En Linux, un dispositivo físico se identifica con tres parámetros:
    - Tipo: **Block/Character**
    - Con dos números: major/minor
      - **Major**: Indica el tipo de dispositivo
      - **Minor**: instancia concreta respecto al major

Device Drivers

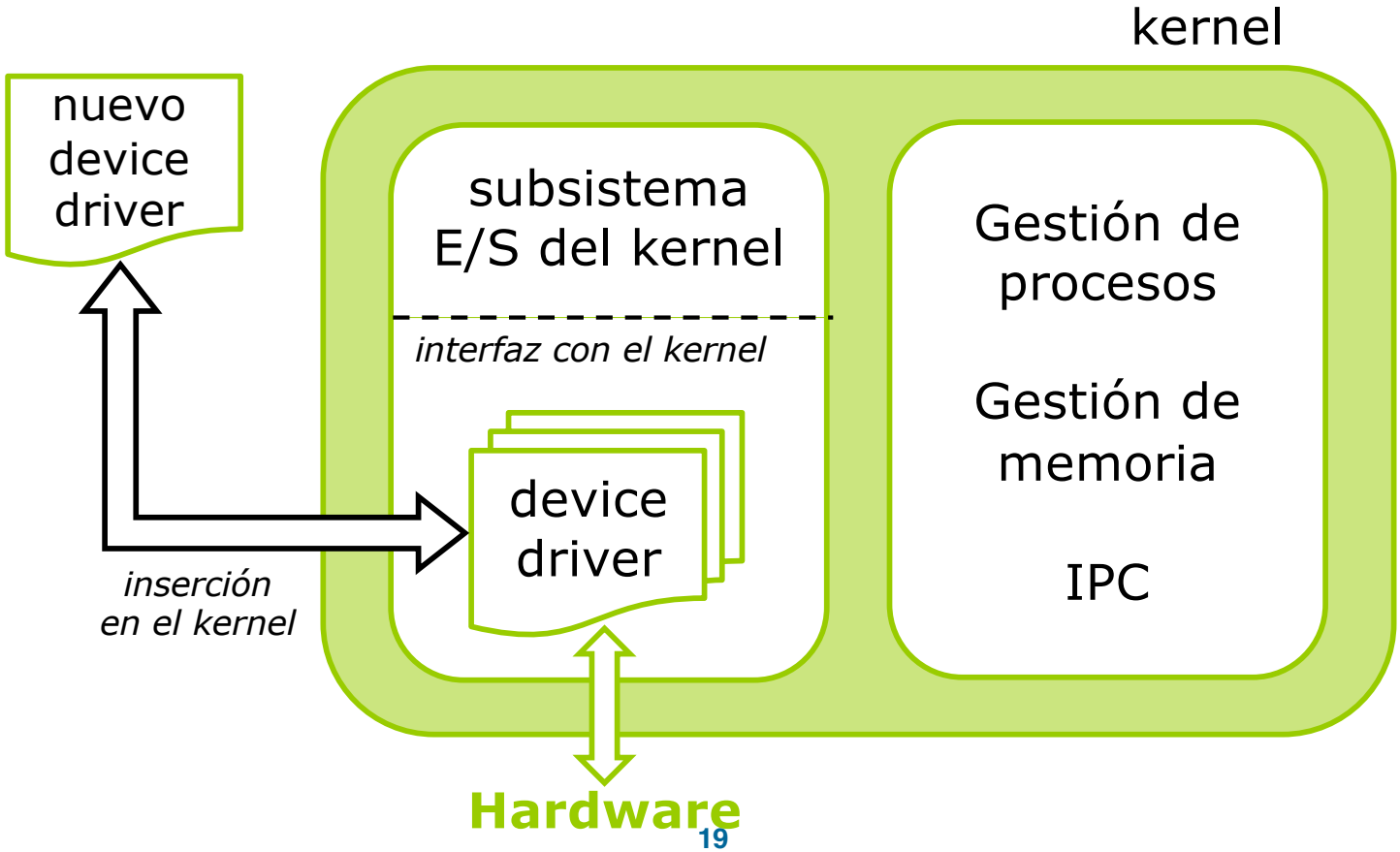
- Para ofrecer independencia, se define el conjunto de operaciones que ofrecen los dispositivos
  - Es un superconjunto de todas las operaciones que se pueden ofrecer para acceder a un dispositivo físico
  - No todos los dispositivos ofrecen todas las operaciones
  - Cuando se traduce de disp. virtual a lógico y físico, de esta manera está definido qué operaciones corresponden

Device drivers

- Los programadores del SO no pueden generar códigos para todos los dispositivos, modelos, etc
- Es necesario que el fabricante proporcione el conjunto de rutinas de bajo nivel que implementan el funcionamiento del dispositivo
  - Al código + datos para acceder a un dispositivo se le conoce como Device driver (controlador de dispositivo)
  - Siguiendo la especificación del interfaz de acceso a las operaciones de E/S definida por el sistema operativo
- Para añadir un nuevo dispositivo podemos:
  1. Opción 1: Recompilar el kernel
  2. Opción 2: Añadir nuevos dispositivos sin recompilar el kernel
     - Es necesario que el SO ofrezca un mecanismo para añadir dinámicamente código/datos al kernel
       - Dynamic Kernel Module Support, Plug & Play

Device Driver

- Se identifican unas operaciones comunes (interfaz) y las diferencias específicas se encapsulan en módulos del SO: ***device driver***
  - Aísla al resto del kernel de la complejidad de la gestión de los dispositivos
  - Además, protege al kernel frente a un código escrito por "otros"



Device Driver

- **Controlador de dispositivo (en genérico)**: Conjunto de rutinas que gestionan un dispositivo, y que permiten a un programa interactuar con ese dispositivo
  - Respetan el interfaz definido por el SO (abrir, leer, escribir, ...)
    - Cada SO tiene definida su propio interfaz
  - Implementan tareas dependientes del dispositivo
    - Cada dispositivo realiza tareas específicas
  - Contienen habitualmente código de bajo nivel
    - Acceso a los puertos de E/S, gestión de interrupciones, ...
  - Quedan encapsuladas en un archivo binario

Inserción dinámica de código: Módulos de Linux

- Los kernels actuales ofrecen un mecanismo para añadir código y datos al kernel, **sin necesidad de recompilar** el kernel
  - Una recompilación completa del kernel podría tardar horas...
- Actualmente la inserción se realiza dinámicamente (en tiempo de ejecución)
  - ***Dynamic Kernel Module Support*** (linux) o Plug&Play (windows)
- Mecanismo de módulos en Linux
  - Archivo(s) compilados de forma especial que contienen código y/o datos para añadir al kernel
  - Conjunto de comandos para añadir/eliminar/ver módulos
  - Lo veremos en el laboratorio

Qué tendremos en el lab…MyDriver1.c

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

```
struct file_operations fops_driver_1 = {
                owner: THIS_MODULE,
                read: read_driver_1,
};
```

Esta estructura define las operaciones que ofrece el driver

```
int read_driver_1 (struct file *f, char __user *buffer, size_t s, loff_t *off) {
…
return size;

}
```

En este caso, solo la operación de lectura (read)

```
static int __init driver1_init(void){
…
}
static void __exit driver1_exit(void){
…
}
```

Operaciones para cargar/eliminar el driver del kernel

```
module_init(driver1_init);
module_exit(driver1_exit);
```

Operaciones del módulo de carga/descarga

Device Driver (DD) en linux: +detalle

- Contenido de un *Device Driver* (DD)

  - **Información general sobre el DD**: nombre, autor, licencia, descripción,...
  - **Implementación de las funciones genéricas de acceso a los dispositivos**
    - open, read, write, ...
  - **Implementación de las funciones específicas de acceso a los dispositivos**
    - Programación del dispositivo, acceso a los puertos, gestión de las ints, ...
  - Estructura de datos con **lista de apuntadores a las funciones específicas**
  - **Función de inicialización**
    - Se ejecuta al instalar el DD
    - Registra el DD en el sistema, asociándolo a un major
    - Asocia las funciones genéricas al DD registrado
  - **Función de desinstalación**
    - Desinstala el DD del sistema y las funciones asociadas
- Ejemplo de DD: ver `myDriver1.c` i `myDriver2.c` en la documentación del laboratorio

Módulos de E/S en linux: +detalles

- Pasos para añadir y utilizar un nuevo dispositivo:

Físic

- **Compilar** el DD, en su caso, en un formato determinado: .ko (kernel object)
  - El tipo (block/character) y el major/minor están especificados en el código del DD
- **Instalar** (insertar) en tiempo de ejecución las rutinas del driver
  - insmod archivo_driver
  - Recordar que el driver está asociado a un major

Lògic

- **Crear un dispositivo lógico** (nombre de fichero) y vincularlo con el dispositivo físico
  - Nombre fichero ←→blockIcharacter + major y minor
  - **Comando** mknod
    - mknod /dev/mydisp c major minor

Virtual

- **Crear el dispositiu virtual (llamada a sistema)**
  - open("/dev/mydisp", ...);

Ejemplos de dispositivos (1)

- Funcionamiento de algunos dispositivos lógicos: terminal, fichero, pipe, socket

1. **Terminal**
   - Objeto a nivel lógico que representa el conjunto teclado+pantalla
   - “Habitualmente” los procesos lo tienen como entrada y salida de datos
2. **Fichero de datos**
   - Objeto a nivel lógico que representa información almacenada en disco. Se interpreta como una secuencia de bytes y el sistema gestiona la posición en la que nos encontramos dentro de esta secuencia.

