

APUNTE TP4 - SISTEMAS OPERATIVOS

Filesystems

INTRODUCCIÓN

No debería haber conflictos. Correr las pruebas del tp3 después de armar la integración para el tp4 (se hace corriendo grade-lab4).

¿Que es un sistema de archivos? Un file system es una forma de organizar información particularmente en un medio persistente. Cada file system tendrá su propia estructura.

Interfaces de almacenamiento de datos

- Distintas interfaces (depende del dispositivo):
 - IDE/ATA (como los discos ópticos)
 - usa direccionamiento I/O del CPU, con instrucciones específicas (e.g. in o out)
 - NVMe (como discos de estado sólido)
 - soporta usar memoria mapeada a I/O, con acceso usando las mismas instrucciones de memoria
- Distintos esquemas:
 - Polling vs Interrupciones
 - PIO (programmed I/O, CPU accede a disco directamente) vs DMA (direct memory access)

Cada file system tiene su forma de organizar un dispositivo de almacenamiento. Hay distintas interfaces, por ejemplo IDE/ATA y NVMe (discos sólidos). Se diferencian porque NVMe soporta usar memoria mapeada a I/O.

Hay distintos esquemas:

- Polling (preguntarle muchas veces al disco, es activo) vs interrupciones (dejo que el disco me avise mediante una interrupción si termino)
- PIO vs DMA

Un driver de disco está formado por el tipo de disco y el esquema a llevar a cabo en el. Pero, ¿cómo manejamos el disco? ¿Qué pasa de ese driver para bajo? Debe ser completamente independiente la forma de manejar discos a el manejo de file system. El file system es la forma de estructurar datos y el disco es donde se guardan los datos - el file system administra las direcciones del disco -.

Bloques vs. Sectores

- Los dispositivos de almacenamiento operan por **sectores**
 - Unidad mínima en la que opera el **hardware**
- El SO lo abstrae en **bloques**
 - Unidad lógica del **software**
 - Múltiplo del tamaño de un sector
- En JOS:
 - Sectores de 512 B
 - Bloques de 4096 B (mismo que una página)



El file system dependiendo de cuantos drives tiene va a soportar distintos discos.

¿Que es un sector? Bloquecitos del disco. Cantidad de bytes contiguos que se puede acceder de forma **atómica** al disco. Vienen de la implementación de los discos duros, por eso se llaman sectores.

El SO lo único que tiene que saber, es que el disco tiene sectores.

Se define una interfaz, especificación. Se le pide a los drivers del disco que implemente primitivas para almacenar y operar sectores (unidad mínima que opera el hardware).

Disco → es un conjunto de sectores, el SO necesita primitivas para interactuar con los sectores (definidos por el hardware). El disco provee una interfaz para operar sobre sectores.

El mapeo se mapea a nivel file system.

El SO tiene una abstracción virtual que se llama bloque:

- Unidad lógica del software
- Múltiplo de sectores

El mapeo se mapea a nivel file system. Puede ser 1 sector = 1 bloque, o 1 sector = múltiplo de sectores.

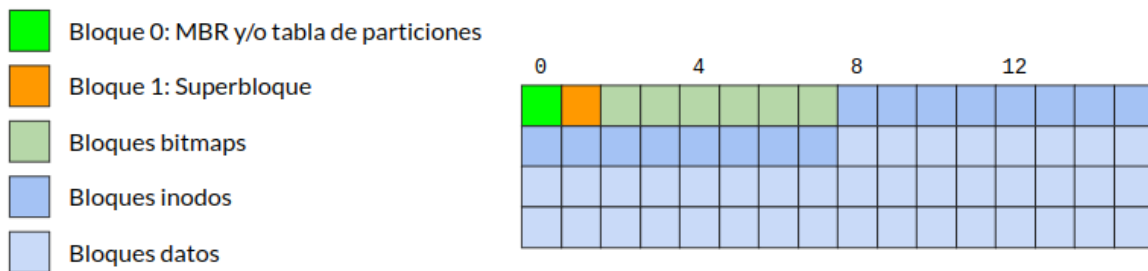
El bloque se define por software.

JOS tiene un drive de disco con 512 bytes por sector y bloques de 4096 bytes (mismo tamaño que una página de memoria virtual).

Como hago una estructura que tenga sentido, como organizar bloques que permitan de forma eficiente almacenar archivos, con estructura jerárquica, etc (???????)

Las respuestas, dependen del SO.

Estructura de un filesystem (unix-like)



El sector es el **único** que depende del disco.

Diseño de un SO

- Sistema de archivos UNIX
- JOS se adapta a este diseño de UNIX

Recordar que los bloques sean múltiplos de los sectores

512 tamaño que leía la BIOS de un disco de memoria

Disco → es una secuencia de bloques.

Cada bloque tiene una dirección (0,1,2...N) y se tiene que organizar de una forma, JOS lo hace de forma jerarquizada. Para lograr esa estructura utiliza lo que es un inodo.

Cada bloque tiene un inodo.

BLOQUES

Bloque 0: El primer bloque de todos, se reserva para almacenar al bootloader. Se reserva para el MBR o la tabla de particiones. Se guarda la metadata.

Bloque 1: Superbloque: La metadata del file system. “Aca este disco está formateado con tal esquema, tiene tantos bitmap de datos, tantos bitmap de inodos”. De este bloque se saca la información concreta del FS. Ej: Cuántos bloques tiene, que es lo que tiene ocupado o no, etc

Bloques bitmaps: Bitmaps de datos: es una estructura de datos donde cada bit representa un dato único, un dato binario y cada bit corresponde a un elemento separado. Señala si el próximo bloque está ocupado o no, en la forma de 0 o 1. Se necesita para llevar la cuenta de cuánto espacio libre hay y saber que bloque tomar para saber donde guarda un nuevo archivo o hacer un ABM a un archivo.

Bloques inodos: (Index Node), guarda metadata sobre los datos de los bloques. Se necesita saber que archivo existe, por lo cual cada archivo tendrá un inodoro que lo identifica. Es un mapeo uno a uno de un archivo o directorio.

Bloques datos: Son datos, por ejemplo, un película guardada el titanic y se guarda de forma binaria en este bloque. Algo que está guardado en el file system, sus datos estarán guardados en algún bitmap de datos.

Directorio → lista de archivos q son inodo (nombre de archivo y en qué nodo encontrarlo)

Límite del disco de implementación de 3G en JOS
Cada archivo es un inodo.

Se tendrán que elegir los parámetros para el disco.

Pregunta: ¿En los bloques de inodo, cuántos bloques hay?

