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'**. Todo el filesystem se estructura en bloques, conformándose **grupos de bloques**. Por ejemplo, se asigna un bloque 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 en alguno de ellos). El tamaño del bloque se define al momento de darle formato a la partición; y este queda guardado en el superbloque. Todos los grupos de bloques tendrán la misma cantidad de bloques, salvo el último que puede tener menos.

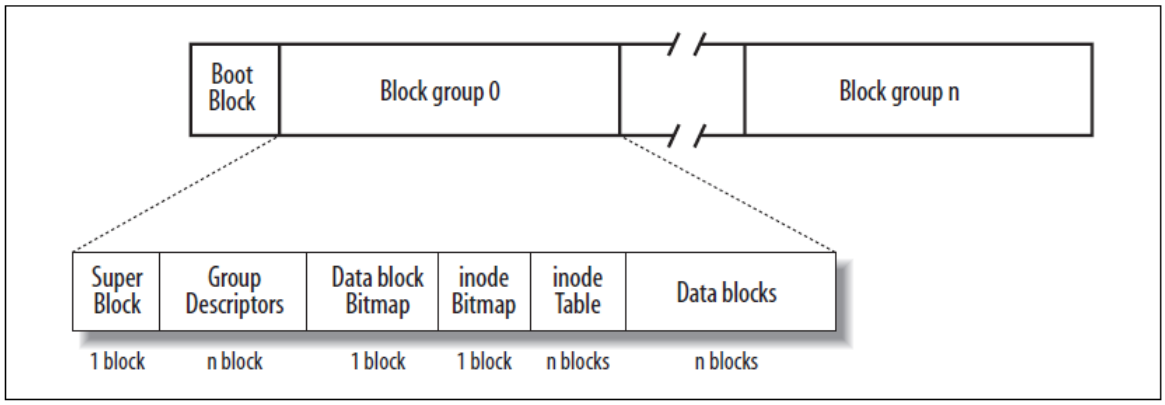


Figure 17-1. Layouts of an Ext2 partition and of an Ext2 block group

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 ó libre) de inodos debe entrar en un bloque. Por ejemplo, si el tamaño de bloque es de 4K, entonces habrá 4096 Bytes * 8 bits = 32.768 inodos. Por otro lado, cada inodo (por definición) ocupa 128 Bytes (pero puede llegar a ocupar más, **ver aclaración al final*), por lo que en este ejemplo en un bloque entrarían 4096/128 = 32 inodos. Entonces, la tabla de inodos ocupará 32.768/32 = **1024 bloques**, donde cada bloque tendrá 32 entradas de inodo. [*todo 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*].

(de todas formas, quien determina cuántos inodos habrá en la tabla de inodos de cada grupo de bloques, es el **superbloque**; y en general ese valor es menor que la cantidad máxima de inodos que se podría 'mapear', por lo que en el bitmap de inodos de cada grupo, de por sí se rellenará de *1*s todos esos bits asociados a inodos que nunca existirán en la tabla de inodos).
(y no es que esos inodos estarán en algún lado reservados para el '*superusuario*' o algo así, directamente no existen; lo que sí hay reservado para el '*superusuario*' son bloques, pero que no están en ningún lugar en 'especial' del filesystem, sino que en caso de que la partición esté a punto de llenarse, ext2 le dará acceso a los bloques restantes solo al '*superuser*').

Nota: Cada vez que se hace referencia a un bloque del filesystem, se hace de manera absoluta. Todo bloque tiene un n° de bloque asociado (*comenzando el conteo desde el bloque 0*).
Por ejemplo, si un descriptor de grupo dice que el bloque del bitmap de inodos es el '12345', será el bloque '12345' del filesystem en su totalidad, y no el bloque '12345' relativo al grupo de bloques.