Ejemplos de dispositivos(2)

- **Pipe**
  - Objeto a nivel lógico que implementa un buffer temporal con funcionamiento FIFO. Los datos de la pipe se borran a medida que se van leyendo. Sirve para intercambiar información entre procesos
    - Pipe **sin nombre**, conecta sólo procesos con parentesco ya que sólo es accesible vía herencia
    - Pipe **con nombre**, permite conectar cualquier proceso que tenga permiso para acceder al dispositivo
- **Socket**
  - Objeto a nivel lógico que implementa un buffer temporal con funcionamiento FIFO. Sirve para intercambiar información entre procesos que se encuentren en diferentes ordenadores conectados por alguna red
  - Funcionamiento similar a las pipas, aunque la implementación interna es mucho más compleja puesto que utiliza la red de comunicaciones

GESTIÓN DE LA E/S EN LINUX

Tipos de ficheros en Linux

- En Linux, todos los dispositivos se identifican con un archivo (que puede ser de diferentes tipos)
  - block device
  - character device
  - Directory
  - FIFO/pipe
  - Symlink
  - **regular file → Son los ficheros de datos o "ordinary files" o ficheros "normales"**
  - Socket
- Llamaremos ficheros "especiales" a cualquier archivo que no sea un archivo de datos : pipes, links, etc

Creación de ficheros en Linux

- La mayoría de tipos de ficheros se pueden crear con la llamada a sistema/comando mknod
  - Excepto ficheros regulares, directorios y soft links
  - En el laboratorio utilizaremos el comando, que **no sirve para crear ficheros de datos**
- mknod nombre_fichero TIPO major minor
  - TIPO = c → Dispositivo de caracteres
  - TIPO = b → Dispositivo de bloques
  - TIPO = p → Pipe (no hace falta poner los parámetros major/minor)

Tipos de Ficheros

- Algunos ejemplos:

dev name	type	major	minor	description
`/dev/fd0`	b	2	0	floppy disk
`/dev/hda`	b	3	0	first IDE disk
`/dev/hda2`	b	3	2	second primary partition of first IDE disk
`/dev/hdb`	b	3	64	second IDE disk
`/dev/tty0`	c	3	0	terminal
`/dev/null`	c	1	3	null device

ESTRUCTURAS DE DATOS DEL KERNEL

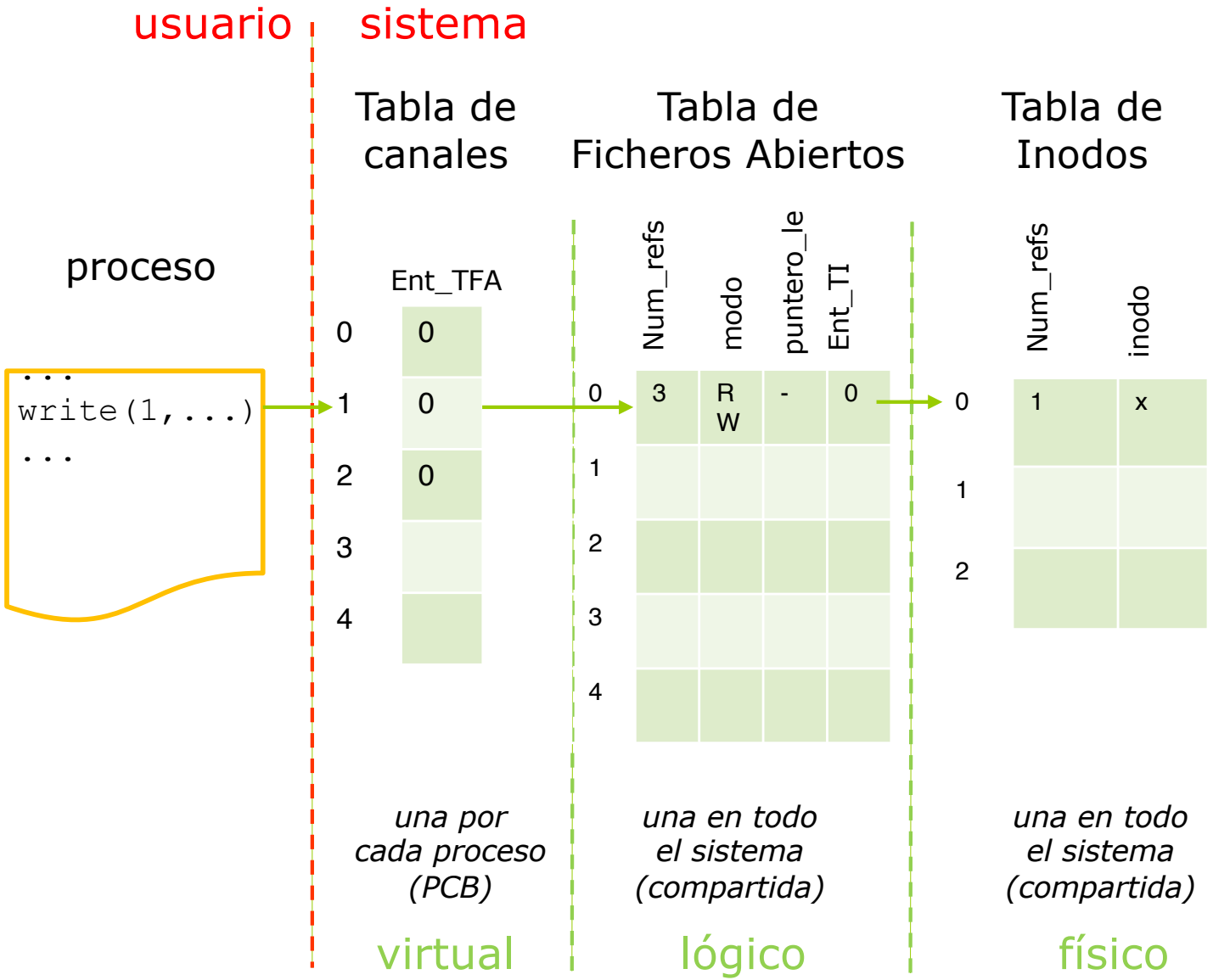
Estructuras de datos del kernel: Inodo

- Estructura de datos que contiene toda la información relativa a un fichero:
  - tamaño
  - tipo
  - protecciones
  - propietario y grupo
  - fecha del último acceso, modificación y creación
  - número de enlaces al inodo (número de nombres de archivos que apuntan a…)
  - índices a los datos (indexación multinivel) → lo veremos al final del tema, relacionado con la gestión de la información del disco
- Está toda la información de un fichero excepto el nombre, que está separado
- Se almacena en disco, pero existe una copia a memoria para optimizar el tiempo de acceso

Estructuras de datos del kernel

- Por proceso
  - Tabla de canales (TC): por cada proceso (se guarda en el PCB)
    - Indica a qué archivos estamos accediendo.
    - Se accede al archivo mediante el canal, que es un **índice en la TC**
    - El canal es el disp. virtual
    - Un canal referencia a una entrada de la tabla de archivos abiertos (TFA)
    - Campos que asumiremos: `num_entrada_TFA`
- Global:
  - Tabla de ficheros abiertos (TFA):
    - Gestiona los archivos en uso en todo el sistema
    - Puede haber entradas compartidas entre más de un proceso y por varias entradas de la TC del mismo proceso
    - Una entrada de la TFA referencia a una entrada de la TI.
    - Campos que asumiremos:: `num_refs, modo , puntero_le, num_entrada_TI`
  - Tabla de inodos(TI):
    - Contiene información sobre cada objeto físico abierto, incluyendo las rutinas del DD
    - Es una copia en memoria de los datos que tenemos en el disco (se copia en memoria por eficiencia)
    - Campos que asumiremos: `num_refs, datos_inodo`

Estructuras de datos del kernel



LLAMADAS A SISTEMA BÁSICAS

Ops bloqueantes y no bloqueantes (1)

- Algunas operaciones pueden ser costosas en términos de tiempo, el proceso no puede estar consumiendo CPU sin hacer nada→ El SO bloquea al proceso (RUN→BLOCKED)
- Operación de E/S **bloqueante**: Un proceso realiza una transferencia de N bytes y se espera a que termine. Devuelve el número de bytes que se han transferido
  - Si los datos están disponibles al instante (aunque quizás no todos), se realiza la transferencia y el proceso retorna inmediatamente
  - Si los datos no están disponibles, **el proceso se bloquea**
    1. Cambio de estado de RUN a BLOCKED
       - Deja la CPU y pasa la cola de procesos que están esperando.
    2. Se pone en ejecución el primer proceso de la cola de preparados (si Round Robin)
    3. Cuando llegan los datos se produce una interrupción
       - La RSI recoge el dato y pone el proceso en la cola de preparados (Si Round Robin)
    4. Cuando le toque el turno, el proceso se pondrá de nuevo en ejecución

Ops bloqueantes y no bloqueantes (2)

- Operación de E/S **no bloqueante**
  - El proceso solicita una transferencia, dispone los datos que pueda en ese momento, y retorna inmediatamente tanto si hay datos como si no
  - Devuelve el número de datos transferidos

Operaciones básicas de E/S

Llamada a sistema	Descripción
open	Dado un nombre de archivo y un modo de acceso, devuelve el número de canal para poder acceder
read	**Lee N bytes de un dispositivo** (identificado con un número de canal) y lo guarda en memoria
write	Lee N bytes de memoria y los **escribe en el dispositivo** indicado (con un número de canal)
close	Libera el canal indicado y lo deja libre para su reutilización
dup/dup2	Duplica un canal. El nuevo canal apunta a la misma entrada de la TFA que el canal duplicado.
pipe	Crea un dispositivo tipo pipe lista para ser utilizada para comunicar procesos
lseek	Mueve la posición actual de L/E (para archivos de datos)

Las llamadas open, read y write pueden bloquear al proceso

Open

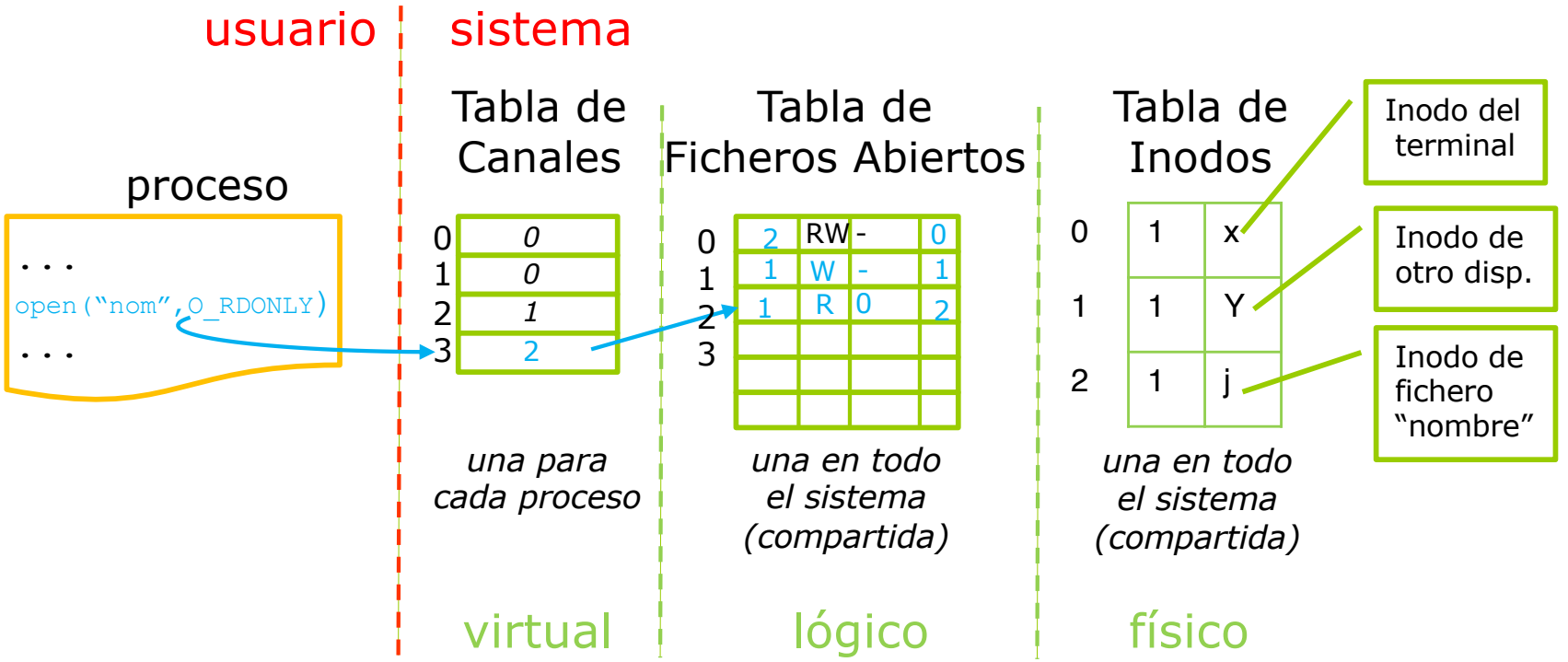
- Por tanto, ¿cómo se asocia un nombre a un dispositivo virtual?
- `fd = open(name, access_mode[,permision_flags]);`
  - Vincula nombre de fichero a dispositivo virtual (descriptor de fichero o canal)
    - Además, permite realizar una sola vez las comprobaciones de protecciones. Una vez verificado, ya se puede ejecutar read/write múltiples veces pero ya no se vuelve a comprobar
  - Se le pasa el **nombre** del dispositivo y devuelve el dispositivo virtual (fd: file descriptor o canal) a través del cual se podrán realizar las operaciones de lectura/escritura ,etc
  - **access_mode** indica el tipo de acceso. Siempre debe ir uno de estos tres como mínimo:
    - O_RDONLY (lectura)
    - O_WRONLY (escritura)
    - O_RDWR (lectura y escritura)