Respuesta: File System nos dice que hay inodos, bitmaps y datos. ¿Cuántos inodos, bitmaps y datos hay? Nada de eso está especificado. Todo esto serán **parámetros** que se tienen que especificar y vamos a poder meter en el disco. Un inodo es más chico que un tamaño de bloque, cosa de que un bloque entren múltiples inodos. Tamaño de inodo en JOS en el Struct Definido de Inodo

Estructura de un filesystem

- El superbloque apunta al inodo del directorio raíz
- Los **inodos** contienen la información de qué **bloques** componen a un archivo/directorio
 - Entre directos e indirectos
- Los **bloques de datos** contienen:
 - para **archivos**: los datos sin formato
 - para **directorios**: una lista de nombres a **inodos**
- Los **bitmaps** nos dicen qué bloques de datos o inodos están libres

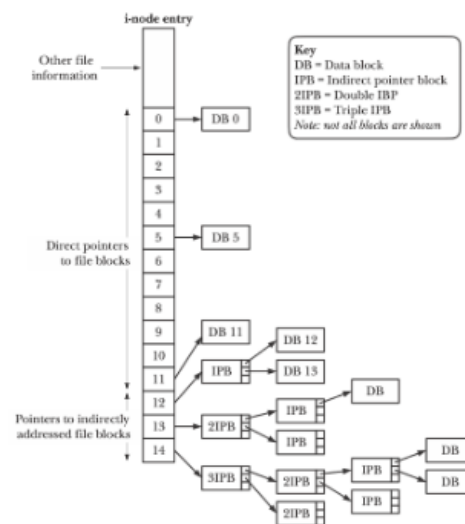


Figure 14-2: Structure of file blocks for a file in an ext2 file system

importante entender cómo afecta la performance en tu disco según los parámetros que elijas

La **metadata** del inodo impacta en el ratio de cuantos inodos por bloques podrás

tener (acordarse que un **archivo es un inodo**)

Ejemplo de metadata: Nombre del usuario, color de archivo, etc

Trade Off (balance): Si se que tengo inodos que pesan mucho, vos a configurar los bloques para almacenar este tipo de inodos y guardarme espacio en memoria ya que por inodo se creará un bloque. De esta forma si tengo muchos inodos que pesan poco, tengo que compensar el no quedarme sin memoria para los inodos, aunque haya bloques disponibles.

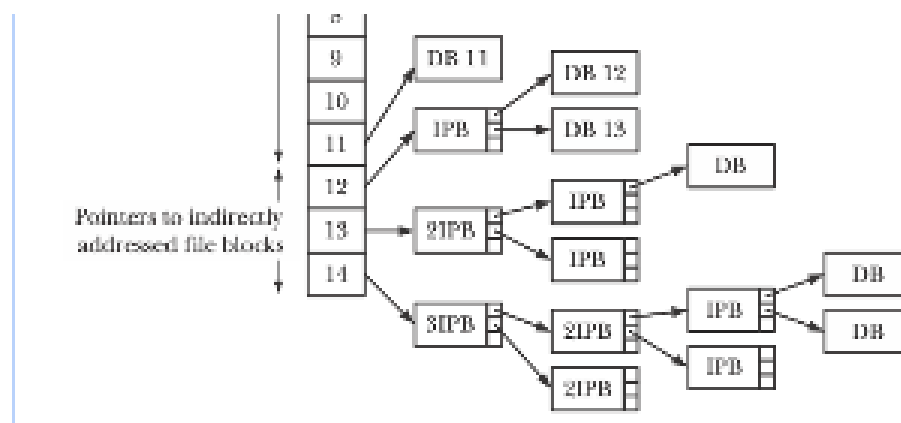
/ directorio raíz

El superbloque apunte al directorio raíz, dice q inodo es el directorio raíz

si tengo un archivo grande necesito varios bloque con inodos que vayan apuntando, una lista de bloques es demasiado grande (lo q se ve en pantalla)

esto no es bueno, se busca que el inodo tenga un tamaño predecible para poder manejarlo, se soluciona teniendo un tamaño grande de bloques para que no exceda el tamaño de inodos

Bloques indirectos: Se almacenan números de bloques que continúan una cadena de bloques de un archivo específico. Esto se hace para que no crezca la lista de bloques directos asociado al inodo (en la imagen se aprecia como los bloques 12, 13 y 14 apuntan a bloques indirectos). Logramos que el inodo tenga un tamaño fijo y chico, pero podemos exponencialmente hacerlo crecer para que almacene archivos de múltiples sheets.



Bloque 12 un nivel de indirección.





Bloque 13 dos niveles de indirección

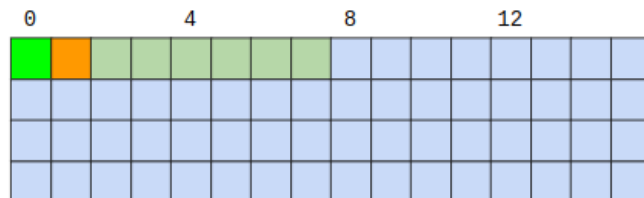
Bloque 14 tres niveles de indirección

Bloque directo = Bloque indirecto = Bloque de datos.

JOS

Estructura de un filesystem en JOS

-  Bloque 0: MBR y/o tabla de particiones
-  Bloque 1: Superbloque
-  Bloques bitmap
-  Bloques datos



JOS → DIRECTORIO = INODO

Estructura de un filesystem JOS

- No hay inodos: los directorios funcionan como inodos
 - El Struct File es un inodo y una directory entry a la vez
- Los bloques de datos contienen:
 - para archivos: los datos sin formato
 - para directorios: una lista de referencias a archivos (incluyendo en qué bloques están)
- Los bitmaps nos dicen qué bloques de datos están libres

```
struct File {
    char f_name[MAXNAMELEN];    // filename
    off_t f_size;                // file size in bytes
    uint32_t f_type;            // file type

    // Block pointers.
    // A block is allocated iff its value is != 0.
    uint32_t f_direct[NDIRECT];  // direct blocks
    uint32_t f_indirect;         // indirect block

    // Pad out to 256 bytes;
    // must do arithmetic in case we're compiling
    // fsformat on a 64-bit machine.
    uint8_t f_pad[256 - MAXNAMELEN - 8 - 4*NDIRECT - 4];
}
```

