Ext2 y Ext3

Las estructuras de los filesystems ext2 y ext3 son prácticamente las mismas, lo que cambia es que ext3 agrega el llamado "**journaling**" (**ver al final la definición*).

Siempre que se mencione "**bloque**", hace referencia a un bloque lógico, es decir, a un **'cluster'**. Por lo que prácticamente todo el filesystem se estructura en clusters. Por ejemplo, se asigna un cluster entero para el área de boot, otro para el superbloque (realmente el primer superbloque podría estar junto con el área de boot, **ver aclaración al final*), otro para los bitmaps, etc… (por más que luego quede espacio libre, ya que recordemos que el tamaño del cluster se define al momento de darle formato a la partición; y este queda guardado en el superbloque).

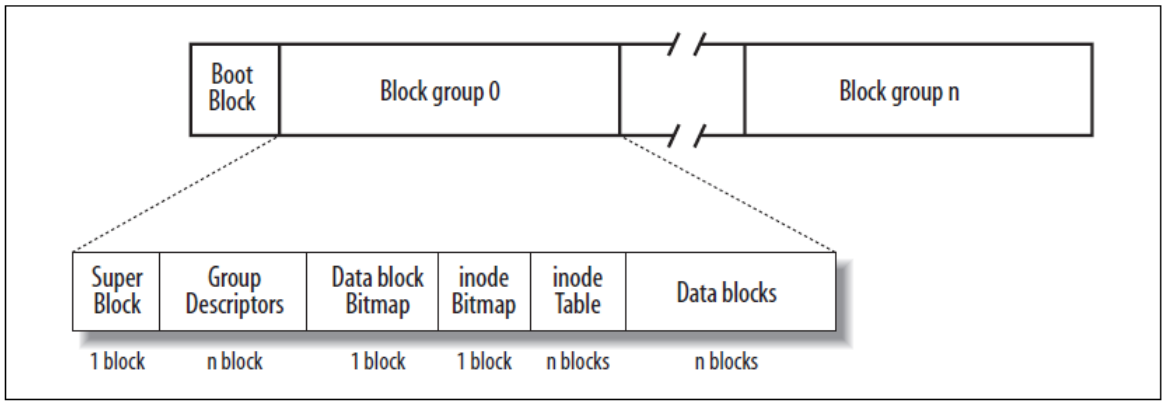


Figure 17-1. Layouts of an Ext2 partition and of an Ext2 block group

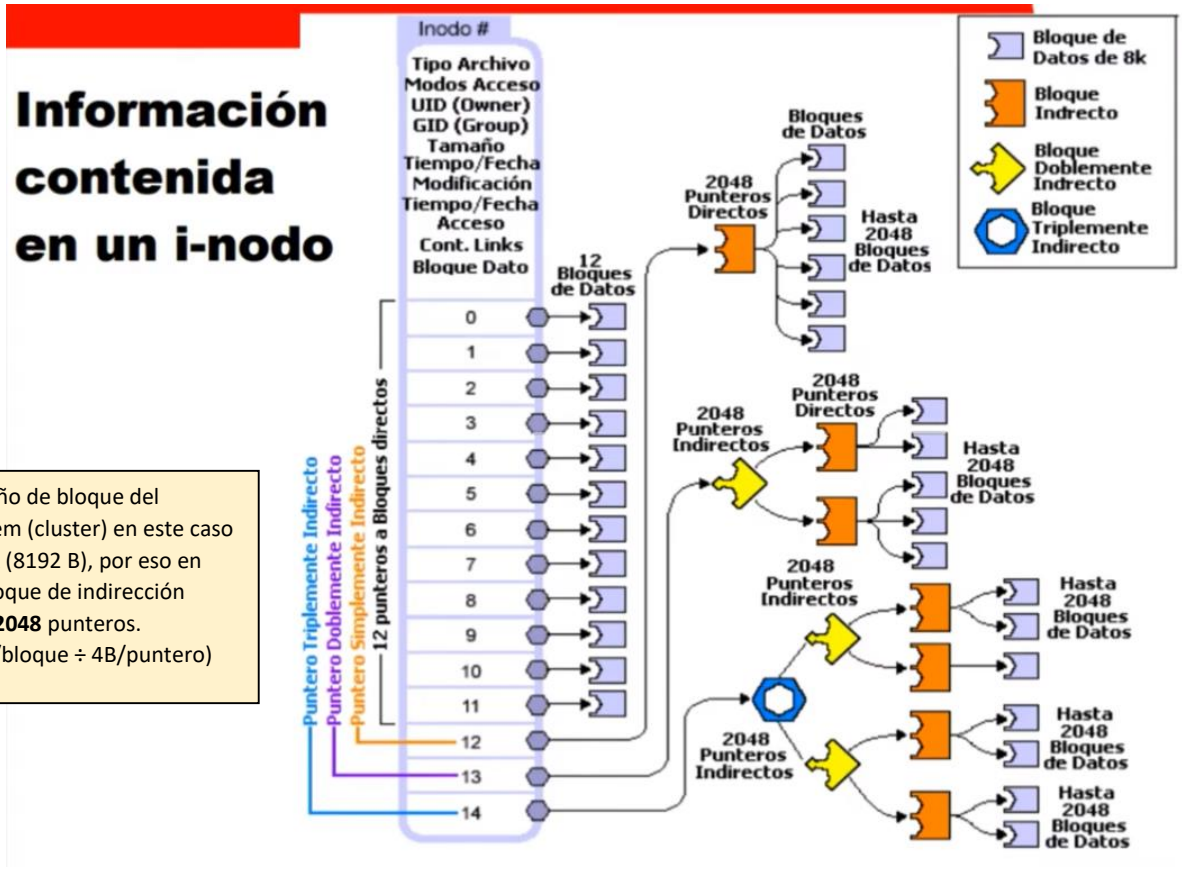
Por lo dicho anteriormente, también se puede determinar por ejemplo cuántas entradas tendrá la tabla de inodos, ya que el bitmap (secuencia de bits que determinan ocupado/libre) de inodos debe entrar en un bloque, y si el tamaño es de por ejemplo 4K (4096 Bytes), entonces habrá 32.768 inodos. Y cada inodo (por definición) ocupa 128 Bytes (**), y en este ejemplo en un bloque entrarían 32 inodos. Entonces, la tabla de inodos ocupará 32.768/32 = **1024 bloques**, donde cada bloque tendrá 32 entradas y en total son 32.768. (Nota: *esto sería para un solo 'block group', pero es lo mismo para los demás, y así podríamos calcular los números totales de todo el filesystem*).