Open: Creación

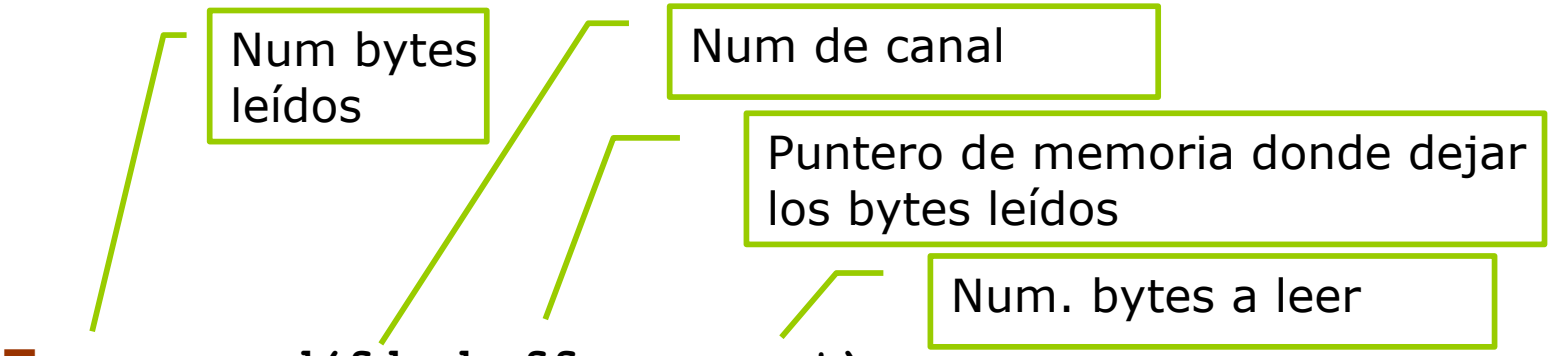
- Los ficheros especiales deben existir antes de poder acceder
- Los ficheros de datos se pueden crear a la vez que hacemos el open:
  - En este caso debemos añadir el flag O_CREAT (con una OR de bits) en el **`access_mode`**
  - Se deben especificar los **`permision_flags`**
    - Es una OR (|) de: S_IRWXU, S_IRUSR, S_IWUSR, etc
- No hay una llamada a sistema para eliminar datos de un fichero parcialmente, sólo podemos borrarlos todos. Si queremos truncar el contenido de un archivo añadiremos el flag O_TRUNC en el **`acces_mode`**
  - Ej1: open("X",O_RDWRIO_CREAT, S_IRUSRIS_IWUSR) → Si el fichero no existía lo crea, si existía no tiene efecto
  - Ej2: open("X",O_RDWRIO_CREATIO_TRUNC, S_IRWXU) → Si el fichero no existía lo crea, si existía se liberan sus datos y se pone el tamaño a 0 bytes.

Open: Estructuras de datos

- Open (cont): Efecto sobre las estructuras de datos internas del sistema
  - Ocupa un canal libre de la TC**. <u>Siempre será el primero disponible</u>**
  - **<u>Ocupa una nueva entrada de la TFA: Posición L/E=0</u>**
  - Asocia estas estructuras al DD correspondiente (major del nombre simbólico). Puede pasar que diferentes entradas de la TFA apunten al mismo DD

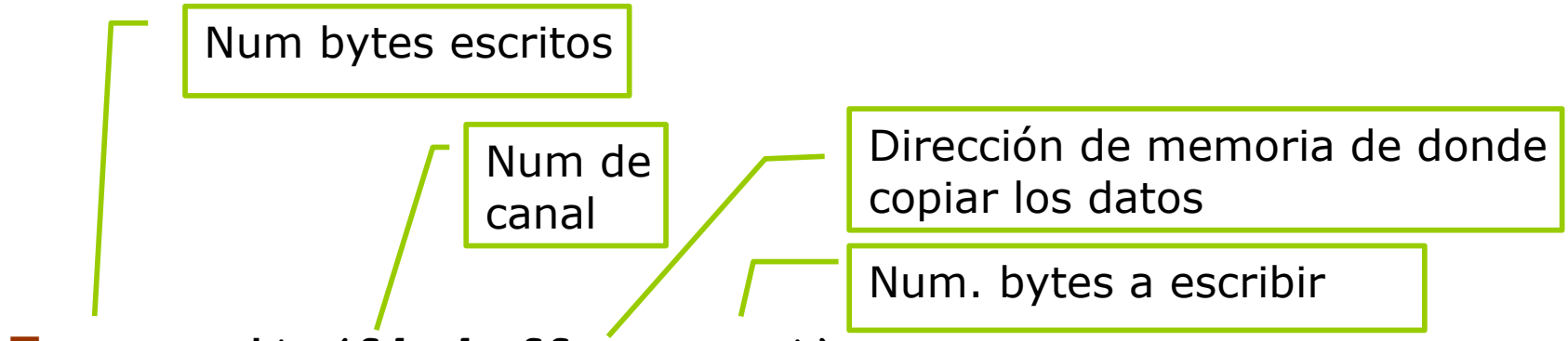


Read



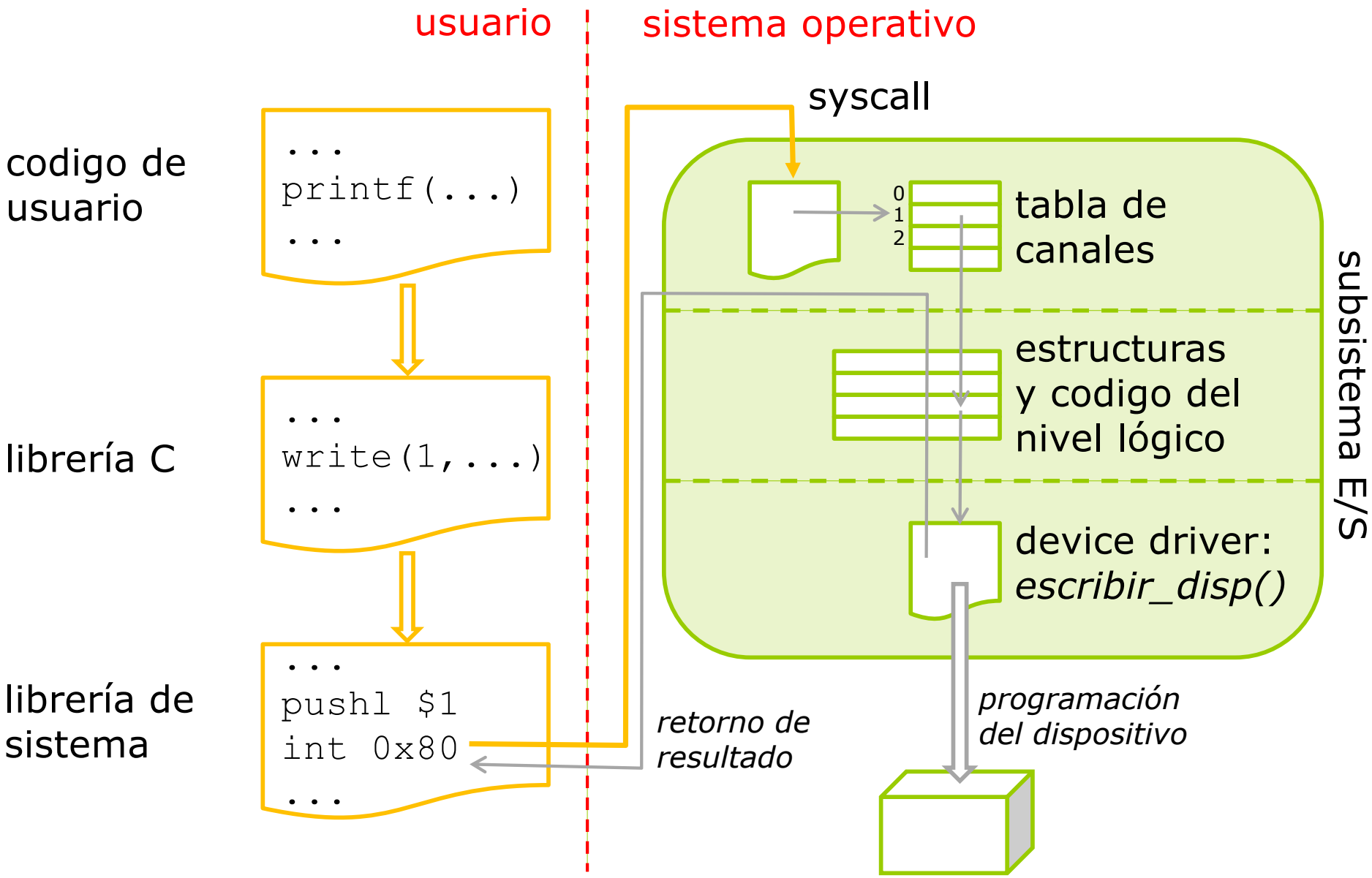
- `n = read(fd, buffer, count);`
  - Pide la lectura de **`count`** caracteres del dispositivo asociado al canal **`fd`**
    - Si hay **`count`**, o más, disponibles, leerá **`count`**
    - Si hay menos, leerá los que haya
    - Si no hay ninguno, dependerá del funcionamiento del dispositivo
      - Puede ser que se bloqueee, esperando a que haya caracteres
      - Puede ser que retorne sin ningún carácter leído
    - Cuando *EOF*, la operación devuelve sin ningún carácter leído
      - El significado de *EOF* depende del dispositivo
  - Devuelve el número de caracteres leídos (**`n`**)
  - El puntero de l/e de la TFA se avanza automáticamente **(lo hace el kernel en la rutina del read)** tantas posiciones como bytes leídos
    - puntero_l/e = puntero_l/e + n

