Tema 5 Gestión IO y FileSystems

GESTIÓN IO

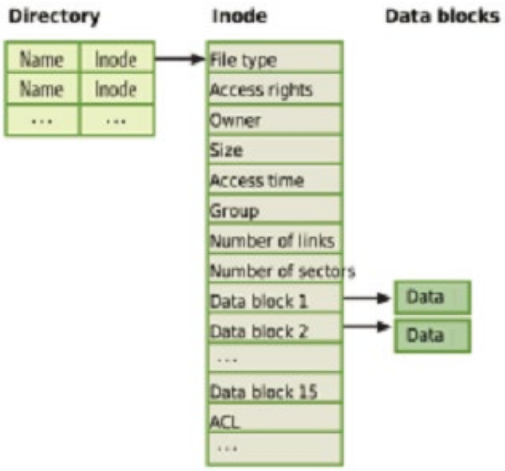
En los 70, los diquetes de 160kb solo permitian 16 ficheros con nombres de 8bits. La metadata era simplemente #pista y #sector.

Hoy en dia podemos tener dos tipos de ficheros: ficheros como tal, y directorios que contienen a los primeros, o a otros directorios. Así nace el árbol de ficheros y es jerárquico, pero es muy profundo.

Para cada directorio tenemos una lista de ficheros que para cada uno contiene: nombre, tipo de fichero (f/d), metadata. Los ficheros tienen datos.

Para reducir las rutas absolutas a un fichero determinado, ahora tenemos un grafo de ficheros. Eso significa que a un fichero se puede acceder desde más de una ruta. Si hago un borrar de este fichero, el sistema de ficheros queda inconsistente, porque en las demás rutas ya no se apunta a ese fichero.

Para resolver este problema tenemos los i-nodos, que contienen el número de referencias, y los metadatos. Su función no es contener los datos de los ficheros, sino este número de referencias, pero se aprovecha para meter los metadatos.




```
  inodo
|    M-m  |   #major y minor
|   #ref     | #num referencias
|              |
|metadata|
```

Ahora, en un directorio tenemos:

```
|nom | inodos|
|        |          |
|        |          |
|  hard links  |  # aquí tenemos guardados los hard links
```

Un hard link es un enlace directo a un fichero o directorio del sistema de ficheros, si se modifica algún atributo del fichero, estos cambios se reflejan en todos los hard links. A diferencia del soft link, que solamente es un acceso directo, apunta a una posición y si el fichero deja de estar ahí, el soft link no cambia.

Un **sector** (512B / 1-2KB) de un disco duro es la unidad mínima de transferencia del hardware, el controlador solo me permite leer y escribir a nivel de sector. Un **bloque** es la unidad mínima de transferencia desde el punto de vista del SO. Para asociar bloques con sectores tenemos 3 formas:

```
|    ID bloque    |   sector   |
```

Todos los bloques tienen el mismo número de sectores, pero no los comparten. Para saber el bloque de un sector hay que hacer: ID * #(sectores por bloque).
El problema de esta asignación es la fragmentación. Se asigna un bloque como mínimo, por lo que al asignar espacio a un fichero, que se hace a nivel de bloque, hay fragmentación cuando el tamaño del fichero no es múltiplo del tamaño de bloque.

Para solucionar esto, hacemos que se pueda tener bloques de tamaños diferentes (siguen sin compartir sectores). En esta segunda implementación necesitamos saber, dado un número de bloque, qué sectores pertenecen a él. Aún hay fragmentación, aunque ahora es más pequeña.

La tercera implementación permite compartir sectores, de manera que se puede eliminar el problema de la fragmentación, se puede definir el tamaño del bloque a la hora de asignarlo, por esto mismo un sector puede pertenecer a varios bloques.

Actualmente se utiliza la primera implementación, pues dado un ID es muy fácil saber el sector, y la fragmentación aparentemente tan notable realmente no lo es porque solo ocurre con el último bloque. La tercera nunca se ha usado.

A partir de ahora al hablar de bloques, todos son iguales y no comparten sectores.

ASIGNACIÓN DE MEMORIA
Para la asignación de memoria para cada proceso, el SO usa una tabla de índices almacenada en la memoria RAM que se inicializa con el sistema. Cada entrada representa un bloque de memoria y se utiliza para acceder a este.