i-nodos

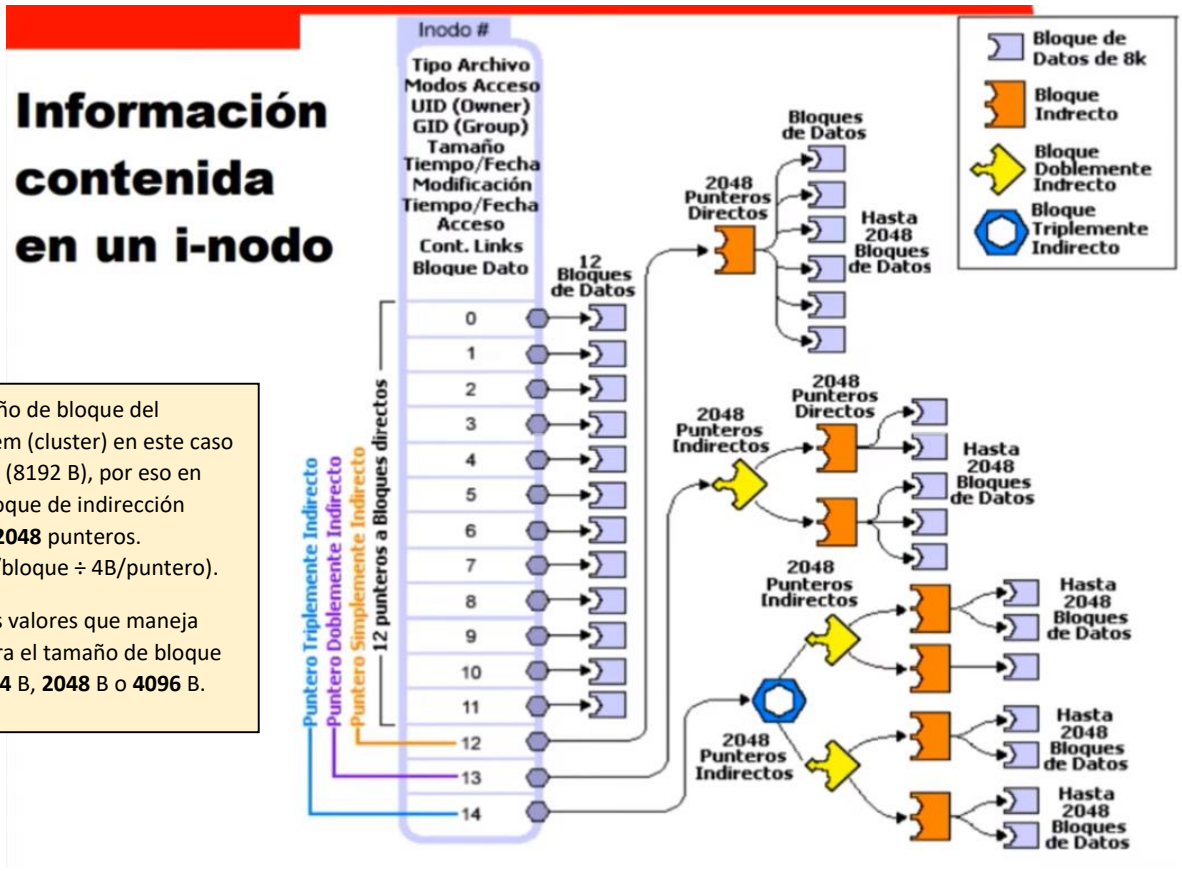
La forma de hacer referencia a los archivos y directorios del filesystem, es mediante los '**inodos**'.
El filesystem tiene tablas de inodos donde justamente cada entrada/registro es un inodo.
Un inodo es una estructura que contiene por un lado **metadatos** (tamaño, fecha de modificación..) y por otro, **punteros** a los bloques de datos del archivo o directorio (*los campos de estos punteros almacenan un n° de bloque del filesystem*). 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? → en efecto, **4 Bytes**).

Revisar 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 indirección", es decir, bloques que únicamente tienen punteros de 4 Bytes en su interior, tantos como entren**.