Write

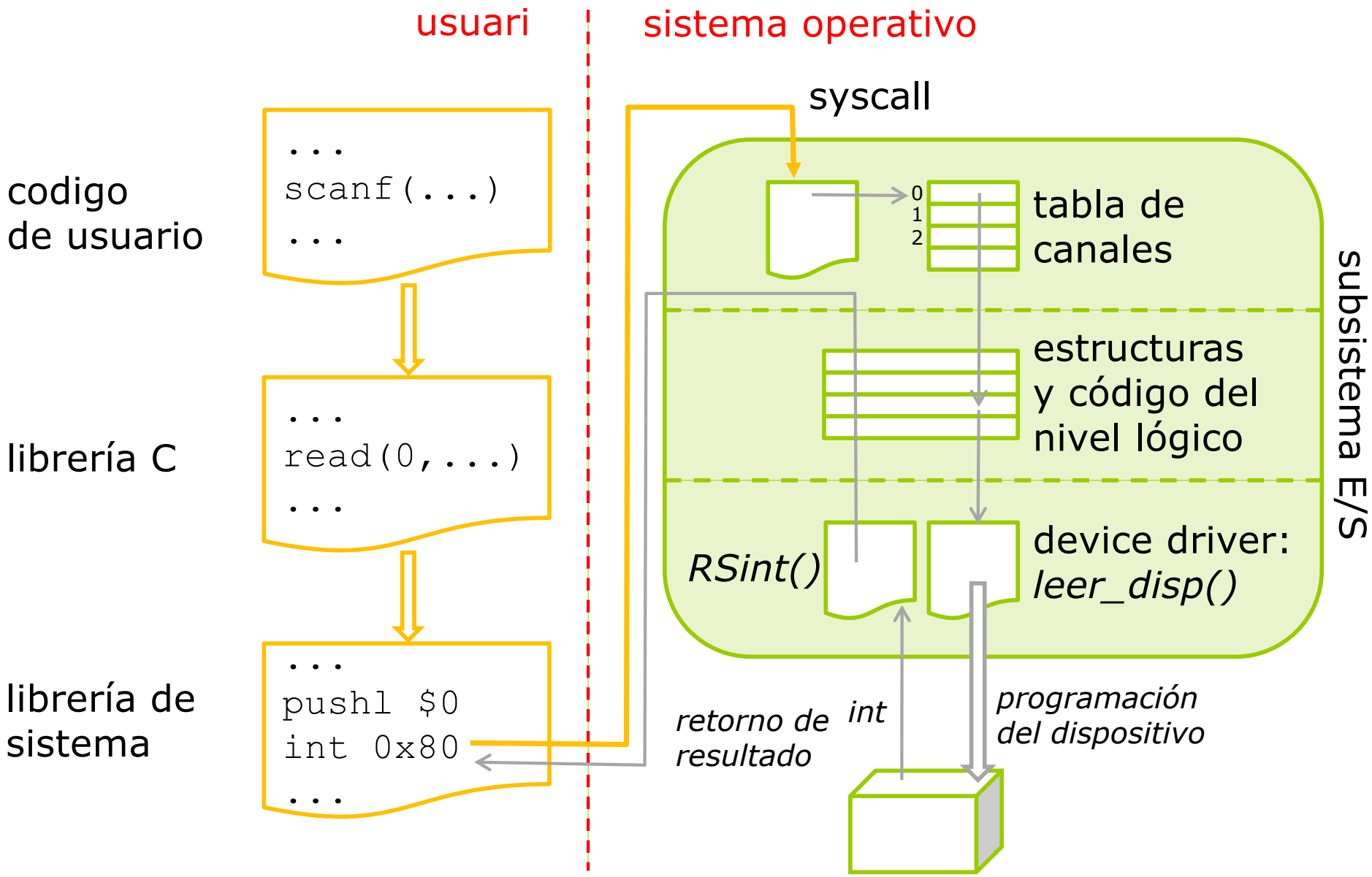


- `n = write(fd, buffer, count);`
  - Pide la escritura de **`count`** caracteres en el dispositivo asociado al canal **`fd`**
    - Si hay espacio para **`count`**, o más, escribirá **`count`**
    - Si hay menos, escribirá los que quepan
    - Si no hay espacio, dependerá del funcionamiento del dispositivo
      - Puede ser que se bloquee, esperando a que haya espacio
      - Puede ser que retorne sin ningún carácter escrito
  - Devuelve el número de caracteres escritos (**`n`**)
  - El puntero de E/S se avanza **automáticamente (lo hace el kernel en la rutina del write)** tantas posiciones como bytes escritos
    - puntero_l/e = puntero_l/e + n

Ejemplo: Escritura en un dispositivo



Ejemplo: Lectura de un dispositiu



Dup/dup2/close

- `newfd = dup(fd);`
  - Duplica un canal
  - Ocupa el primer canal libre, que ahora contendrá una copia de la información del canal original `fd`
  - Retorna `newfd`, l'identificador del nou canal

- `newfd = dup2(fd, newfd);`
  - Igual que *dup*, pero el canal duplicado es forzosamente `newfd`
  - Si `newfd` era un canal válido (ya en uso), previamente se cierra

- `close(fd);`
  - Libera el canal `fd` y las estructuras asociadas de los niveles inferiores
  - Hay que tener en cuenta que diferentes canales pueden apuntar a la misma entrada de la TFA (p.ej. si dup), por tanto, cerrar un canal significaría decrementar en 1 el contador de referencias de esa entrada
  - Lo mismo ocurre con los apuntadores en la tabla de inodos

pipe

- `pipe(fd_vector); // Dispositivo de comunicaciones FIFO`
  - Crea una *pipe* **sin** nombre. Devuelve 2 canales, uno de lectura y otro de escritura
  - No utiliza ningun nombre del sistema de ficheros (*pipe* **sin nombre**) y, por tanto, no ejecuta la llamada a sistema *open*
  - Solo podrà ser usada para comunicar el proceso que la crea y cualquier descendiente suyo (directo o indirecto) (pq herederán los canales)
- Para crear una *pipe* **con** nombre, hay que ejecutar: *mknod + open*
- Internamente: También crea 2 entradas en la Tabla de Ficheros Abiertos (uno de lectura y uno de escritura) y una entrada temporal en la Tabla de Inodos

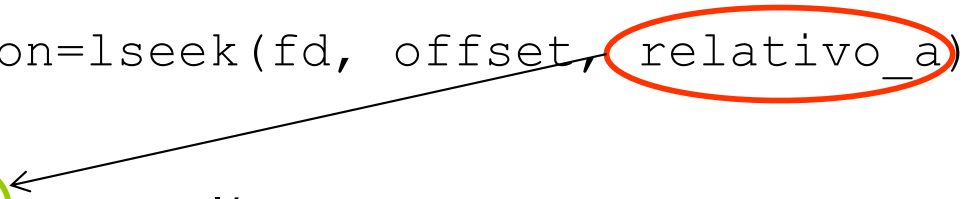
pipe

- Utilización
  - Dispositivo para comunicar procesos
  - Es bidireccional pero, idealmente cada proceso lo utiliza en un único sentido, en este caso, el kernel gestiona la sincronización
- Dispositivo bloqueante:
  - Lectura: Se bloquea el proceso hasta que haya datos en la pipe, aunque no necesariamente la cantidad de datos solicitados.
    - Cuando no hay ningún proceso que pueda escribir y pipe vacía, provoca EOF (*return*==0) → *Si hay procesos bloqueados, se desbloquean*
  - Escritura: El proceso escribe, a menos que la pipe esté llena. En este caso se bloquea hasta que se vacíe.
    - Cuando no existe ningún proceso que pueda leer el proceso recibe un signal del tipo *SIGPIPE* → *Si hay procesos bloqueados, se desbloquean*
  - ¡Es necesario cerrar los canales que no se vayan a utilizar! De lo contrario bloqueo
- Estructuras de datos
  - Se crean dos entradas en la Tabla de canales (R/W)
  - Se crean dos entradas en la TFA (R/W)
  - Se crea una entrada temporal en la Tabla de Inodos

lseek

- El usuario modifica manualment el puntero de l/e. Permite hacer accesos directos a posiciones concretas de archivos de datos (o archivos de dispositivos que ofrezcan acceso secuencial)
  - Se inicializa a 0 en el open
  - Se incrementa automáticamente al hacer read/write
  - Podemos moverlo manualmente con la llamada lseek

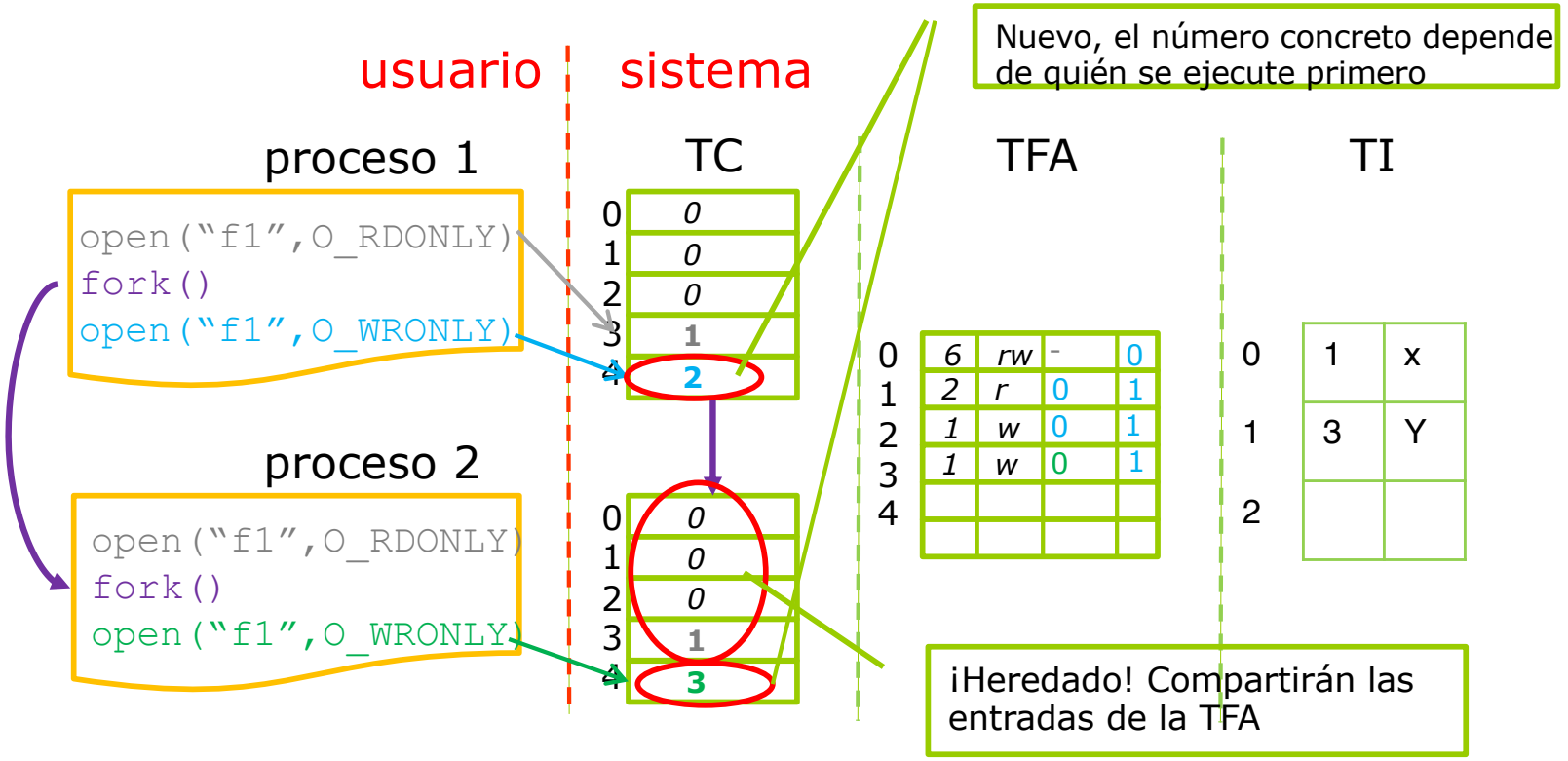
- `nueva_posicion=lseek(fd, offset, relativo_a)`