- Asignación Contigua :

Los bloques se asignan de manera contigua. Se suele usar para backups, que se guardan los datos de manera contigua y si queremos hacer otro se tira todo y se hace otro. También para juegos, bluray...
El acceso secuencial es bueno, el aleatorio también.
El problema: fragmentación externa (huecos entre bloques), asignación estática (necesita saber tamaño del fichero), reubicación.

- Asignación Encadenada:

Los bloques no se asignan de manera contigua, sino que cada bloque tiene un puntero (último 4 bytes del bloque) que apunta al siguiente en la lista enlazada.
El acceso secuencial es bueno, el aleatorio no. Asignación dinámica que soluciona reubicación.
El problema: fragmentación externa, más memoria para los punteros.

- Asignación Encadenada en Tabla – FAT:

Cada entrada en la tabla contiene información sobre el estado del bloque y su tamaño. La tabla enlaza los bloques libres y encuentra los bloques contiguos adecuados para asegurar la eficiencia.
La FAT que hay en el disco crea una copia y la pega en el disco. Siempre accede en la copia de memoria para leer o escribir, por lo tanto se recorre en memoria, mejorando el acceso aleatorio.
Lo bueno es que es más rápido asignar, bloques de tamaño variable, menor fragmentación.
Lo malo: mayor tiempo de acceso, la tabla ocupa más.

- Asignación Indexada – NTFS:

Se guardan bloques de índices que contienen datos de usuarios. Solo trae a memoria los índices con datos, no todos, es como el FAT pero reparte estos encadenamientos por todo el disco.
Las tablas que se ven en la foto son bloques del disco. Se trae al disco los bloques que necesita. Para acceder a un bloque de memoria, se utiliza el índice para averiguar su dirección base.
Acceso secuencial y random es bueno.
Lo bueno: tamaño de bloque variable, acceso rápido y asignación eficiente.
Lo malo: si hay muchos bloques pequeños hay fragmentación.

- Asignación Indexada Multinivel:

Crea una jerarquía de bloques de índices. Cada nivel contiene índices que apuntan a a las tablas del nivel siguiente. Se utiliza en sistemas con grandes espacios de direcciones.
Lo malo: mayor consumo de memoria, complejidad de implementación.
Lo bueno: acceso eficiente y baja fragmentación.
Disco Duro:

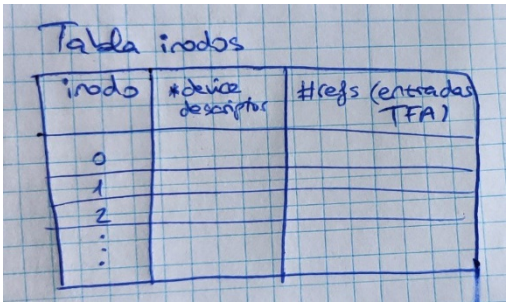
MBR|4 particiones|Superblock|InodesRecords|BBDD

En cada partición tenemos 1024 ficheros diferentes. Actualmente las particiones pueden crecer, normalmente se dobla la cantidad, no por cantidades aleatorias.

INODOS

Un **inodo** es una estructura de datos que contiene la información de ficheros y archivos, por ejemplo los punteros a sus bloques en disco. Se crean al generar el sistema de ficheros, por lo que ni se redimensionan ni se destruyen, solo ser asociados.

La **tabla de inodos** es una caché de inodos que indica cuales estoy usando, los que no se almacenan en el disco, ya que su utilidad es rebajar los accesos a disco. Es una única para todo el sistema. Si quiere usar un inodo lo trae a la caché, si ya está, lo usa.

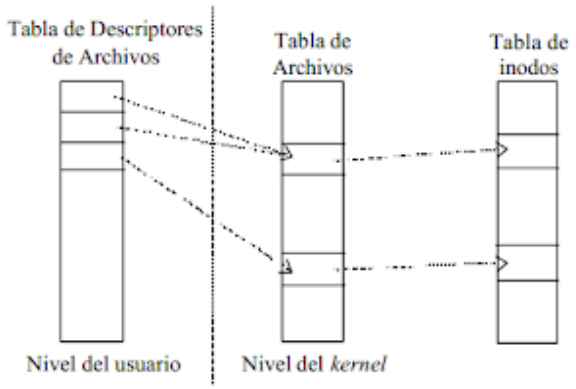


Dentro de cada entrada, entre otros campos, tiene el inodo, el puntero al device descriptor y el número de referencias que indica cuantas entradas de la TFA hacen referencia a ese inodo; también será el critero para vaciar o no de la tabla de inodos una entrada a disco.