i-nodos

La forma de hacer referencia a los archivos y directorios del filesystem, es mediante los '**inodos**'. El filesystem tiene tablas de inodos donde justamente cada entrada/registro es un inodo.
Un inodo es una estructura que contiene por un lado **metadatos** y por otro, **punteros** a los bloques de datos del archivo o directorio. Tiene 12 punteros directos a los bloques de datos y 3 punteros indirectos, uno simple (apunta a un bloque que tiene 'n' punteros directos), uno doble (apunta a un bloque que tiene 'n' punteros indirectos simples), y uno triple (apunta a un bloque que tiene 'n' punteros indirectos dobles).

El 'n' queda determinado por el tamaño del bloque dividido el tamaño de un tipo puntero. Por ejemplo, si los bloques son de 4k (4096 B), y los punteros ocupan 4 Bytes -> n = 1.024 punteros. (¿Cuál es el verdadero tamaño de los punteros en ext2/3? → **4 Bytes**).

PREGUNTAR SI ESTO ES ASI (SI LOS PUNTEROS INDIRECTOS APUNTAN A BLOQUES COMO TAL RELLENOS DE SOLO PUNTEROS, O APUNTAN A ESTRUCTURAS CON SOLO 12 PUNTEROS). **Sí, es así. Los punteros indirectos apuntan a bloques "de indireccion", es decir, bloques que únicamente tienen punteros de 4 Bytes en su interior, tantos como entren**.