  - SEEK_SET: puntero_l/e = `offset`
  - SEEK_CUR: puntero_l/e = puntero_l/e + `offset`
  - SEEK_END: puntero_l/e = file_size + `offset`
  - "`offset`" puede ser negativo (excepto para el caso SEEK_SET)

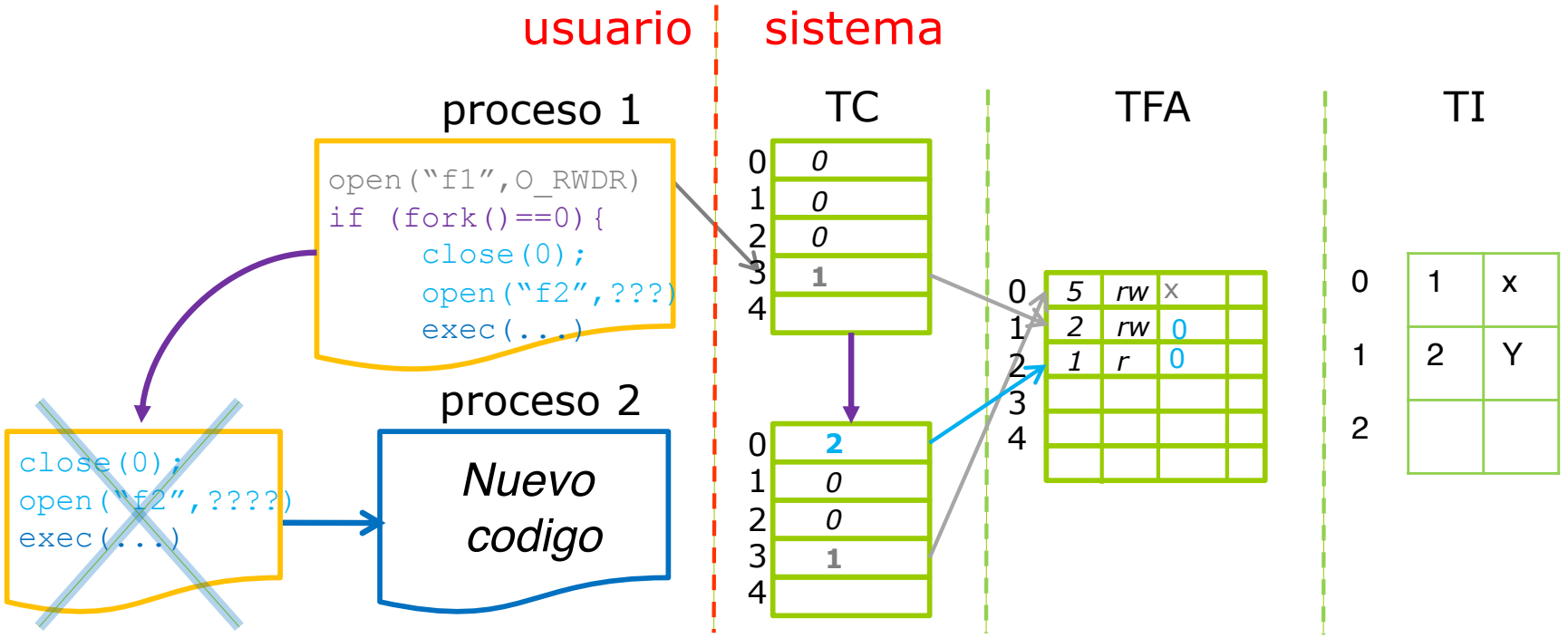
E/S y ejecución concurrente (1)

- E/S y *fork*
  - El proceso hijo hereda una **copia** de la Tabla de Canales del padre
    - Todas las entradas abiertas apuntan al mismo sitio de la TFA
  - Permite **compartir** el acceso a los dispositivos abiertos antes del *fork*
  - Los siguientes *open* ya serán independientes



E/S y ejecución concurrente (2)

- E/S y *exec*
  - La nueva imagen **mantiene** las estructuras internas de E/S del proceso
  - El hecho de ejecutar *fork+exec* permite realizar redirecciones antes de cambiar la imagen



E/S y ejecución concurrente (3)

- Si un proceso está bloqueado en una operación de E/S y la operación es interrumpida por un signal podemos tener dos comportamientos diferentes:
  - Después de tratar el signal, la operación interrumpida se reinicia (el proceso continuará bloqueado en la operación)
  - Tras tratar el signal, la operación devuelve error y la variable `errno` toma por valor `EINTR`
- El comportamiento depende de:
  - La gestión asociada al signal:
    - si flag `SA_RESTART` en el sigaction → se reinicia la operación
    - en caso contrario → la operación devuelve error
  - La operación que se está realizando (por ejemplo, nunca se reinician operaciones sobre sockets, operaciones de espera de signals, etc.)
- Ejemplo de cómo proteger la llamada al sistema según el modelo de comportamiento:

```
while ( (n = read(...)) == -1 && errno == EINTR );
```

Gestión de las características de los dispositivos

- Aunque las llamadas a sistema son uniformes, los dispositivos no lo son
- Existen llamadas a sistema para modificar características específicas de los dispositivos lógicos y virtuales.
- Dispositivo lógico
  - `ioctl(fd, cmd [, ptr]);`
- Dispositivo virtual
  - `fcntl(fd, cmd, [, args]);`
- Los parámetros son muy genéricos para ofrecer flexibilidad

Características terminal

- Terminal
  - El sistema mantiene, por cada terminal, un buffer interno para guardar los caracteres tecleados. Esto permite borrar caracteres antes de ser tratados
  - Posix define una funcionalidad particular para ciertos caracteres especiales:
    - ^H: borrar un carácter
    - ^U: borrar una línea
    - ^D: EOF o final de fichero (final de entrada de datos)
  - Puede tener también un buffer de escritura por cada terminal e implementar ciertas funciones especiales, del tipo “subir N líneas”, “avanzar izquierda”
  - Cada controlador puede implementar el terminal tan complicado como desee. P.ej. que se pueda modificar texto del medio de la línea
  - Canonical/No canonical: Pre-procesado o no de los caracteres antes de enviarlos al proceso (afecta a la lectura)

Características terminal

- Terminal: Funcionamiento (canonical ya que es el caso por defecto)
  - Lectura
    - **El buffer guarda caracteres hasta que se pulsa CR**
    - Si hay proceso bloqueado esperando caracteres, le da los que pueda
    - De lo contrario lo guarda; cuando un proceso lo pide le da
    - ^D provoca que la lectura actual acabe con los caracteres que haya en ese momento, aunque no haya ninguno:

```
while ( (n=read(0, &car, 1)) > 0 )
        write(1, &car, 1);
```

    - Eso, por convención, se interpreta como final de fichero (EOF)
  - Escritura
    - Escribe un bloque de caracteres
      - Puede esperar a que se escriba el CR para ser visualizado por pantalla
    - El proceso no se bloquea
  - Este comportamiento bloqueante puede ser modificado mediante llamadas a sistema que cambian el comportamiento de los dispositivos

Características pipes

- **Pipe**
  - Lectura
    - Si hay datos, coge los que necesita (o los que haya)
    - Si no hay datos, se queda bloqueado hasta que haya algo
    - Si la *pipe* está vacía y no existe ningún posible escritor (todos los canales de escritura de la pipe cerrados), el proceso lector recibe EOF
      - Hay que cerrar siempre los canales que no se utilicen
  - Escritura
    - Si hay espacio en la pipe, escribe los datos (o los que pueda)
    - Si la *pipe* está llena, el proceso se bloquea
    - Si no hay posible lector, recibe *SIGPIPE*
  - Este comportamiento bloqueante puede ser modificado mediante llamadas al sistema que cambian el comportamiento de los dispositivos (fcntl)

Dispositivos de red

- **Dispositivos de red**
  - Aunque es un dispositivo de E/S, la red tiene un comportamiento que no puede ser cubierto con las operaciones genéricas de acceso a los dispositivos
  - La gestión de los dispositivos de red se realiza de forma separada
    - P.ej: `/dev/eth0` no dispone ni de *inodo* ni de *device driver*
  - Disponen de llamadas al sistema específicas, y hay múltiples propuestas
  - Implementa los protocolos de comunicación de red (asignatura XC)

Dispositivos de red

- **Socket**: Funcionamiento
  - Funciona similar a una *pipe*, excepto que en lugar de dos canales (fd[2]) utiliza un único canal para la lectura y la escritura del *socket*
  - Se crea un *socket*, que se conecta al *socket* de otro proceso en otra máquina, conectada a la red
    - Se puede solicitar conexión a un *socket* remoto
    - Se puede detectar si otra quiere conectarse al *socket* local
    - Una vez conectados, se pueden enviar y recibir mensajes
      - *read* /*write* o, mas específicamente, *send* /*recv*
  - Se dispone de llamadas adicionales que permiten implementar servidores concurrentes o, en general, aplicaciones distribuidas
  - Habitualmente, se ejecutan aplicaciones con un modelo cliente-servidor

Dispositivos de red

- **Socket**: Ejemplo (*pseudo-codigo*)