Nota1: cuando digo que los inodos apuntan a bloques 'de datos', es porque justamente los inodos representan a archivos (archivos regulares, directorios, etc), y los bloques asignados a ellos por ext2, se consideran bloques 'de datos', ya que son bloques pertenecientes al área de datos de algún grupo de bloques. Pero más allá de que sean punteros a bloques 'de datos', cualquier puntero a los bloques del filesystem tendrá un tamaño de 4 bytes (como el puntero al primer bloque útil del fs en el superbloque, o los punteros a los bloques de los bitmaps en los descriptores de grupo). Por ende, en el filesystem podremos tener como mucho 2^32 bloques (*del bloque 0 al bloque 2^32 - 1*), que es lo máximo que puede referenciar un puntero de 4 bytes, independientemente del tamaño de bloque (*igual el sistema operativo impone algunas limitaciones en cuanto a tamaños: https://www.nongnu.org/ext2-doc/ext2.html#def-blocks*).

Nota2: todo inodo en el filesystem, al igual que los bloques, se identifica con un número (*no puede haber 2 inodos con el mismo 'id', o sea cuando ext2 crea un inodo, lo empezará a identificar con un número que esté 'libre', sino al escribirlo en disco, se pisarían los datos de un inodo que se estaba usando*); y toda referencia a un inodo es de manera 'global', o sea, si por ejemplo decimos: 'el inodo 12345 representa al archivo notas.txt', estamos hablando del inodo '12345' del filesystem en su totalidad, y no del inodo '12345' de un determinado grupo de bloques (ese mapeo "n° de inodo global – n° de inodo dentro de un grupo de bloques" posteriormente se hará).



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).

Igual los valores que maneja ext2 para el tamaño de bloque son **1024** B, **2048** B o **4096** B.

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 (estos 2 metadatos que mencioné creo que se guardan más 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 o directorio, es el que tiene los metadatos: **nombre y n° de inodo**. Estos bloques de directorio, en sí, contienen '**directory entries**').

Directorios

Los directorios (o 'carpetas') son considerados un archivo más del filesystem, pero sus bloques de datos, en vez de contener datos como tal, contienen algunos metadatos sobre los archivos o directorios que hace referencia (*digo 'algunos' porque la mayoría de metadatos están en los inodos asociados a esos archivos*). Por ejemplo, tienen por cada entrada (cada 'registro' asociado a un archivo) el **nombre** del archivo o directorio y el **identificador** del **inodo** que hace referencia al archivo. Son parecidos a los '*directory entry*' de FAT32, pero con otros atributos/campos, y además en este caso, los registros **son de tamaño variable**, debido a que el campo del nombre del archivo es variable, entre 1 y 255 bytes, que representan caracteres (*este campo no puede tener el byte 0, ya que no se permiten archivos sin nombre*).
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.