La **Tabla de Ficheros Abiertos** lleva un control sobre cuántos links llevan a un mismo inodo, para llevar la cuenta y no dejar inconsistencias en las rutas del sistema de ficheros. Almacena el puntero de R/W (modo de acceso), el puntero al inodo, y #referencias. El puntero de R/W determina el offset del fichero a partir de donde hará R/W, el puntero de inodo apunta a este inodo en la Tabla de inodos, y las #referencias de la TFA indica cuantos canales apuntan a su entrada de TFA.

La **Tabla de Canales** es una tabla que almacena los file descriptors de los dispositivos virtuales, y es apuntada desde el PCB de cada proceso con un puntero, y cada entrada a su vez, hace referencia a la TFA.

<u>Relaciones</u>:
La conexión TFA – Tinodos permite un acceso concurrente al dispositivo lógico, porque podemos estar escribiendo con un proceso en el byte 0, y con otro en el byte 1290, gracias al puntero R/W.
La conexión TFA – TC nos permite el acceso compartido al dispositivo, ya que tenemos varios canales apuntando a la TFA, y compartiendo el puntero R/W, por lo que lo que se haga en un canal se verá reflejado en otro.



DISPOSITIVOS

El hardware está compuesto de dispositivos (tarjeta de red, de gráficos...) y el SO llega a ellos a través de procesos. Para ello, se utilizan llamadas al sistema, el cual el programador no conoce, solamente el SO. Estos dispositivos son tratados como ficheros, y es lo que percibe el usuario.

Tipos:

- **Dispositivo hardware:** tarjeta de red, tarjeta de gráficos, ratón, teclado...

- **Dispositivo lógico:** ficheros y directorios. Los de hardware se visualizan a través de estos, un fichero para el teclado, otro para ratón, tarjeta de red...
  Su función puede ser:
    - **Abstración de hardware**: representa dispositivo hw. Ej: */dev/kbd01 = teclado*
    - **Agregar hardware:** agrupar dispositivos hw, por ejemplo, la terminal (*/dev/tty*) agrupa la pantalla, el teclado, el ratón y la tarjeta de red.
    - **Añadir características del SO:** por ejemplo */dev/null* no está asignado a ningún disp. físico, por lo que no guardará nada en memoria.

- **Dispositivo virtual (canales o file descriptors):** son instancias de dispositivos lógicos. Cuando abrimos un fichero, trabajamos en este dispositivo lógico a través de uno virtual asociado, si trabajamos con la tarjeta de vídeo, trabajamos con el virtual asociado al lógico asociado al físico.

Durante el proceso init, a través de la BIOS, el SO escanea información sobre el hardware conectado y dónde está conectado. Escanea en su lista de drivers para encontrar si existe algún código capaz de controlar ese dispositivo (si no, ha de instalarse con el CD), para eso necesita el PID (Product Identifier único, compuesto por header y código) con el que identifica el dispositivo.

Una vez adquirido el controlador, alocata memoria para la estructura del **device descriptor**, donde tiene un header que almacena el PID, nombre del driver, y el **major**, un número único que identifica al driver y a la familia de productos que pueden trabajar con él. En el **minor**, guarda dónde está conectado el dispositivo. A través del major y el minor sabemos que dispositivo es y dónde está conectado, por ejemplo *teclado en USB0.*

Una vez alocatado y completado el device descriptor, la memoria carga el driver, guardando en el device descriptor los punteros a las **funciones del código**, entre las cuales todos los dispositivos tienen: open, write, read, close... **Los device descriptors son todos iguales, por lo que cada función se guarda en la misma posición**, todos los read, etc.

Para asociar un dispositivo hardware a uno lógico, creamos un dispositivo lógico de caracteres o de bloques, asociando un device descriptor a un inodo, y para ello utiliza la llamada al sistema **mknod**, con 4 parámetros:

- b / c, que indica blocks o chars, por ejemplo el teclado es de chars.
- Major del device descriptor que quiere utilizar.
- Minor del device descriptor.
- Nombre del driver.