Cliente	Servidor
	`...`
	`sfd` `= socket(...)`
`...`	`bind(sfd, ...)`
`sfd` `= socket(...)`	`listen(sfd, ...)`
`connect(sfd, ...)`	**`nfd`** `= accept(sfd, ...)`
`write/read(`**`sfd`**`, ...)`	`read/write(`**`nfd`**`, ...)`

EJEMPLOS

Acceso byte a byte

- Lectura por la entrada estándar y escritura por la salida estándar

```
while ((n = read(0, &c, 1)) > 0)
    write(1, &c, 1);
```

  - Observaciones:
    - Por lo general, los canales ya están abiertos por defecto
    - Se lee hasta que no quedan datos (`n==0`), que depende del tipo de dispositivo
    - El total de llamadas a sistema depende de cuántos bytes hay para leer
    - Los canales 0 y 1 pueden estar vinculados con cualquier dispositivo lógico que permita leer y escribir

```
#exemple1 →                    entrada=terminal, salida=terminal
#exemple1 <disp1 →             entrada=disp1, salida=terminal
#exemple1 <disp1  >disp2 →     entrada=disp1, salida=disp2
```

Acceso con buffer de usuario

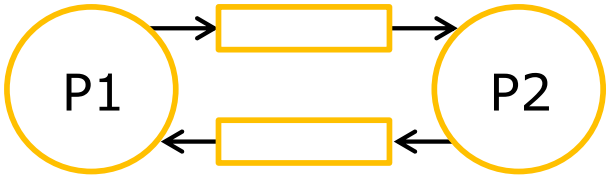
- Igual, pero leemos bloques de bytes (chars en este caso)

```
char buf[SIZE];
...
while ((n = read(0, buf, SIZE)) > 0)
    write(1, buf, n);
```

  - Observaciones:
    - ¡Ojo! Hay que escribir `n` bytes
      - Pedimos SIZE bytes, ya que la variable lo permite, pero es un máximo, puede que hayamos leído menos
    - ¿Qué diferencias existen en cuanto a rendimiento? ¿Cuántas llamadas a sistema haremos?

Intercambio de información con pipes

- Crear un esquema de procesos que sea equivalente a este



- 2 pipes
- P1 envia por pipe1 i recibe de pipe2
- P2 al revés

```
void p1(int fdin,int fdout);
void p2(int fdin,int fdout);
```

```
1.  int pipe1[2], pipe2[2],pidp1,pidp2;
2.  pipe(pipe1);
3.  pipe(pipe2);
4.  pidp1=fork();
5.  if (pidp1==0){
6.        close(pipe1[0]);
7.        close(pipe2[1]);
8.        p1(pipe2[0],pipe1[1]);
9.        exit(0);
10. }
11. close(pipe1[1]);
12. close(pipe2[0]);
13. pidp2=fork();
14. if (pidp2==0){
15.       p2(pipe1[0],pipe2[1]);
16.       exit(0);
17. }
18. close(pipe1[0]);close(pipe2[1]);
19. while(waitpid(-1,null,0)>0);
```

Acceso aleatorio y cálculo del tamaño

- ¿Qué hace el siguiente fragmento de código?

```
fd = open("abc.txt", O_RDONLY);
while (read(fd, &c, 1) > 0) {
    write(1, &c, 1);
    lseek(fd, 4, SEEK_CUR);
}
```

Encontraréis este código en: ejemplo1.c

- ¿Y este otro?

```
fd = open("abc.txt", O_RDONLY);
size = lseek(fd, 0, SEEK_END);
printf("%d\n", size);
```

Encontraréis este código en : ejemplo2.c

pipes i bloqueos

```
int fd[2];
...
pipe(fd);
pid = fork();
if (pid == 0) {                          // Codigo del hijo
    while (read(0, &c, 1) > 0) {
        // Lee el dato, lo procesa y lo envia
        write(fd[1], &c, 1);
    }
}
else {                                   // Codigo del padre
    while (read(fd[0], &c, 1) > 0) {
        // Recibe el dato, lo procesa i lo escribe
        write(1, &c, 1);
    }
}
...
```

Tenéis disponible este código en: pipe_basic.c

- ¡Ojo! ¡El padre tiene que cerrar `fd[1]` si no se quiere quedar bloqueado!

Compartir el puntero de l/e

- ¿Qué hace este fragmento de código?

```
...
fd = open("fitxer.txt", O_RDONLY);
pid = fork();
while ((n = read(fd, &car, 1)) > 0 )
        if (car == 'A') numA++;
sprintf(str, "El número d'As és %d\n", numA);
write(1, str, strlen(str));
...
```

Tenéis disponible este código en: exemple1.c

No compatir el punter de l/e

- ¿Qué hace este fragmento de código?

```
...
pid = fork();
fd = open("fitxer.txt", O_RDONLY);
while ((n = read(fd, &car, 1)) > 0 )
       if (car == 'A') numA++;
sprintf(str, "El número d'As és %d\n", numA);
write(1, str, strlen(str));
...
```

Tenéis disponible este código en: exemple2.c

Redirección de la entrada y la salida std

- ¿Qué hace este fragmento de código?

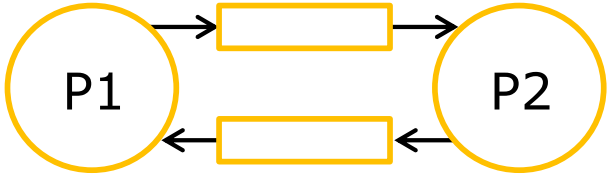
```
...
pid = fork();
if ( pid == 0 ) {
    close(0);
    fd1 = open("/dev/disp1", O_RDONLY);
    close(1);
    fd2 = open("/dev/disp2", O_WRONLY);
    execv("programa", "programa", (char *)NULL);
}
...
```

Pipes y redirección

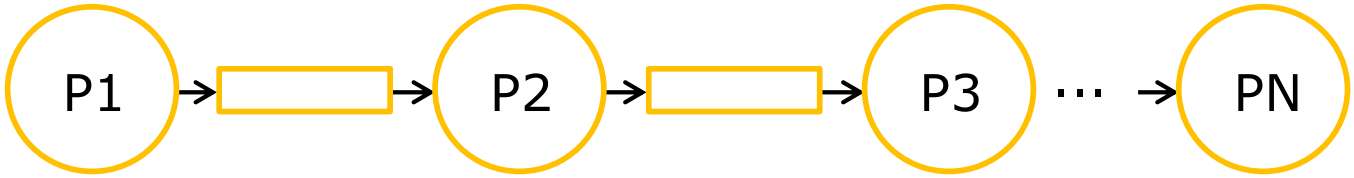
```
...
pipe(fd);
pid1 = fork();
if ( pid1 != 0 ) {                              // PADRE
    pid2 = fork();
    if ( pid2 != 0 ) {                          // PADRE
        close(fd[0]); close(fd[1]);
        while (1);
    }
    else {                                      // HIJO 2
        close(0); dup(fd[0]);
        close(fd[0]); close(fd[1]);
        execlp("programa2", "programa2", NULL);
    }
}
else {                                          // HIJO 1
        close(1); dup(fd[1]);
        close(fd[0]); close(fd[1]);
        execlp("programa1", "programa1", NULL);
}
```

Ejercicios en clase (1) (Col.P clase)

- Escribid un fragmento de código que cree dos procesos p1 y p2, y los conecte mediante dos pipes por los canales de E/S estándar: en la primera p1 escribirá y p2 leerá, en la segunda p2 escribirá y p1 leerá

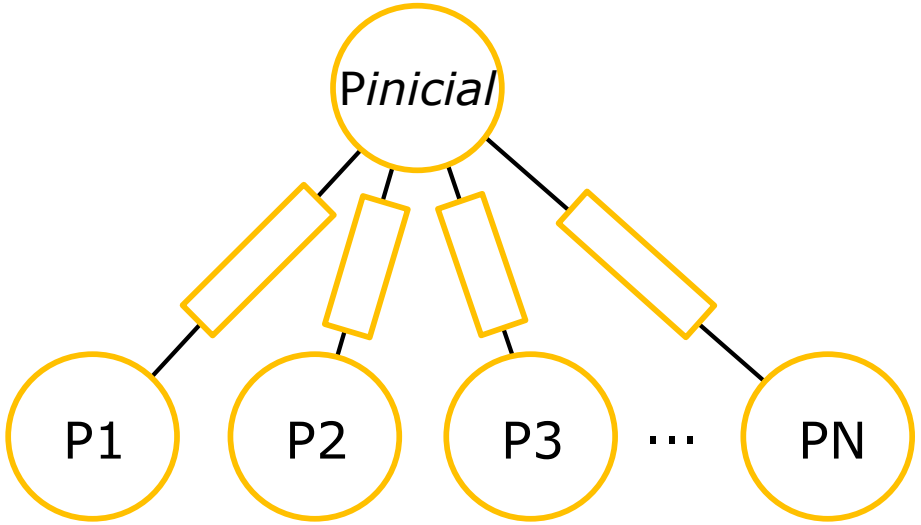


- Escribid un fragmento de código que cree una secuencia de N procesos en cadena: cada proceso p_i crea un solo descendiente p_i+1, hasta llegar a p_N. Cada proceso debe comunicarse en cadena con el ascendente y el descendente mediante una pipe por el canal de E/S estándar, de modo que lo que escriba el primer proceso pueda ser enviado en cadena hasta el último proceso



Ejercicios en clase (2)

- Escribid un fragmento de código que cree N procesos en secuencia: el proceso inicial crea todos los descendientes p_1 hasta llegar a p_N. Cada proceso debe comunicarse con el proceso padre mediante una pipe; el padre debe poder escribir en todas las pipes (a través de los canales 3..N+2) y los hijos deben leer de la pipe por su entrada d.