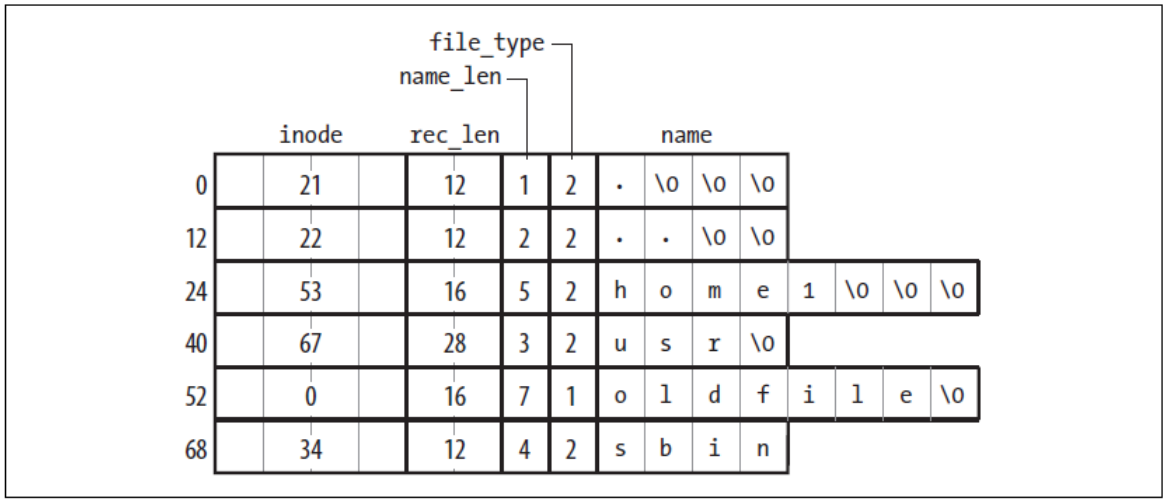
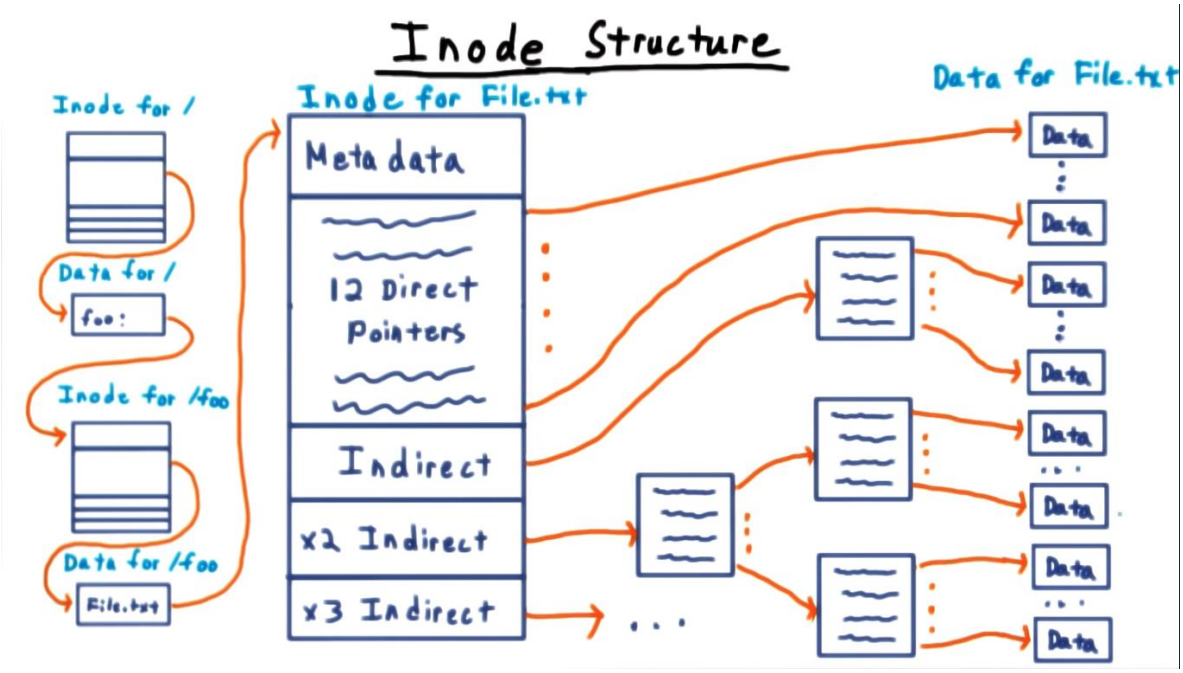


Figure 17-2. An example of the EXT2 directory

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, lo que significa que pertenece a otro 'block group', ¿cómo hace? (por ejemplo, todos los inodos están ocupados y se crea un archivo dentro de un directorio del 'block group' actual). 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** (https://www.nongnu.org/ext2-doc/ext2.html#idm140660447281728).

(lo que sí 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 sé si hay o no un inodo en la tabla[0] reservado por esta razón) -> **los inodos (*a diferencia de los bloques*) 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 así…**

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



https://www.youtube.com/watch?v=tMVj22EWq6A

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

Es decir, 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 a 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***').

Nota: cada directorio contiene 2 directorios 'ocultos': uno con la referencia del inodo actual '**.**' y otro con la del padre '**..**' (se podría decir que ambos son una especie de 'acceso directo' o 'hlink').