El directorio en JOS es un arreglo, donde hay un struct File detrás de otro. Cada uno de ellos representa a un nuevo directorio o un archivo regular. El struct File es un inodo y una directory entry a la vez.

```
// JOS
directory = File[]

struct File {
    char f_name[MAXNAMELEN];    // filename
    off_t f_size;                // file size in bytes
    uint32_t f_type;            // file type

    // Block pointers.
    // A block is allocated iff its value is != 0.
    uint32_t f_direct[NDIRECT]; // direct blocks
    uint32_t f_indirect;        // indirect block

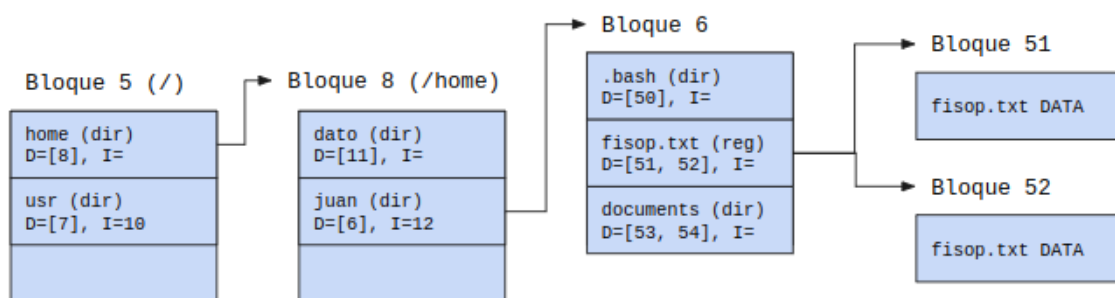
    // Pad out to 256 bytes;
    // must do arithmetic in case we're compiling
    // fsformat on a 64-bit machine.
    uint8_t f_pad[256 - MAXNAMELEN - 8 - 4*NDIRECT - 4];
}
```

En JOS no se implementa el HardLink.

El límite de archivos por directorio, es la cantidad de struct files que entran por bloques de datos.

En JOS hay 10 bloques directos y 1 bloque indirecto.

Ejemplo, acceso a /home/juan/fisop.txt

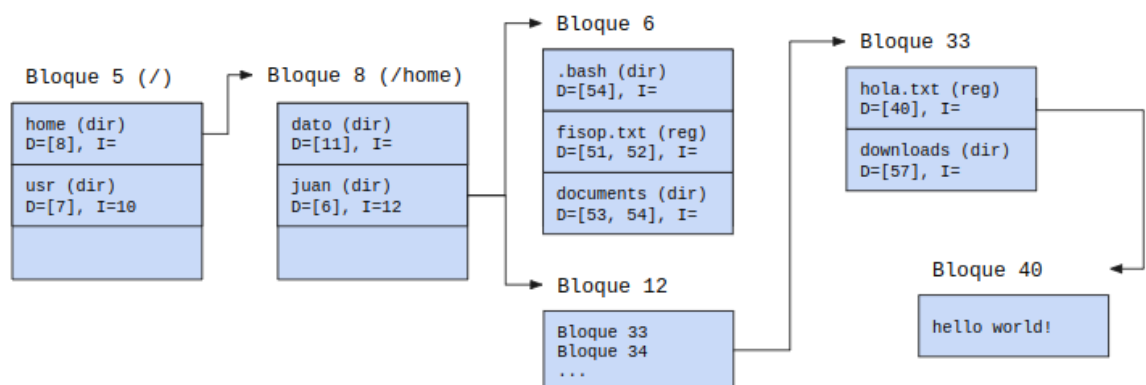


Si se quiere cargar home/juan/fisop.txt:

1. **El Superbloque** me indica que mi directorio raíz está en el bloque 5.

2. **Bloque 5:** Está contenido los directorios de bloque raíz (/). Tiene el archivo home (de tipo directorio, que tiene el bloqueo directo 8 y no tiene bloques indirectos) y al archivo usr (de tipo directorio, que tiene el bloque directo 7 y al bloque indirecto 10). Se lee el directorio de home. Esto obtiene el bloque 8
3. **Bloque 8:** Se espera un bloque con el mismo formato del bloque 5. Se tiene que interpretar como un bloque de datos de directorio.
Vamos a tener una lista de archivos y encontramos el directorio Juan.
Juan es un directorio, que tiene el bloque directo 6 y el bloque indirecto 12.
Se revisa uno a uno, a ver donde está lo que buscamos. La preferencia está sobre los directos ya que por convención están así en el inodo, y los bloques indirectos están referenciados a partir de los directos.
4. **Bloque 6:** Tiene tres archivos. .bash que es un directorio que tiene el bloque directo 50 y no posee bloques indirectos. El archivo fisop.txt que es un archivo regular que posee los bloques directos 51 y 52. Y por último, documents que es un directorio que posee a los bloques directos 51 y 52, y no posee bloques indirectos.
Se lee el bloque 51 y 52, ambos directos. Lo que se va a esperar es que contenga la data que tiene ese archivo.
5. **Bloque 51:** Contiene la data
6. **Bloque 52:** Contiene la data

Ejemplo, acceso a /home/juan/hola.txt



Si se quiere cargar home/juan/hola.txt:

1. **El Superbloque** me indica que mi directorio raíz está en el bloque 5.
2. **Bloque 5:** Está contenido los directorios de bloque raíz (/). Tiene el archivo home (de tipo directorio, que tiene el bloqueo directo 8 y no tiene bloques

indirectos) y al archivo usr (de tipo directorio, que tiene el bloque directo 7 y al bloque indirecto 10). Se lee el directorio de home. Esto obtiene el bloque 8

3. **Bloque 8:** Se espera un bloque con el mismo formato del bloque 5. Se tiene que interpretar como un bloque de datos de directorio.

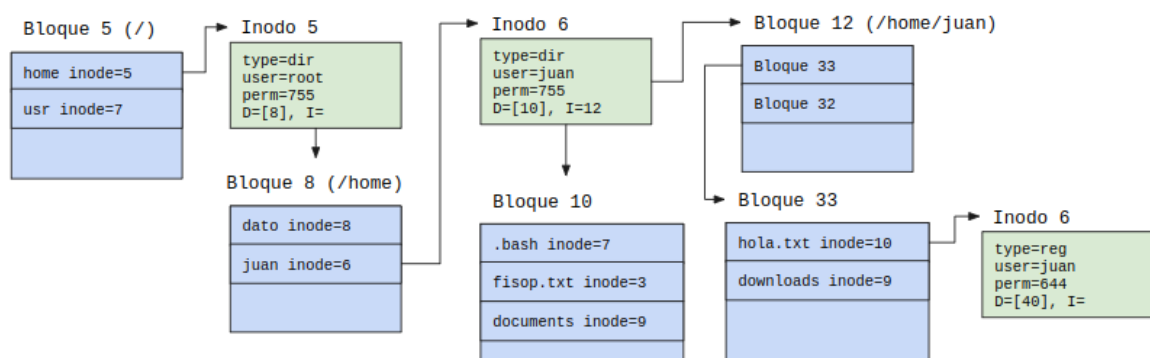
Vamos a tener una lista de archivos y encontramos el directorio Juan.