SISTEMA DE FICHEROS

Tareas del sistema de ficheros

- Responsabilidades del Sistema de ficheros respecto a gestionar ficheros y almacenar datos
  - Organizar los ficheros del sistema → espacio de nombres y esquema de directorios
  - Garantizar el acceso correcto a los ficheros (permisos de acceso)
  - Gestionar (asignar/liberar) espacio libre/ocupado por los ficheros de datos
  - Encontrar/almacenar los datos de los archivos de datos

- En cualquier caso, todos los archivos (de cualquier tipo) tienen un nombre que debe almacenarse y gestionarse. Los nombres de archivo se organizan en directorios

Espacio de nombres: Directorio

- Directorio: Estructura lógica que organiza los archivos.
- Es un fichero especial (tipo directorio) gestionado por el SO (los usuarios no pueden abrirlo y manipularlo)
- Permite asociar los nombres de los archivos con sus atributos

En Linux, todo esto está en el inodo

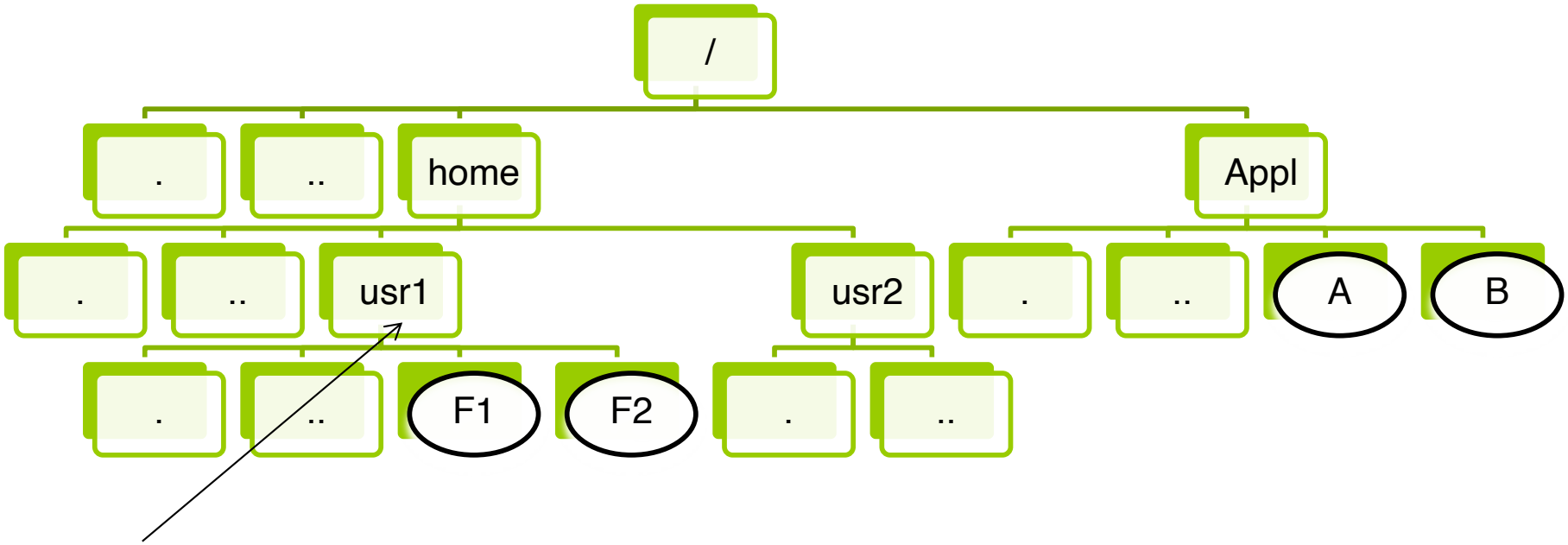
- atributos
  - Tipo de archivo: directorio(d), block(b), character(c), pipe(p), link(l),socket(s), de datos(-)
  - tamaño
  - propietario
  - permisos
  - Fechas de creación, modificación, ...
  - ...
- Ubicación en el dispositivo de sus datos (en caso de archivos de datos)

Linux: Visión de usuario

- Los directorios se organizan de forma jerárquica (forman un grafo)
- Permiten que el usuario clasifique sus datos
- El Sistema de Archivos organiza los dispositivos de almacenamiento (cada uno con su esquema de directorios), en un espacio de nombres global con un único punto de entrada
- El punto de entrada es el directorio "root" (Raíz), de nombre "/"
- Un directorio siempre tiene, como mínimo, 2 archivos especiales (de tipo directorio)
  - . Referencia al directorio actual
  - .. Referencia al directorio padre

Linux: Visión de usuario

- Cada fichero se puede referenciar de dos maneras:
  - Nombre absoluto (único): Camino desde la raíz + nombre
  - Nombre relativo: Camino desde el directorio de trabajo + nombre



Si estamos aquí, podemos referirnos a F2 como:
Nombre relativo: F2
Nombre absoluto: /home/usr1/F2

Linux: Nombres de ficheros

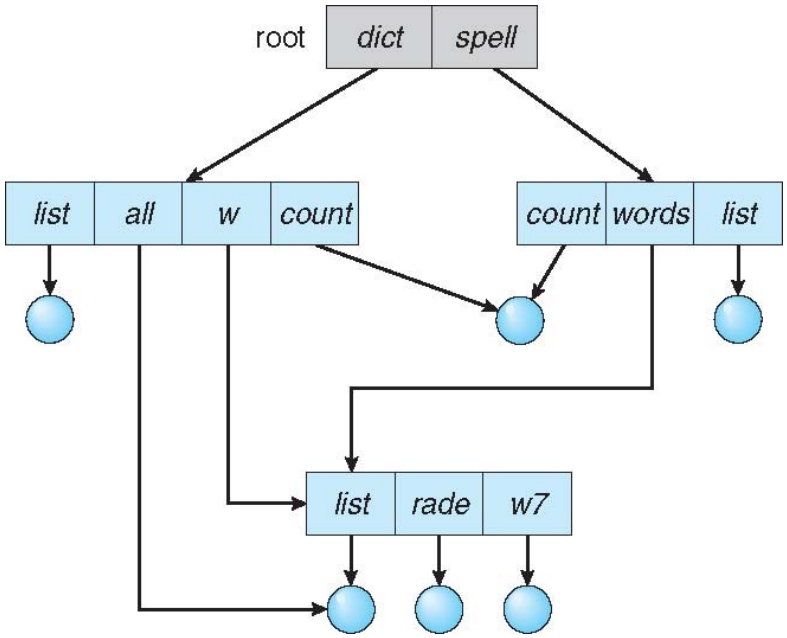
- Como el nombre del fichero está separado de la información (inodo), en Linux se permite que un inodo sea referenciado por más de un nombre de archivo
- Hay dos tipos de links entre nombre de archivo e inodo
  - Hard-link
    - El nombre del archivo referencia directamente al número de inodo donde están los atributos+info de datos
    - Es lo más habitual
    - En el inodo hay un número de links que indica cuántos nombres de archivos apuntan a este inodo
      - ¡¡¡Es diferente al número de referencias que hay en la Tabla de Inodos!!!
  - Soft-link
    - El nombre del archivo no apunta directamente a la información sino que apunta a un inodo que contiene el nombre del archivo destino (path absoluto o relativo)
    - Es un tipo de archivo especial (l)
    - Se crea con un comando (ln) o con una llamada a sistema (symlink)

Implementación jerarquía directorios

- La existencia de los dos tipos de links influye en la estructura del directorio (que es un grafo)
  - No se permiten ciclos con hard-links
  - Sí se permiten ciclos con soft-links

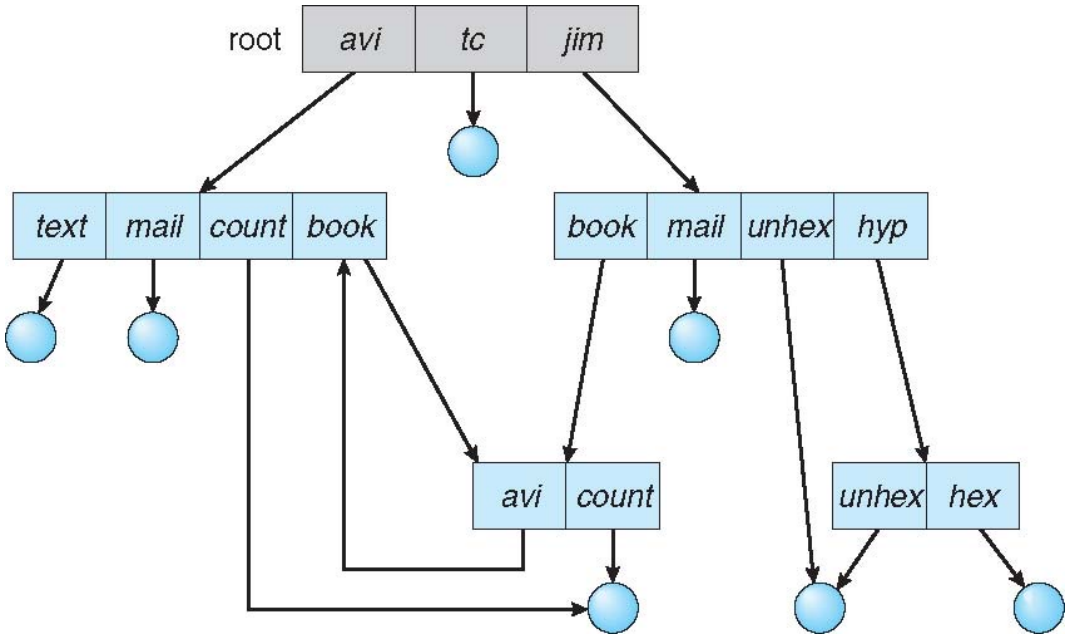
Grafo Acíclico

el SF verifica que no se creen ciclos



Grafo Cíclico

el SF tiene que controlar los ciclos infinitos

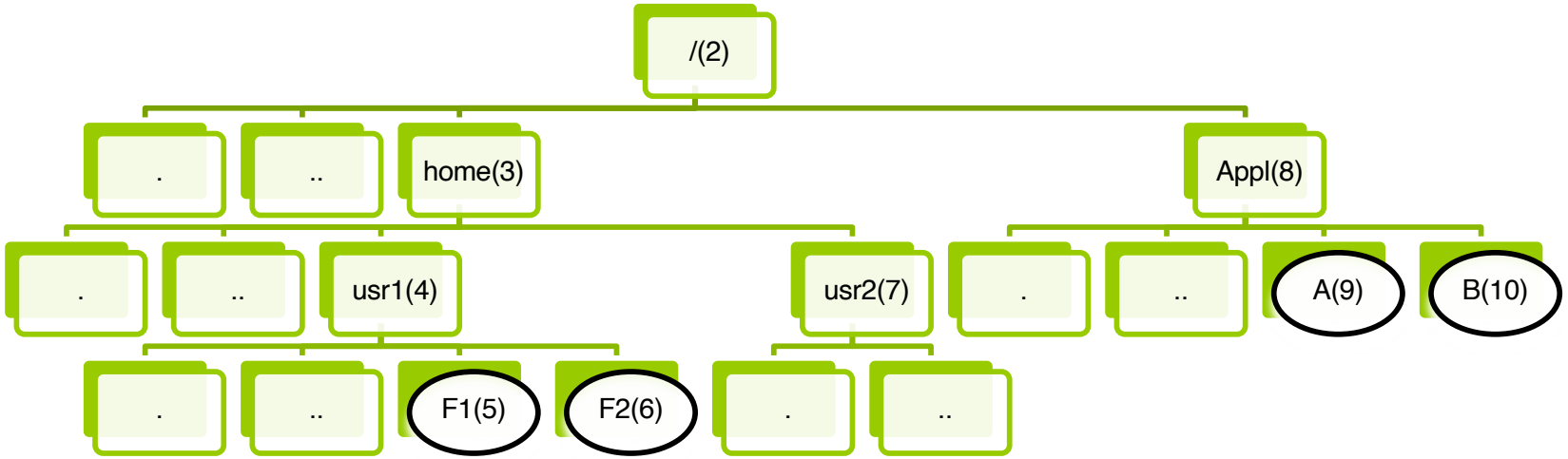


Problemas de los directorios en grafo

- Backups (copias de seguridad)
  - No hacer dos copias del mismo fichero
- Borrado de un fichero
  - Soft links
    - El sistema no comprueba si hay soft-links a un fichero. A la hora de acceder se detectará que no existe.
  - Hard links
    - El fichero solo se elimina cuando el contador de referencias llega a cero