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.
   El *journal* como tal, es un **archivo** más del filesystem, generalmente ubicado en el medio de la partición. Al ser considerado un archivo, tendrá su respectivo inodo que lo represente, perteneciente a algún grupo de bloques. Los bloques de datos asignados al *journal*, están estructurados de una determinada manera, para que se puedan leer y escribir las 'bitacoras'.
   Por esta razón, **las estructuras de datos de ext3 son las mismas que las de ext2, y se agrega un *journal*** (*a grandes rasgos ext3 = ext2 + journal*): el *journal* al estar contenido en un archivo, no afecta a la organización en disco; el *journal* estará ubicado en el área de bloques de datos de algún grupo de bloques y listo. Lo que sí se agrega, son campos extras al superbloque, para ubicar al *journal* en el filesystem, pero no se altera su tamaño, sigue siendo de 1024 bytes ya que en ext2 se le dejaron unos cuantos bytes libres para futuras versiones (*también se agregan significados de bits, respecto al journal, en el bitmap de 'features' del superbloque y en el bitmap de 'flags' de los inodos*).

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."

"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."

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

"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. El '***bitmap de bloques de datos***' de cada grupo de bloques, no es que marque únicamente como libres u ocupados a los bloques de datos asociados a **archivos**, si no que toma en cuenta a **TODOS** los bloques que conforman al grupo de bloques en cuestión (*bloques asignados al superbloque, a la tabla de descriptores de grupo, a ambos bitmaps, a la tabla de inodos y al espacio restante para bloques de datos*). Por lo que estaría mejor decirle: '*bitmap de bloques*'.

   Los bloques de datos como tal, serían los que están asignados a los archivos de datos 'puros' (*conjunto de bytes sin estructura**), que se ubican justamente en el área de bloques de datos de cada grupo de bloques (al final del mismo). Igualmente, dentro de esta área también se ubican los bloques asignados a los directorios, por más que no sean bloques de datos como tal, sino bloques de 'metadatos' (concretamente almacenan entradas de directorio, una estructura de datos).

***** *O sea, no tienen una estructura definida por el filesystem. Son archivos que guardamos nosotros, y ya si el archivo tiene una estructura, como los XML, será problema nuestro saber cómo interpretarla cuando hagamos un read, ya que el filesystem nos devolverá los bytes en crudo. En síntesis, para ext2 los archivos regulares son un conjunto de bytes sin estructura, por ende para él no hay nada que parsear en sus bloques de datos*.

En términos generales, todo **archivo** del filesystem tendrá su contenido almacenado en bloques de datos (por más que puedan llegar a tener metadatos). Estos bloques de datos se ubican en el área de bloques de datos de algún grupo de bloques.

Y para ext2/3, hay 7 tipos de archivos: Archivos regulares (a lo que yo me refería con 'archivos de datos'), Directorios, Dispositivos de caracteres, Dispositivos de bloques, Archivos pipe (FIFO), Sockets y Enlaces simbólicos.

4. Siguiendo con el tema ***bitmap de bloques***, el conteo de bloques libres (bit 0) u ocupados (bit 1) depende exclusivamente de los bloques de datos que se vayan asignando_a o liberando_de **archivos**, o sea, una vez que se marquen como ocupados los bloques del superbloque (*1*), de la tabla de descriptores de grupo (*n*), de ambos bitmaps (*1 y 1*) y de la tabla de inodos (*n*), los bloques de datos son los que van a generar que aumente o disminuya la cantidad de bloques libres dentro de un grupo de bloques. Ya que como las estructuras tienen un tamaño fijo (en el caso de las tablas, se determinará al momento de crear el filesystem), siempre en cada grupo de bloques va a haber una cantidad fija* de bloques ocupados (por más que no haya ningún bloque de datos ocupado por algún archivo), por ende, siempre los bitmaps de bloques tendrán al principio muchos bits en 1 (*0xFFFFF…*), que nunca serán puestos en 0 (*por lo menos en situaciones normales*). O sea, no queda otra de que las próximas modificaciones que se hagan en el bitmap de bloques, sean a causa de los bloques del área de datos que fueron asignados o liberados.