Para asociar un dispositivo lógico a uno virtual, se utiliza la llamada a sistema **open,** que no abre un fichero sino que crea una instancia de dispositivo lógico, creando una entrada en la tabla de inodos, en la TFA y en la TC. Los parámetros son:

- Ruta del dispositivo lógico
- Modo apertura (leer, escribir...)
- Flags adicionales, como permisos, etc.

Comprueba: Esta función coge la ruta, atraviesa el grafo de directorios, hasta llegar al inodo correspondiente a este dispositivo lógico. Comprueba si existe, si no, da error. También comprueba que son correctos el major y el minor.
Crea entrada en tabla inodos y TFA: Ahora, accede a la tabla de inodos y crea una entrada para guardar el inodo y el puntero a su device descriptor. Una vez hecho esto, accede a la TFA (Tabla de Ficheros Abiertos), y asigna una entrada nueva, definiendo el puntero R/W, asignando el puntero al inodo, y 0 en #refs.
Crea entrada en TC: Por último, accede a la TC a través del puntero almacenado en el PCB y crea una entrada, que apunta a su vez a la entrada creada en la TFA, a la que incrementa en 1 el #refs.
Devuelve el número de entrada del TC.

---

dup , dup2 y fork :

dup(int fd) duplica el contenido de el canal que se le pasa por parámetro en la primera entrada que esté libre
dup2(int fd_dest, int fd_src) duplica el contenido origen al canal destino. Sirven para management de la TC.
Si hacemos un dup de un canal, tendremos dos canales referenciando a la *misma entrada de la TFA. Por lo tanto el número de referencias a esa entrada de la TFA va a incrementar en 1.
fork duplica la tabla de canales del proceso porque crea un proceso nuevo copiando una PCB en otra vacia, con una nueva TC en la que se copia la TC del padre. Eso significa que va a duplicar el número de referencias a la entrada de la TFA.

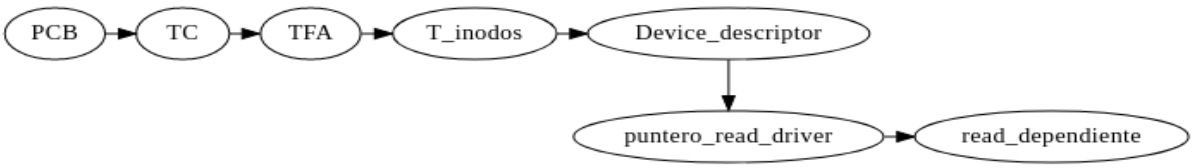
READ
Los parámetros de read son:

- Fd, el canal, file descriptor, dispositivo virtual
- Buffer, almacena los datos leidos
- Length, tamaño de estos datos

Esta función es un wrapper, que contiene la invocación del syscall_handler y de ahí la llamada a sistema sys_read con la que pasamos a modo sistema.
Por otro lado, fd nos indica la entrada de la TC, dentro del PCB del proceso, a la que queremos acceder para acceder a la TFA a través del puntero almacenado, y desde la TFA, al puntero R/W, que nos dice desde dónde leeremos la cantidad length de datos.

Después, coge el puntero al inodo desde la TFA, y accede al inodo para comprobar la validez de la instrucción read, por ejemplo si se pueden leer tantos bytes en ese fichero, etc, y desde la tabla de inodos puede también acceder al device descriptor, ya que contiene el puntero. Ahora, dentro de este accede a la función de read, para leer en el fichero.

Llama a la función que apunta el puntero de read, este código será dependiente del dispositivo. Hasta este punto, todo lo que ha ocurrido ha sido código independiente del dispositivo, incluido el acceso al device descriptor. Como programadores de SO, solo debemos implementar hasta este punto, después se encargan los fabricantes.