Visión interna de directorios

- El contenido de un directorio permite relacionar el nombre del fichero con su número de inodo
- Pej: En el ejemplo anterior, si asignamos números de inodos



Nombre	inodo
.	2
..	2
home	3
Appl	8

Directorio /

Nombre	inodo
.	4
..	3
F1	5
F2	6

Directorio /home/usr1

Caso especial

Permisos de acceso a ficheros

- El SF permite asignar diferentes permisos a los ficheros
  - Se definen niveles de acceso y operaciones que se pueden realizar
- Linux:
  - Niveles de acceso: Propietario, Grupo de usuarios, Resto de usuarios
  - Operaciones: Lectura (r) , Escritura (w), Ejecución (x) → ¡No confundir con el modo de acceso!
  - Cuando se definen de forma numérica, se utiliza la base 8 (octal). Algunos valores

R	W	X	Valor numérico
1	1	1	7
1	1	0	6
1	0	0	4

- Revisar documentación laboratorio

Llamadas a sistema: espacio de nombres y permisos

Servicio	Llamada a sistema
Crear/eliminar enlaces a ficheros/soft-link	link / unlink/symlink
Cambiar permisos de un fichero	chmod
Cambiar propietario/grupo de un fichero	chown / chgrp
Obtener información del Inodo	stat, lstat, fstat

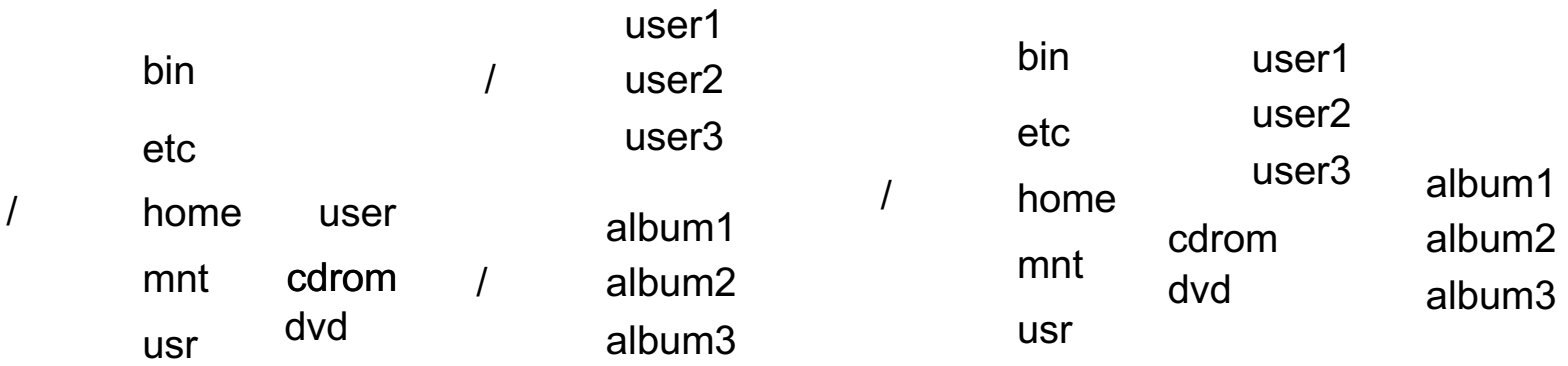
- Existen más llamadas a sistema que nos permiten manipular:
  - Permisos
  - Links
  - Características
  - etc

Unidades de trabajo del disco

- Un dispositivo de almacenamiento está dividido en partes que llamaremos sectores
- La unidad de asignación del SO es el bloque (1 bloque se corresponde con 1 o más sectores)
- Partición o volumen o sistema de archivos
  - Conjunto de sectores **consecutivos** con un identificador único (dispositivo lógico) y con identidad propia. Son gestionados por el SO como una entidad lógica independiente
    - C:, D: (Windows); /dev/hda1, /dev/hda2 (UNIX)
  - Cada partición tiene su propia estructura de directorios y archivos independiente del resto de particiones

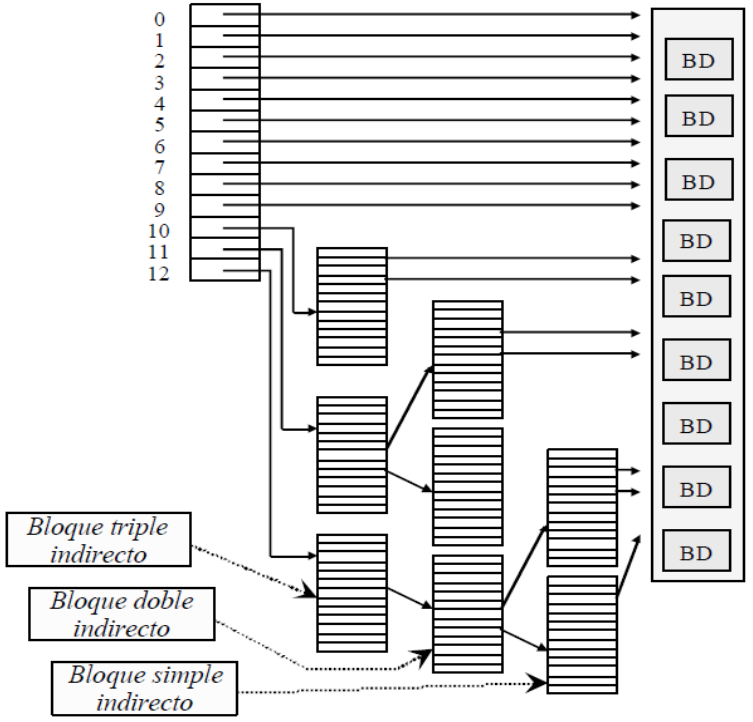
Acceso a particiones

- Montar partición: acción de hacer accesible una partición en el sistema de directorios global (solo lo puede hacer el administrador de la máquina)
  - Linux: mount (para montar), umount (para desmontar)
    - # mount –t ext2 /dev/hda1 /home
    - # umount /dev/hda1



Gestión del espacio ocupado

- Para cada fichero, el SO tiene que saber cuáles son sus bloques.
  - Tabla de índices: bloques asignados a un ficheros
  - Está en el inodo del fichero: ¿Cuántos índices?
    - Esquema multinivel: índices directos e indirectos
- Índices a bloques de datos (tamaño de bloque1Kb/4Kb)
  - 10 índices directos
    - (10 bloques = 10Kb/40Kb)
  - 1 índice indirecto
    - (256/1024 bloques = 256Kb/4Mb
  - 1 índice indirecto doble
    - (65K/1M bloques = 65Mb/4 Gb)
  - 1 índice triple indirecto
    - (16M/1G bloques = 16Gb/4 Tb)



Gestión del espacio libre

- Gestión del espacio libre:
  - Lista de bloques libres y lista de inodos libres
  - Cuando hace falta un nuevo inodo (creación de un archivo) o un nuevo bloque (para hacer crecer un archivo) se coge el primero de la lista correspondiente

Metadatos

- Metadatos persistentes: **guardados en disco**
  - Inodo y lista de bloques de un fichero
  - Directorios
  - Lista de bloques libres
  - Lista de inodos libres
  - … Y otra información necesaria para el sistema de archivos (cuál es el inodo del directorio raíz, tamaño de bloque…)
- Superbloque: bloques de la partición que contiene los metadatos del sistema de archivos (redundancia para que sea tolerante a fallos)
- Se utiliza memoria para guardar estos metadatos mientras están en uso (y así ahorrar accesos a disco)

Metadatos y memoria

- Zona de memoria para guardar los últimos inodos leídos
- Zona de memoria para guardar el últimos directorios leídos
- Buffer cache: zona de memoria para guardar los últimos bloques leídos
- Superbloque: se guarda también en memoria

RELACIÓN ENTRE LLAMADAS A SISTEMA / ESTRUCTURAS DE DATOS

Open

- Busca el inodo del archivo con el que se quiere trabajar y lo deja en memoria
  - ¿Cuál es el inodo? Es necesario leer el directorio del archivo. Dos situaciones:
    - O el directorio está en la caché de directorios y simplemente se accede
    - O el directorio está a disco y debe leerse (y se deja en memoria).
      - ¿Cómo sabemos qué bloques leer? → están en el inodo
      - ¿Cuál es el inodo? → está en el directorio (repetimos el proceso)
- Si localiza el inodo
  - Comprueba si el acceso es correcto (los permisos de acceso son los adecuados) → si no devuelve errro
  - Si es un soft-link, lee el path del archivo al que apunta y localiza el inodo correspondemos (repetimos el proceso)
    - Si el path es corto, se encuentra en el mismo inodo
    - Si el path es largo, se guarda en un bloque de datos separado
  - Modifica la tabla de canales, la tabla de archivos abiertos y la tabla de inodos
- Si no localiza el inodo (y no se usa el flag de creación):
  - Error por acceder a un archivo que no existe
- No accede a ningún bloque de datos del archivo objetivo

Open (cont)

- Si utilizamos el open para crear un nuevo archivo
  - Se debe reservar e inicializar un nuevo inodo
    - Actualizar lista de inodos libres en el superbloque
    - Actualizar directorio donde se crea el archivo
      - modificar el bloque de datos del directorio
      - modificar el inodo con el nuevo tamaño del directorio

Read

- Tenemos en la tabla de inodos el inodo correspondiente (y la información sobre los bloques de datos)
- Tenemos en la TFA el valor del puntero de lectura/escritura (la posición del archivo que toca acceder)
- Hay que calcular los bloques involucrados:
  - Primero comprobamos si estamos al final del archivo (valor del puntero == tamaño del archivo)
  - Después calculamos los bloques
    - Dividiendo el valor del puntero entre el tamaño de bloque obtenemos el índice del primer bloque a leer
    - Con el parámetro tamaño de la syscall podemos calcular cuántos bloques más es necesario leer
- Si los bloques ya se habían utilizado antes los podemos encontrar en memoria, si no es necesario leerlos de disco (y se dejan en memoria)

Write

- La diferencia con el read es que si la escritura se hace al final del archivo puede ser necesario añadir más bloques al archivo. En este caso, será necesario modificar la lista de bloques libres del superbloque y actualizar el inodo del archivo con los nuevos bloques y el nuevo tamaño del archivo

Close

- Provoca la actualización del inodo con los cambios que se hayan hecho (como mínimo la fecha de último acceso)