   ***** Realmente es variable, porque no en todos los grupos de bloques hay una copia del superbloque y de la tabla de descriptores de grupo (*estas copias se hacen por si el primer grupo se corrompe, ya que ahí están las originales que el driver de ext2 hace referencia*).
   Por ende, **la cantidad 'base' de bloques ocupados en cada grupo**, sería:
   bloques de los bitmaps + bloques de la tabla de inodos (*estos 3 siempre sí o sí están en los grupos, y por supuesto el área de datos también, pero si no hay archivos estará toda libre*).
   A partir de esa cantidad base, va a haber algunos grupos que tengan más bloques ocupados debido al superbloque y a la tabla de descriptores de grupo (*dejando de lado que haya o no bloques ocupados por algún archivo*).
   Recordemos además, que todos los grupos de bloques tienen el mismo tamaño, o sea, la misma cantidad de bloques (*salvo el último que a veces tiene menos por no encajar perfecto al final de la partición*). Por ende, el hecho de que uno no tenga la copia del superbloque y de la tabla de descriptores de grupo, genera que su área de bloques de datos sea más grande (*los bitmaps y la tabla de inodos son de tamaño fijo, no se destinarán más bloques a ellos*), por lo que habrá más bloques para ser asignados a archivos.

5. **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 (*la revisión nos la 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** (ya que así se definió originalmente en ext2), así que sea cual sea el tamaño del inodo, en los primeros 128 bytes nos encontraremos con los campos convencionales de un inodo (*el tamaño exacto del inodo también nos lo informa el superbloque*).

   Como dato, los inodos mayores a 128 bytes se introdujeron en ext4, pero en su momento se preparó a ext2 para tenerlos en cuenta (*concretamente en la versión de Linux 1.3.98, mucho antes de que se lanzara ext4*), por si en un futuro justamente se implementaban. Igualmente, el driver de ext2 no sabrá cómo interpretar esos bytes extras, o sea, leerá del disco el tamaño de inodo que sea, pero solo sabrá manejar los primeros 128 bytes. Ext2 únicamente requiere saber el tamaño de un inodo para leerlos bien del disco.

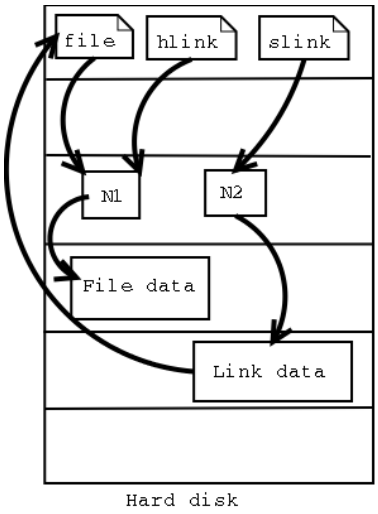
> In ext2 and ext3, the inode structure size was fixed at 128 bytes (`EXT2_GOOD_OLD_INODE_SIZE`) and each inode had a disk record size of 128 bytes. Starting with ext4, it is possible to allocate a larger on-disk inode at format time for all inodes in the filesystem to provide space beyond the end of the original ext2 inode. The on-disk inode record size is recorded in the superblock as `s_inode_size`.

```
346         __u16   s_def_resgid;           /* Default gid for reserved blocks */
347         /*
348          * These fields are for EXT2_DYNAMIC_REV superblocks only.
349          *
350          * Note: the difference between the compatible feature set and
351          * the incompatible feature set is that if there is a bit set
352          * in the incompatible feature set that the kernel doesn't
353          * know about, it should refuse to mount the filesystem.
354          *
355          * e2fsck's requirements are more strict; if it doesn't know
356          * about a feature in either the compatible or incompatible
357          * feature set, it must abort and not try to meddle with
358          * things it doesn't understand...
359          */
360         __u32   s_first_ino;            /* First non-reserved inode */
361         __u16   s_inode_size;           /* size of inode structure */
362         __u16   s_block_group_nr;       /* block group # of this superblock */
363         __u32   s_feature_compat;       /* compatible feature set */
364         __u32   s_feature_incompat;     /* incompatible feature set */
365         __u32   s_reserved[231];        /* Padding to the end of the block */
366 };
```

```
377 /*
378  * Revision levels
379  */
380 #define EXT2_GOOD_OLD_REV       0       /* The good old (original) format */
381 #define EXT2_DYNAMIC_REV        1       /* V2 format w/ dynamic inode sizes */
382
383 #define EXT2_CURRENT_REV        EXT2_GOOD_OLD_REV
384 #define EXT2_MAX_SUPP_REV       EXT2_DYNAMIC_REV
385
386 #define EXT2_GOOD_OLD_INODE_SIZE 128
```