Juan es un directorio, que tiene el bloque directo 6 y el bloque indirecto 12.

Se revisa uno a uno, a ver donde está lo que buscamos. Dado el ejemplo anterior, ya sabemos que no está en el bloque directo 6. Por lo que buscamos en el bloque indirecto 12.

4. **Bloque 12:** ¿Qué esperamos encontrar en el bloque 12? Va a tener punteros a mis bloques. Bloque 33, bloque 34,..., bloque n. Recorremos uno a uno.
5. **Bloque 33:** Esta el archivo regular hola.txt que contiene al bloque directo 40.
6. **Bloque 40:** posee la data del archivo hola.txt

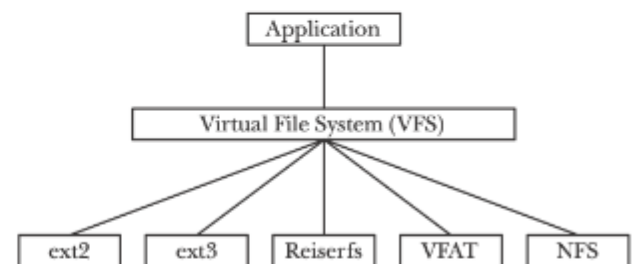
¿Cómo sería con inodos?



1. **Bloque 5:** Busca el inodo 5
2. **Inodo 5:** Pasa al bloque 8
3. **Bloque 8:** Se manda al inodo 6
4. **Inodo 6:** Te manda al bloque 10
5. **Bloque 10:** No tiene nada, pasa al bloque indirecto 12
6. **Bloque 12:** Se busca el bloque 33
7. **Bloque 33:** Me referencia al inodo 10
8. **Inodo 10:** Datos

Virtual File System de Unix

- Una **interfaz** entre cualquier implementación de filesystem y userspace
 - Conjunto de syscalls: **read**, **write**, **open**, etc.
- El kernel puede incluir múltiples implementaciones de filesystem (incorporadas o como módulos) y elegir cual usar
- Incluso se puede **delegar al usuario** (FUSE)
 - El kernel intercepta las syscalls pero las "redirige" a un proceso de usuario



Todas las syscall termina en el virtual file system (VFS)
Es la interfaz entre las aplicaciones y los distintos discos

Discos en JOS

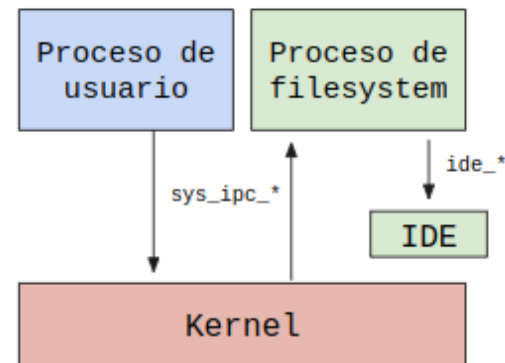
- Se usa qemu para emular **2 discos**:
 - El disco 0 -> usado **únicamente en boot** con la imagen del kernel
 - El disco 1 -> usa una **imagen separada**
- JOS implementa un controlador IDE, **sin interrupciones**
- ¿Cómo operamos con disco? ¿Qué interfaz se proveerá al usuario?
- ¿Qué es un **file descriptor**? ¿Qué es **VFS**?



Se implementará un driver y un file system. El driver ya está hecho
se tendrá que hacer las primitivas para utilizar este driver

El proceso de FS - concepto

- En JOS no hay syscall de acceso a disco (no hay VFS)
- Existe un proceso con **privilegios especiales** que se encargará de manejar acceso a disco (envfs)
 - Environment de tipo FS
- Un proceso se comunica con **envfs** usando IPC
- **envfs** accede **directamente** a disco
 - ¿Qué privilegios necesita?



JOS va a funcionar como un proceso de usuarios con privilegios que va a controlar el disco. El kernel le va a dar potestad para que acceda al driver a las funciones de disco.

Con el mecanismo de IPC se maneja la comunicación de los open, read, etc un flags en eflags que permite interactuar con procesos de hardware para interactuar con disco

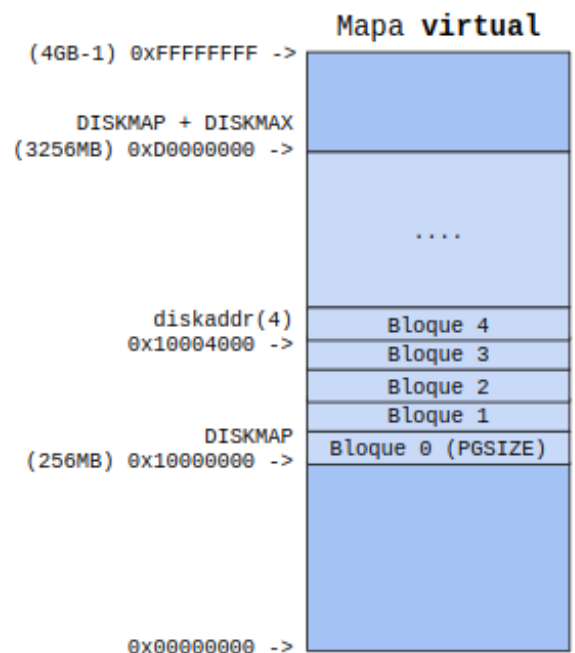
Va haber 2 tipos de procesos, los procesos de usuario convencionales y el proceso de tipo filesystem (que hay uno solo) que corre en ring 3 en el espacio de usuarios. Este proceso filesystem será parte del sistema operativo, JOS lo va a lanzar ni bien arranca (es una de sus últimas tareas de JOS en el arranque, es verificar que este proceso de usuario esté corriendo) y no le va a permitir al usuario crear más procesos del tipo filesystem. El kernel es quien tiene control sobre ese proceso.

¿Cómo se implementa el proceso de file system? [forma de JOS]

Proceso FileSystem: va a implementar un caché de Disco (va interactuar directamente con Disco). Se van a mantener guardados los datos en memoria, y se encargará de que esté actualizado con lo que esté en el disco. Escribir y leer en disco es muy lento. Esto produce una ventaja de lectura y escritura rápida. Este caché es lo que hace muy eficiente al proceso, esta caché no es necesario pero si aporta eficiencia.