E/S SÍNCRONA

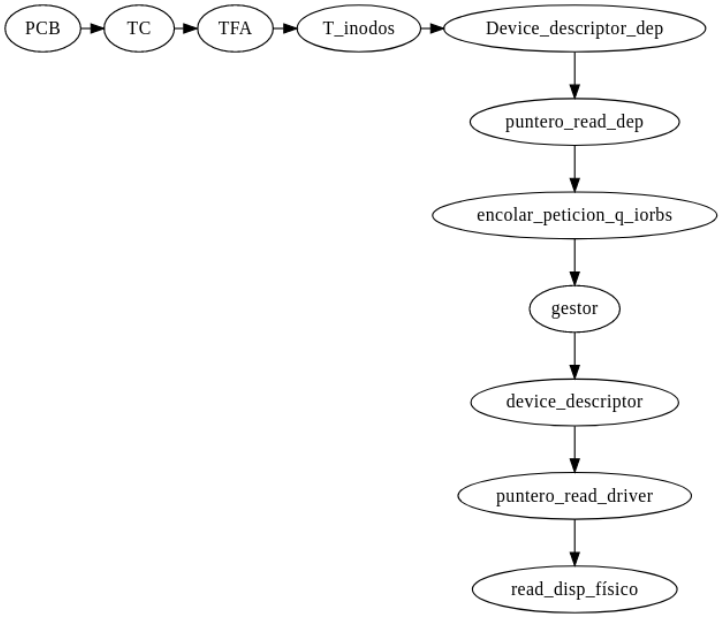
El **problema** de este proceso es que todo el acceso a tablas se hace en el sys_read en modo sistema, un acceso a HDD por ejemplo sería un proceso muy lento, por lo que afectaría gravemente al rendimiento del sistema, ya que todo esto se hace desde un mismo proceso. La solución es modificar el sys_read para solapar la E/S con la ejecución de otro proceso.

A partir de ahora no harán E/S, sino peticiones de E/S porque cuando un proceso solicite una operación E/S lo vamos a poner en el estado blocked (para operaciones de alta latencia) mientras esperan.

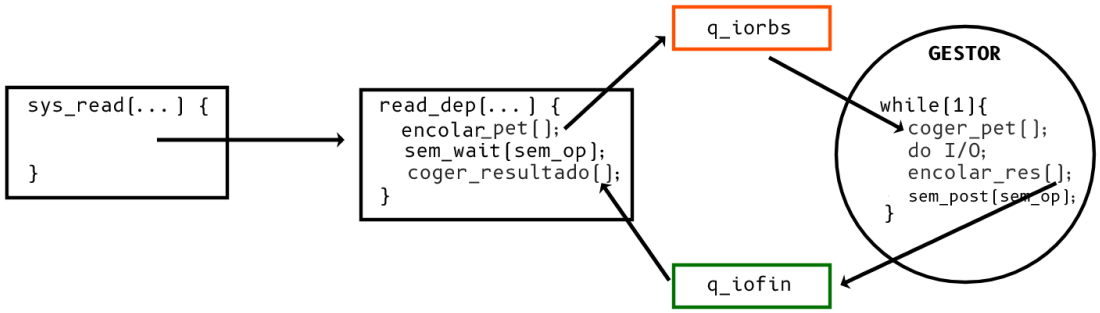
Aparece un proceso de sistema nuevo, la del gestor de E/S, que se encarga de hacer las acciones de E/S que sí acceden al modo sistema, estos gestores se crean en tiempo de boot. **A partir de ahora el puntero al device descriptor que antes apuntaba directamente al driver, apuntará a un device descriptor secundario que contiene los punteros a las funciones dependientes del gestor que apuntan a las funciones del driver del dispositivo.**

Así, el gestor es un bucle infinito que se encarga de ir aceptando y procesando peticiones de E/S según pueda para ir desbloqueando los procesos bloqueados que están esperando a recibir respuesta, de manera que liberan la CPU en esta esper, pero el gestor no se queda la latencia, porque espera una interrupción. Puede haber un gestor por dispositivo, o varios dispositivos que usen el mismo, por lo que necesitamos una cola de peticiones, la **q_iorbs** (input/output request blocks queue). Los resultados de las peticiones son devueltos a una segunda cola, la **q_iofin** (input/output finished).

Ahora el read accede a todas las tablas, y cuando va a coger el puntero al device descriptor desde Tinodos, lo que recoge es un puntero al read dependiente del gestor, que encola una petición en q_iorb, para que cuando el gestor pueda, acceda al device descriptor original y haga toda la operación, mientras el proceso original espera la respuesta en q_iofin.