El tamaño de bloque del filesystem (cluster) en este caso sería **8K** (8192 B), por eso en cada bloque de indirección entran **2048** punteros. (8192B/bloque ÷ 4B/puntero)

https://www.youtube.com/watch?v=izsYILDYW-A

Esta estructura de inodos, permite que podamos posicionarnos en cualquier bloque de datos de un archivo del filesystem en a lo sumo **4 accesos directos** (3 indirecciones y la lectura del bloque; *inodo_entry_pointer*[14] → *indireccion_3*[x] → *indireccion_2*[y] → read(*indireccion_1*[z])).

Aunque realmente los bloques de indirección creo que habría que recorrerlos secuencialmente.

Nota: Cada estructura de inodo se asocia con un **único** archivo/directorio; no puede contener metadatos o punteros referidos a más de 1 archivo. Pero varios archivos pueden tener una referencia al mismo inodo (pudiendose así tener 'accesos directos': **hards links [**softs no sé si se pueden → **Sí, también** (y también se conocen como *'symbolic links'*)**]** ***ver imagen al final***).

Esto es así porque dentro del filesystem tendremos una determinada cantidad de inodos, cada uno con un **identificador** único, y cada archivo tendrá como metadato el identificador del inodo que lo representa. Además, como metadato también tiene su propio nombre de archivo; no está dentro de los metadatos del inodo (EMMM, ESTOS 2 METADATOS QuE MENCIONÉ CREO QuE SE GUARDAN MAS BIEN EN EL BLOQUE DE DATOS ASOCIADO AL DIRECTORIO QUE 'CONTENGA' AL ARCHIVO. **En efecto, es así**. Los bloques de datos de los archivos, tienen solo datos. El bloque de datos del directorio que "contiene" a un archivo/directorio, es el que tiene los metadatos: nombre y n° de inodo. Estos bloques de directorio, en sí, contienen '**directory entries**').

Directorios

Los bloques de datos de los directorios (recordemos que también son archivos), en vez de contener datos como tal, contienen metadatos sobre los archivos o directorios que hace referencia. Por ejemplo, tienen por cada entrada (cada 'registro' asociado a un archivo) el identificador del inodo que hace referencia al archivo (son como los 'directory entry' de FAT32, pero con otros atributos/campos y además **son de tamaño variable**).
Recordemos además, que cada directorio está asociado a un inodo, por lo que estos bloques de (meta)datos de los directorios, son referenciados justamente por un inodo.