Cache de bloques

- Estrategia de disco mapeado en memoria
 - 1 página = 1 bloque
- Se mantiene un mapeo lazy de todo el disco en memoria! (simil COW)
 - Máximo de 3Gib (espacio reservado)
 - Utiliza su propio pgfault handler!
- El proceso **envfs** recupera y flushea a disco las páginas para persistir las modificaciones
 - ¡Aprovechar el uso de **PTE_D** para saber si hubo modificaciones!



Cada bloque estará mapeado en memoria ram (direcciones virtuales).

1 página = 1 bloque (es acá donde está el por qué un bloque equivale a 4096 bytes)

Esto nos permite encontrar de forma fácil los bloques y sus mapeos.

Ej **bloque 4 = DISKMAP + 4**

Leer un bloque se traduce en leer una dirección en memoria.

Copy on write para el tema de que se pida una lectura sobre algún bloque.

Caché de mapeo directo: mapeo uno a uno. Una posición de caché, puede mapear a una unica posicion de disco.

En JOS hay 3GB para mapear el disco. Es una decisión de diseño.

Es un mapeo lazy. Una de las desventajas es que una vez que se hace un mapeo de una página, no se va a sacar.

La MMU va a poner el PTE_D en 1, si alguien escribió la página.

Bitmap de bloques

- Bitmap **compacto**
 - Cada bloque está representado por 1 bit
- Se almacena en disco!
 - Alguno de los bloques será el bitmap
- Indica qué bloques están libres
 - Se usan para alocar espacio para archivos nuevos/borrar archivos
- Ejemplo:
 - bloque 5 = ???
 - bloque 23 = ???
 - bloque 40 = ???

	4				0			
bitmap								
bitmap+1								
bitmap+2								
bitmap+3								
bitmap+4								
bitmap+5								

Uno de esos bloques de memoria, va a ser el bitmap. Bitmap es un arreglo de enteros. Cada fila es un byte (8 bits). En cada uno, hay un 1 o un 0 que indica si está ocupado o no respectivamente.

Bitmap de bloques

- Bitmap **compacto**
 - Cada bloque está representado por 1 bit
- Se almacena en disco!
 - Alguno de los bloques será el bitmap
- Indica qué bloques están libres
 - Se usan para alocar espacio para archivos nuevos/borrar archivos
- Ejemplo:
 - bloque 5 = ???
 - bloque 23 = ???
 - bloque 40 = ???

	4				0			
bitmap			5					
bitmap+1								
bitmap+2	23							
bitmap+3								
bitmap+4								
bitmap+5								40

Bitmap de bloques

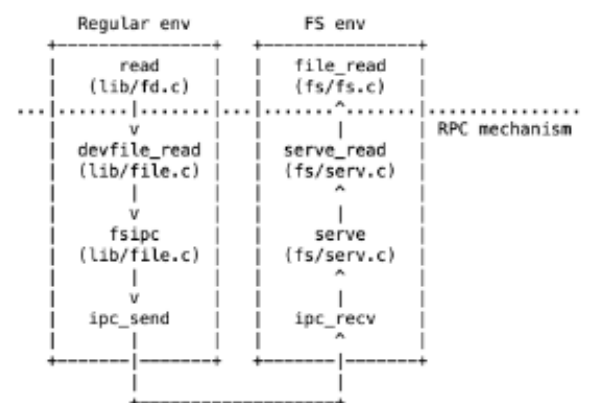
- **Bitmap compacto**
 - Cada bloque está representado por 1 bit
- Se almacena en disco!
 - Alguno de los bloques será el bitmap
- Indica qué bloques están libres
 - Se usan para alocar espacio para archivos nuevos/borrar archivos
- Ejemplo:
 - bloque 5 => $\text{bitmap}[0] \mid = 1 \ll 5 (32)$
 - bloque 23 => $\text{bitmap}[2] \mid = 1 \ll 7 (128)$
 - bloque 40 => $\text{bitmap}[5] \mid = 1 \ll 1 (1)$
 - bloque N => ???

	4				0			
bitmap	0	0	1	0	0	0	0	0
bitmap+1	0	0	0	0	0	0	0	0
bitmap+2	1	0	0	0	0	0	0	0
bitmap+3	0	0	0	0	0	0	0	0
bitmap+4	0	0	0	0	0	0	0	0
bitmap+5	0	0	0	0	0	0	0	1

BLOQUE 40 => $\text{bitmap}[5] \mid = 1 \ll 0 (0)$

El proceso de FS - interfaz IPC

- **envfs** se queda looppeado escuchando requests (**ipc_rcv**)
- Los proceso de usuario envían requests
 - El **tipo** de operación (open, read, write, etc) se codifica con el **entero** de IPC
 - La **data** (e.g. buffer para recibir lectura) se **comparte** en una página mapeada
- La información viaja en la página mapeada en forma de **union FSIPC**



El proceso de FS - ejemplo fsipc

```
// Read request
struct Fsreq_read {
    int req_fileid;
    size_t req_n;
} read;
```

```
// Read response
struct Fsret_read {
    char ret_buf[PGSIZE];
} readRet;
```

```
// Write request
struct Fsreq_write {
    int req_fileid;
    size_t req_n;
    char req_buf[PGSIZE - (sizeof(int) + sizeof(size_t))];
} write;
```

Struct Fsreq_read

- req_fileid: de que archivo voy a querer leer
- req_n: cuántos bytes voy a querer leer

Struct fsret_read

- ref_buf[PGSIZE]:

Struct fsreq_write

- req_fileid: número de archivo
- req_n: le paso el tamaño de lo que quiero escribir
- req_buf[.]: le embebo en la misma página los datos que quiero escribir

El proceso de FS - file descriptors

- envfs mantiene una lista de archivos abiertos (struct Openfile)
 - Cada archivo abierto tiene su propio identificador dentro de envfs
- Cada proceso **recibe** una referencia a ese archivo abierto, y lo guarda en una **tabla de file descriptors**
 - Implementado en la librería de JOS
 - El proceso de usuario opera con el índice en su tabla (e.g. 0, 1, 2, 3) pero usa el ID de envfs para los requests

Proceso de usuario	
0	empty
1	empty
2	dev=x fileid=102

envfs	
...	
102	file=addr
...	

Por último, se va a tener una tabla de archivos abiertos **[envfs]**, que va a tener el id (102) y donde esta (va a guardar la dirección virtual que mapea al bloque donde está guardado el archivo que uno quiere referenciar). Por lo tanto, el proceso de file system se guarda para todos los archivos que se le pidió que se abran, les va a generar un id único y va a guardar la dirección. En linux es lo mismo, pero almacena el inodo.