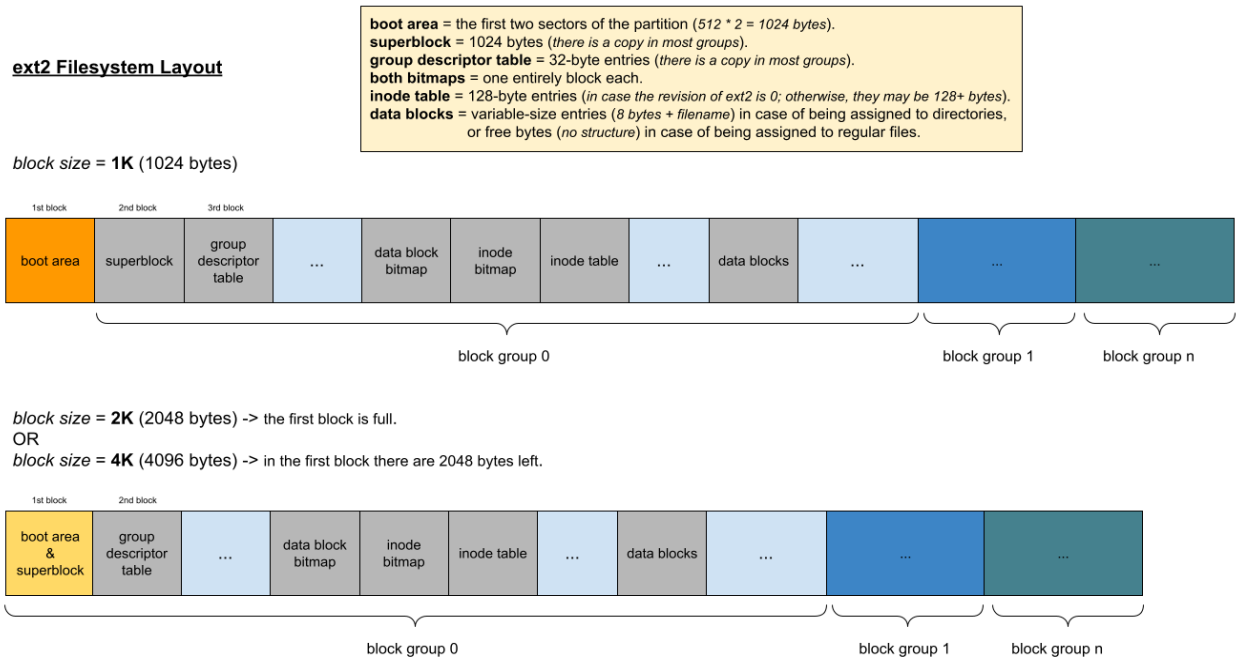
Extra: https://unix.stackexchange.com/216987/what-is-the-purpose-of-ext2-dynamic-node-size

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



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

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



8. **Familia de los Filesystem Ext**: la familia de los Ext's es nativa de Linux, y se caracteriza por estar basada en '*features*', es decir, una versión posterior toma como base la versión anterior y le agrega una serie de características nuevas, logrando así una compatibilidad entre versiones. De esta manera, el driver de ext3 sabrá cómo manejar un filesystem ext2, y el driver de ext4 sabrá cómo manejar un filesystem ext3 (*más info*).