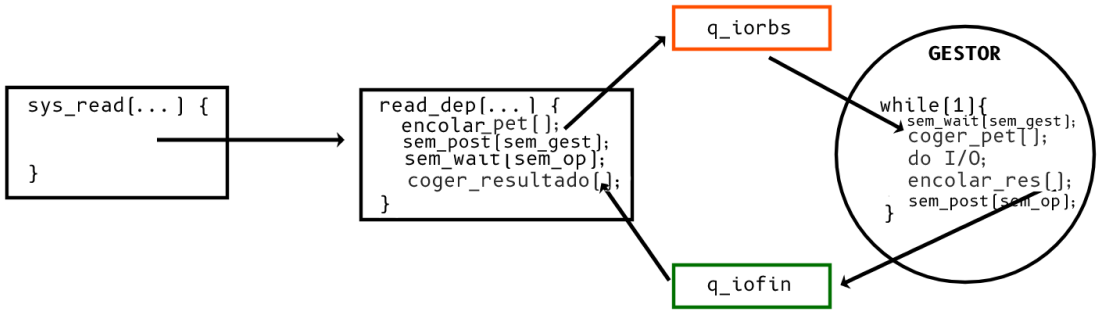


El iorb indica si es R/W y tiene un buffer para la operación. También hay un semáforo inicializado a cero en cada petición para evitar inconsistencias, y cuando se devuelve el resultado a q_iofin se hace sem_post.



El gestor no espera a que el driver acabe, sino que espera que salte una interrupción. Como tenemos semáforos en las peticiones, podemos hacer cambios de contexto sin que se pierda la consistencia, por lo que podemos solapar acciones E/S con la ejecución de otros procesos.

Si el gestor no tiene peticiones se bloquea, y cuando haya una se desbloqueará. Ponemos un sem_post después de encolar una petición, para desbloquear el gestor, y un sem_wait en el gestor para bloquearse hasta que haya una petición encolada. El sem_wait del read es un semáforo que espera a que haya resultados (cuando acaba el gestor) para continuar.



Con esto sys_read hará los accesos a todas las tablas que hemos visto antes del mismo modo; cuando llegue al punto de coger el puntero aldevice descriptor ahora lo que tendrá será un puntero para hacer peticiones al gestor. El sys_read hará un read_dependiente que encolaráuna petición al gestor en la q_iorb y se esperará al resultado en la q_iofin .

Las colas q_iorbs y q_iofin han de ser accedidas en exclusión mutua, por si otros threads alteran estas colas, necesitan semáforos inicializados a 1.

En conclusión, cada gestor necesita 3 semáforos, bloqueo de este, y dos para colas, y 1 semáforo para por petición dentro de q_iorbs, el total de semáforos será 3 + N (num de peticiones en q_iorbs).

Por eso es síncrona, porque se bloquea el thread hasta tener resultados.

E/S ASÍNCRONA
Se hace una copia de read que se llamará **async_read** y que tendrá un pequeño cambio, en lugar de bloquearse al hacer la petición, continúa ejecutándose hasta que en algún momento decide comprobar si ya tiene el resultado, con una llamada a sistema data_available, si no están los datos entonces sí que se bloquea.

SISTEMA DE FICHEROS ##
Los SO disponen de un sistema de ficheros virtual, estos ficheros y archivos no existen en ningún sitio del disco, es una abstracción. El usuario lo ve todo igual, pero en la realidad, cada dispositivo tiene un sistema de ficheros diferente, NTFS, ext4...

La tabla de inodos del VFS hace referencias a inodos virtuales. La función del VFS es relacionar estos inodos virtuales con los inodos originales. Un punto de montaje es un punto que indica que a partir de este el árbol de directorios ya no hace referencia al mismo disco, y puede que se lea de diferente manera. Para un VFS un sistema de ficheros es un dispositivo lógico como cualquier otro que hace referencia a un driver.
Si por ejemplo C:\ es un punto de montaje, su major será *mountpoint*, y le permite ir al device descriptor que gestiona los puntos de montaje, con los majors el manager le dice al SO que se lee un sistema de ficheros diferente, y con el minor el punto de montaje que es dentro de la tabla de puntos de montaje del driver.

Un punto de montaje es un conjunto de datos que indica a qué dispositivo se está refiriendo, qué sistema de ficheros tiene y, por lo tanto ,qué device descriptor debo utilizar para trabajar con el dispositivo.

El VFS va recorriendo el árbol mirando el major y el minor, y cuando encuentra un *mountpoint* habla con el driver para saber cómo seguir leyendo el filesystem. **Las syscalls, pues, acaban usando llamadas específicas para cada filesystem.** El VFS tiene una caché de inodos virtuales que referencian a los originales, trabaja con inodos virtuales, pero cuando debe pasar información a un dispositivo lógico, le pasa el puntero específico del inodo que tiene los datos necesarios.