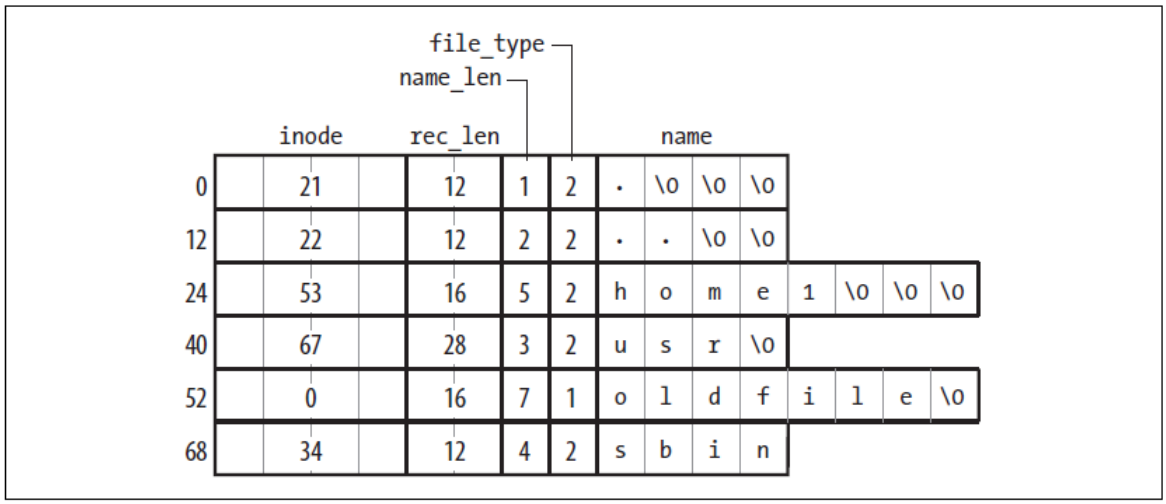
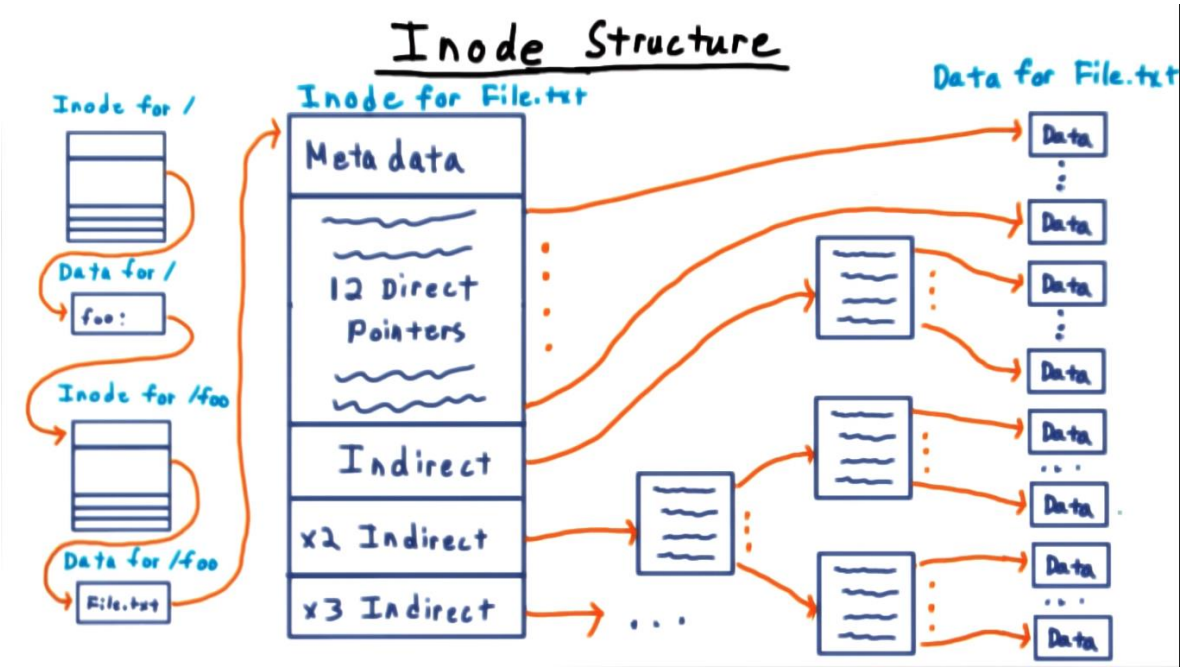


Figure 17-2. An example of the EXT2 directory

Nota: lo que no se bien es cómo hace el mapeo del número de inodo, porque si por ejemplo se pasa del límite de la tabla de inodos del 'block group' actual (todos los inodos están ocupados y se crea un archivo dentro de un directorio del 'block group' actual), lo que significa que pertenece a otro 'block group', ¿cómo hace? Quizás se lea el siguiente 'block group' y se resta el número del inodo con la longitud de la tabla de inodos anterior. **Sí, es algo así. Los números de inodos son absolutos (a todo el filesystem), y no relativos al block group actual**. (lo que si tengo que revisar es si por ejemplo el inodo 2 corresponde a la entrada 2 de la tabla de inodos o a la 3, ya que supuestamente el inodo 0 es usado para indicar 'null', pero no se si hay o no un inodo en la tabla[0] reservado por esta razón) -> **los inodos empiezan a contarse desde 1, no hay un inodo 0 como tal, ese número está reservado justamente para indicar 'inodo nulo'. Es decir, el inodo de la tabla[0] es el inodo 1, el de la tabla[1] es el inodo 2, y asi…**