   Historia de los Ext:
   https://opensource.com/article/18/4/ext4-filesystem
   https://opensource.com/article/17/5/introduction-ext4-filesystem

9. **Timestamps**: Los timestamps en ext2 se basan en el **Tiempo Unix**, que es la cantidad de segundos que pasaron desde la **Unix Epoch** (*01/01/1970 00:00:00 UTC+0*).

   Lo que se guarda **internamente** (*bytes escritos en el disco*) son la cantidad de segundos que pasaron desde la Unix Epoch hasta el día de hoy (o sea, hasta la fecha y hora del sistema de la computadora que provocó un cambio en algún timestamp, generalmente contemplando su *zona horaria* en vez de convertirla primero a *UTC+0* *****).

   Nota*****: *esto último depende de qué manera está seteada la fecha y hora del sistema, si contempla su zona horaria, o si está establecida en UTC+0, ya que por lo que entiendo, el timestamp (cantidad de segundos) se calcula en base a la fecha del sistema tal cual está. Info muy útil: timezone configuration utility y date and time setting not working*.

   Se manejan timestamps en los **inodos** (fechas de archivos y directorios) y en el **superbloque** (fechas generales del filesystem).

inode	superblock
i_atime	s_mtime
i_ctime	s_wtime
i_mtime	s_lastcheck
i_dtime	s_checkinterval

   Links útiles:
   https://es.wikipedia.org/wiki/Tiempo_Unix
   https://www.epochconverter.com/
   https://lwn.net/Articles/397442/

   Links extras:
   https://security.stackexchange.com/questions/33413/is-time-zone-information-saved-in-files
   https://stackoverflow.com/questions/30119732/do-filesystems-store-time-zone-information
   https://stackoverflow.com/questions/6892117/how-to-get-modified-date-in-utc

Prueba del manejo de fechas en un inodo:


```
>>>
>>> from inode import Inode
>>> inodo1 = Inode()
>>> inodo1
<inode.Inode object at 0x0000025F8A86D760>
>>> print(inodo1)
File type and access rights:          u---------
Owner identifier:                     0
File length in bytes:                 0
Time of last file access:             Not defined
Time that inode last changed:         Not defined
Time that file contents last changed: Not defined
Time of file deletion:                Not defined
Group identifier:                     0
Hard links counter:                   0
Number of data blocks of the file:    0 (in units of 512 bytes)
File flags:
Direct pointers to data blocks:       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Pointer to simple indirect block:     0
Pointer to doubly-indirect block:     0
Pointer to triply-indirect block:     0

>>> inodo1.i_atime
>>> inodo1._i_atime
datetime.datetime(1970, 1, 1, 0, 0, tzinfo=datetime.timezone.utc)
>>> print(inodo1._i_atime)
1970-01-01 00:00:00+00:00
>>>
>>> print(inodo1.i_atime)
None
>>>
>>>
>>> inodo1.i_atime = 60
>>> inodo1.i_atime
datetime.datetime(1970, 1, 1, 0, 1, tzinfo=datetime.timezone.utc)
>>> print(inodo1.i_atime)
1970-01-01 00:01:00+00:00
>>>
>>> print(inodo1)
File type and access rights:          u---------
Owner identifier:                     0
File length in bytes:                 0
Time of last file access:             1970-01-01 00:01:00+00:00
Time that inode last changed:         Not defined
Time that file contents last changed: Not defined
Time of file deletion:                Not defined
Group identifier:                     0
Hard links counter:                   0
Number of data blocks of the file:    0 (in units of 512 bytes)
File flags:
Direct pointers to data blocks:       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Pointer to simple indirect block:     0
Pointer to doubly-indirect block:     0
Pointer to triply-indirect block:     0

>>> inodo1.i_atime.timestamp()
60.0
>>> inodo1.i_ctime.timestamp()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'timestamp'
>>>
>>> inodo1._i_ctime.timestamp()
0.0
>>>
```