Después, cada proceso de usuario cuando se haga un open va a guardar su propia tabla de fs **[Proceso de usuario]**. Que es donde va a guardar los descriptors de archivos, donde va a mapear, tu archivo abierto en tu descriptor de archivos 2, en realidad es el dispositivo de disco x y ese dispositivo de disco tiene el id 102.

Entonces, el proceso de usuario regular tiene una tabla de descriptors de archivo. Cada uno de esos descriptors (0,1,2) tiene una referencia a que controlador maneja ese archivo (dentro del Virtual FileSystem) y cual es el id interno de ese archivo abierto.

En cambio, el proceso de file system va a tener una tabla de procesos abiertos y cada tabla de archivo abierto va a matchear los ids contra la posición en el disco (la dirección de disco) donde está mapeado el struct file correspondiente.

Cuando al file system le llegue una request del 0 o 1 (del proceso de usuario), no lo va a mandar a disco, si no que a una parte que maneje stdin o stdout. Podría haber un stderr, pero en JOS no lo hay.

A través del device (dev) es como la librería estándar de JOS sabe a quién mandarle el request.

La diferencia de esto entre JOS y linux, es que en linux como es monolítico el kernel ambas tablas están en el kernel. En JOS la de descriptors de archivos, está en el proceso de usuario.



La librería de JOS - file descriptors y devs

- La librería de JOS soporta múltiples dispositivos (devices) para los file descriptors
- Los de tipo file van a parar al filesystem
- Están implementados otros dos: pipe y cons
 - pipe es similar a pipe(2) pero en userspace
 - cons soporta escribir directamente a una terminal
- ¿Cómo funcionan?

TEORÍA

En Unix todo es un archivo.

Un file system permite a los usuarios organizar sus datos para que se persistan a través de un largo periodo de tiempo.

Formalmente: un filesystem es una **abstracción** del SO que provee datos persistentes (se almacenan hasta que explícitamente se borran) con un nombre (para que cualquier humano pueda acceder a ellos por un identificador que el sistema de archivos asocia).

Existen dos partes fundamentales de esta abstracción:

- Los archivos → que están compuestos por un conjunto de datos
- Los directorios → que agrupan archivos y directorios.

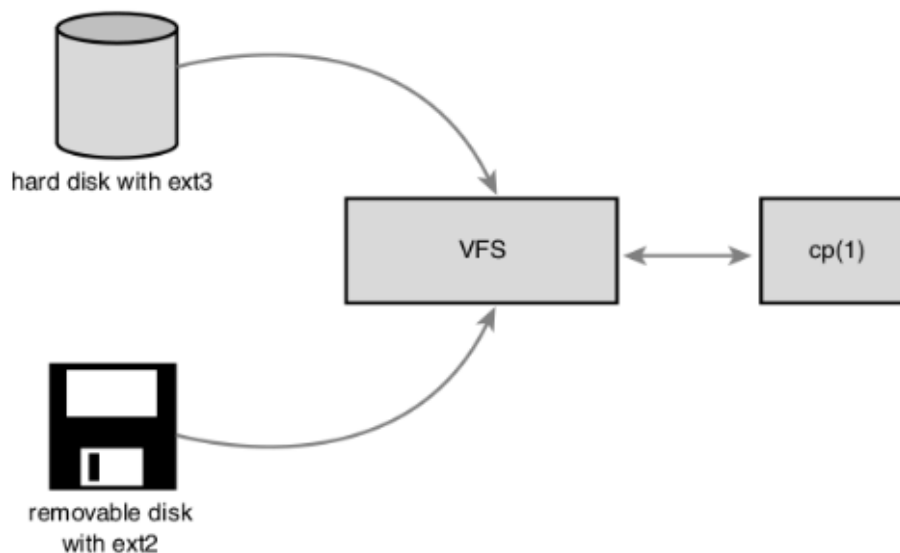
Virtual FileSystem

El **VFS** es el subsistema del kernel que implementa la interfaz que tiene que ver con los archivos y el sistema de archivos provistos a los programas corriendo en modo usuario.

Todos los sistemas de archivos deben basarse en VFS para:

- coexistir
- interoperar

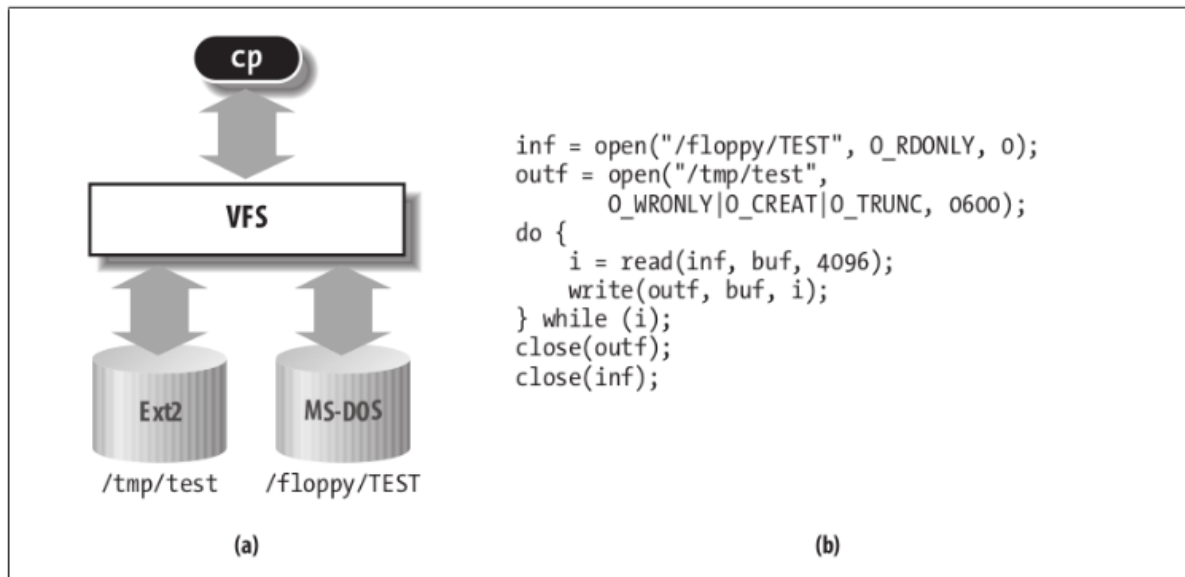
Esto habilita a los programas a utilizar las system calls de unix para leer y escribir en diferentes sistemas de archivos y diferentes medios.



VFS es el pegamento que habilita a las system calls como por ejemplo `open()`, `read()` y `write()` a funcionar sin que estas tengan en cuenta el hardware subyacente.