Ejemplo de un directorio que contiene (*hace referencia a*) un archivo:



https://www.youtube.com/watch?v=tMVj22EWg6A

Nota: *el directorio '**/**' es el directorio **raíz** (así se representa en los filesystems ext).*

A grandes rasgos podríamos decir que: *(→: tiene/hace_referencia_a ó "linkea" a)*

Filesystem ext2/3 → inodos de directorios → nombres de archivos → inodos de archivos → datos

El filesystem tiene referencias de inodos de archivos e inodos de directorios. Los inodos de archivos tienen la referencia a los datos en sí. Los inodos de directorios tienen la referencia de nombres de archivos o directorios, y estos al inodo que los representa. Para acceder a los datos de un archivo, nos vamos moviendo por los inodos de los directorios que lo contienen (empezando por el directorio raíz), hasta llegar al inodo del archivo que tiene las referencias a los bloques de datos (este recorrido sería el 'path', que se va formando gracias a los 2 directorios 'ocultos' que contiene cada directorio: uno con la referencia del inodo actual '**.**' y otro con la del padre '**..**').

Extras

1. El '**journaling**' es un log/bitácora de las escrituras/write's que están planeadas hacerse en archivos; ya que a veces, realmente se escribe en disco cuando desmontamos correctamente una partición (*mientras tanto se va 'escribiendo' en un buffer en memoria, y cada tanto se vuelca eso al disco; las estructuras del filesystem sí quizás solo se actualicen al desmontar la partición, pero depende bastante también de cómo gestione todo esto el sistema operativo*), y si antes de eso, o justo durante el proceso de escritura, por ejemplo se nos apaga la pc **abruptamente** por 'x' motivo, luego se checkea el journaling para ver qué escrituras llegaron a hacerse, y cuáles no (en vez de tener que revisar todo el disco en busca de fallos). Entre las que no, en general podemos elegir si continuar con la escritura, o cancelarla.

2. **Estructura en general**: realmente el primer superbloque podría estar junto con el área de boot, en el mismo bloque, luego las copias sí están en bloques individuales.

   "For the curious, **block 0** always points to the first sector of the disk or partition and will always contain the boot record if one is present."

   "The superblock is always located at byte offset 1024 from the start of the disk or partition. In a 1KiB block-size formatted file system, this is **block 1**, but it will always be **block 0** (at 1024 bytes within block 0) in larger block size file systems."

   "The block group descriptor table starts on the first block following the superblock. This would be the third block (**block 2**) on a 1KiB block file system, or the second block (**block 1**) for 2KiB and larger block file systems. Shadow copies of the block group descriptor table are also stored with every copy of the superblock. Depending on how many block groups are defined, this table can require multiple blocks of storage. Always refer to the superblock in case of doubt."

   And here's the organisation of a 20MB ext2 file system, using 1KiB blocks:

   Table 3.2. Sample 20mb Partition Layout

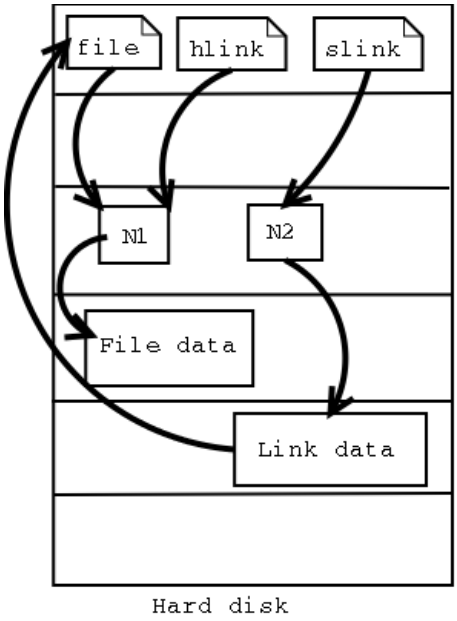
Block Offset	Length	Description
byte 0	512 bytes	boot record (if present)
byte 512	512 bytes	additional boot record data (if present)
-- block group 0, blocks 1 to 8192 --		
byte 1024	1024 bytes	superblock
block 2	1 block	block group descriptor table
block 3	1 block	block bitmap
block 4	1 block	inode bitmap
block 5	214 blocks	inode table
block 219	7974 blocks	data blocks
-- block group 1, blocks 8193 to 16384 --		
block 8193	1 block	superblock backup
block 8194	1 block	block group descriptor table backup
block 8195	1 block	block bitmap
block 8196	1 block	inode bitmap
block 8197	214 blocks	inode table
block 8408	7974 blocks	data blocks
-- block group 2, blocks 16385 to 24576 --		
block 16385	1 block	block bitmap
block 16386	1 block	inode bitmap
block 16387	214 blocks	inode table
block 16601	3879 blocks	data blocks

"Notice how block 0 is not part of the block group 0 in 1KiB block size file systems. The reason for this is block group 0 always starts with the block containing the superblock. Hence, on 1KiB block systems, block group 0 starts at block 1, but on larger block sizes it starts on block 0. For more information, see the s_first_data_block superblock entry."

"Nevertheless, unless the image was crafted with controlled parameters, the position of the various structures on disk (except the superblock) should never be assumed. Always load the superblock first."

https://www.nongnu.org/ext2-doc/ext2.html

3. Estructura de los ***hard links*** y ***soft links***.



https://tldp.org/LDP/intro-linux/html/sect_03_03.html

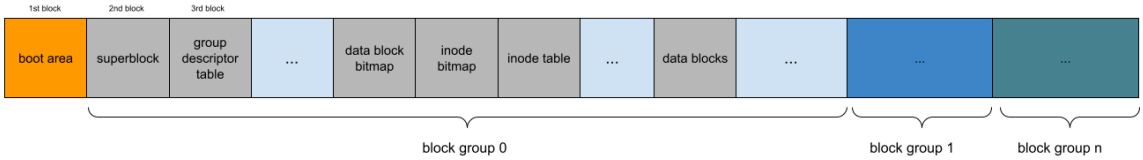
4. **Tamaño de un inodo** (**): realmente la estructura de un inodo no es siempre de 128 bytes, sino que depende de la *revisión* de Ext2 que tengamos (*esto nos lo informa el superbloque*). Si la revisión es 0, entonces sí el tamaño será de 128 bytes, pero si es la revisión 1 en adelante, puede variar (debe ser una potencia perfecta de 2 y debe ser menor o igual al tamaño del bloque). Pero sí está definido que el tamaño **base** es **128 bytes**, así que mínimamente habrá datos en los primeros 128 bytes de un inodo (*el tamaño exacto del inodo también nos lo informa el superbloque*).

5. **Layout** de la **organización** en disco de una partición con **ext2** (*hecho por mi*).

ext2 Filesystem Layout

boot area = the first two sectors of the partition (*512 * 2 = 1024 bytes*).
superblock = 1024 bytes (*there is a copy in most groups*).
group descriptor table = 32-byte entries (*there is a copy in most groups*).
both bitmaps = one entirely block each.
inode table = 128-byte entries (*in case the revision of ext2 is 0; otherwise, they may be 128+ bytes*).
data blocks = variable-size entries (*8 bytes + filename*) in case of being assigned to directories, or free bytes (*no structure*) in case of being assigned to regular files.

block size = **1K** (1024 bytes)



block size = **2K** (2048 bytes) -> the first block is full.
OR
block size = **4K** (4096 bytes) -> in the first block there are 2048 bytes left.