FileSystem abstraction Layer

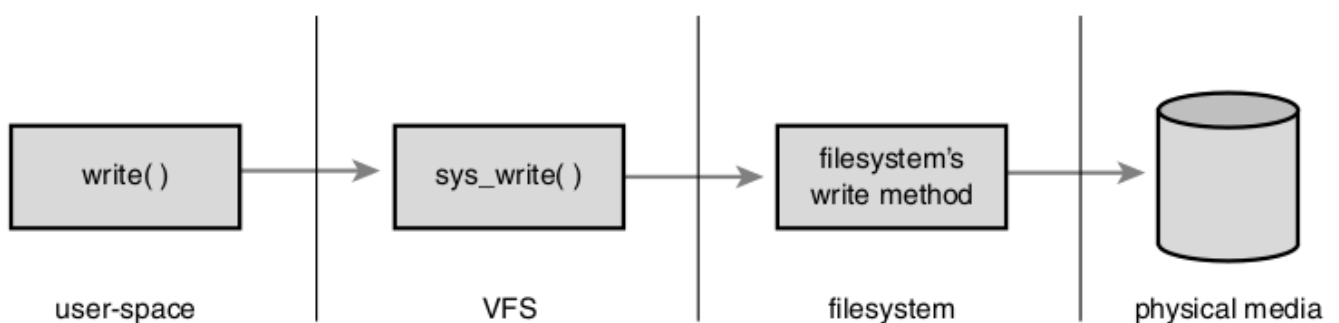
- Es un tipo genérico de interfaz para cualquier tipo de filesystem que es posible sólo porque el kernel implementa una capa de abstracción que rodea esta interfaz para con el sistema de archivo de bajo nivel.
- Esta capa de abstracción habilita a Linux a soportar sistemas de archivos diferentes, incluso si estos difieren en características y comportamiento.
- Esto es posible porque VFS provee un modelo común de archivos que pueda representar cualquier característica y comportamiento general de cualquier sistema de archivos.



El resultado de una capa de abstracción general le permite al kernel manejar muchos tipos de sistemas de archivos de forma fácil y limpia.

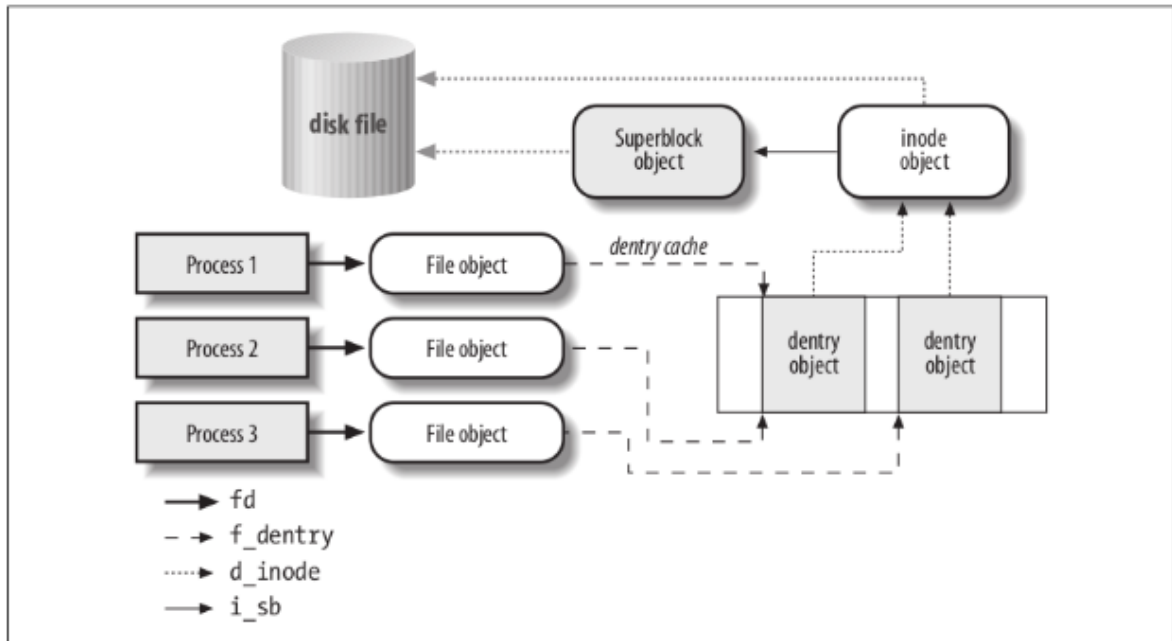
Los filesystems amoldan su visión de conceptos como “esta es la forma de cómo abro un archivo” para matchear las expectativas del VFS, todos estos sistemas de archivos soportan nociones tales como archivos, directorios y además todos soportan un conjunto de operaciones básicas sobre estos.

El resultado es una capa de abstracción general que le permite al kernel manejar muchos tipos de sistemas de archivos de forma fácil y limpia.



Los objetos (sus estructuras) de VFS:

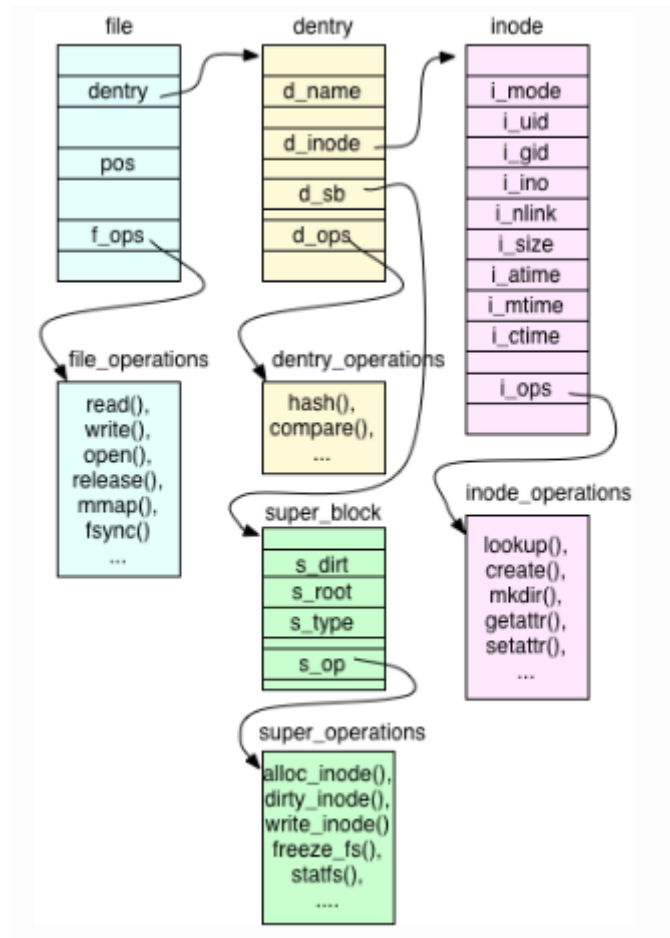
- El **super bloque**, que representa a un sistema de archivos.
- El **inodo**, que representa a un determinado archivo.
- El **dentry**, que representa una entrada de directorio, que es un componente simple de un path.
- El **file** que representa a un archivo asociado a un determinado proceso



No existe objeto directorio conceptualmente, es tratado como un archivo normal que listan a los archivos contenidos en ellos.

Operaciones:

- Las **super_operations** métodos aplica el kernel sobre un determinado sistema de archivos, por ejemplo `write_inode()` o `sync_fs()`.
- Las **inode_operations** métodos que aplica el kernel sobre un archivo determinado, por ejemplo `create()` o `link()`.
- Las **dentry_operations** métodos que se aplican directamente por el kernel a una determinada directori entry, como por ejemplo, `d_compare()` y `d_delete()`.
- Las **file_operations** son métodos que el kernel aplica directamente sobre un archivo abierto por un proceso, `read()` y `write()`, por ejemplo.



Archivos

Un archivo es una colección de datos con un nombre específico. Los archivos proveen una abstracción de más alto nivel que la que subyace en el dispositivo de almacenamiento; un archivo proporciona un nombre único y con significado para referirse a una cantidad arbitraria de datos.

Además, la información que se almacena en un archivo se divide en dos partes:

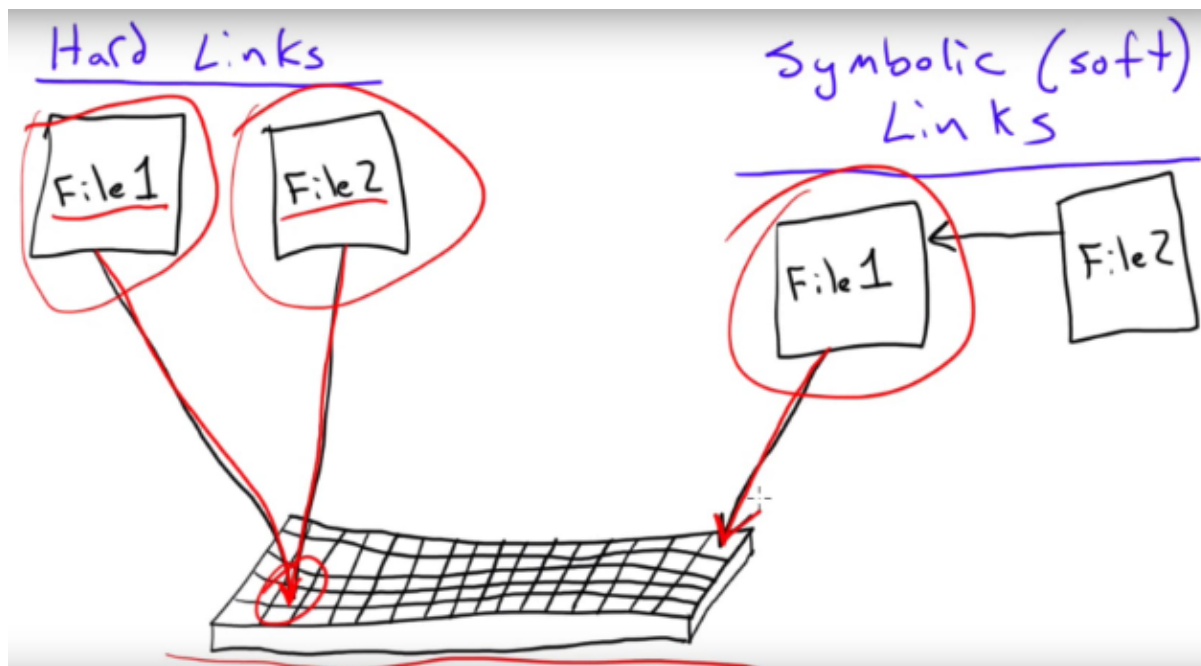
1. Metadata: información acerca del archivo que es comprendida por el Sistema Operativo, esta información es :
 - a. tamaño
 - b. fecha de modificación
 - c. propietario
 - d. información de seguridad (que se puede hacer con el archivo).
2. Datos: son los datos propiamente dichos que quieren ser almacenados. Desde el punto de vista del Sistema Operativo, un archivo o file no es más que un arreglo de bytes sin tipo.

Directorios

Los directorios proveen los nombres para estos archivos. En particular un directorio es una lista de nombre human-friendly y su mapeo a un archivo o a otro directorio.

Definiciones importantes

- **path o ruta** es el string que identifica unívocamente a un directorio o archivo dentro de un dispositivo.
- **root directory o directorio raíz** es el directorio de que cuelgan todos los demás.
- **absolute path** es la ruta desde el directorio raíz e.i. `"/home/mariano/prueba"`.
- **relative path** es el path relativo que se interpreta a partir del directorio actual.
- **current directory** es el directorio actual en el cual se está ejecutando el proceso.
- **hard link** es el mapeo entre el nombre y el archivo subyacente, esto implica que la estructura de un file system que permite múltiples hard links ya no es de árbol invertido. Aquellos S.O. que lo permiten se cuidan de no crear ciclos asegurándose que la estructura sea un grafo dirigido acíclico.
- **soft links** se da cuando un archivo puede ser llamado por distintos nombres.



- **Volumen** es una abstracción que corresponde a un disco lógico. En el caso más simple un disco corresponde a un disco físico. Es una colección de recursos físicos de almacenamiento.
- **mount point** es un punto en el cual el root de un volumen se engancha dentro de la estructura existente de otro file system.

La metadata es información de vital importancia ya que mantiene información como qué bloque de datos pertenece a un determinado archivo, el tamaño del archivo, etc. Para guardar esta información, en los sistemas operativos unix-like, se almacena en una estructura llamada **inodo**. Los inodos también deben guardarse en el disco, para ello se los guarda en una tabla llamada **inode table** que simplemente es un array de inodos almacenados en el disco. Un inodo simplemente es referido por un número llamado **inumber** que sería lo que hemos llamado el nombre subyacente en el disco de un archivo. Cabe destacar que los inodos no son estructuras muy grandes, normalmente ocupan unos 128 o 256 bytes.

El superbloque contiene la información de todo el file system, incluyendo:

- cantidad inodos
- cantidad de bloques
- donde comienza la tabla de inodos → bloque 3
- donde comienzan los bitmaps