File systems
ext2, ext3 and ext4

Table of contents

- Ext2 file system
  - Directories
  - Data structures on disk
  - Disk space management
- Ext3 file system
  - Journaling
  - Storing directory entries in H-trees
- Ext4 file system
  - Extents
  - Large inodes
  - Nanosecond timestamps
  - Block allocation enhancements
  - Journal

What should you take into account when designing a file system!

How long does it take?
100-200 person years of effort.
5-7 of calendar time before they are enterprise ready.

File systems of the ext* family

Linux supports many file systems, but **ext* family** systems are native to it.

Ext2 (second extended file system)

- Introduced in **1993**. Main developer is **Rémy Card**.
- The maximum file size allowed is from **16 GB to 2 TB**.
- The total file system size is between **2 TB and 32 TB**.
- A directory can contain **32,000** subdirectories.
- Recommended on **flash drives** and **USB**, because it does not introduce overhead associated with journaling.
- A **journaling extension** to the ext2 has been developed. It is possible to add a journal to an existing ext2 filesystem without the need for data conversion.

General features of ext2

The **main features of ext2** affecting its **performance**:

- When creating the system, the administrator can choose the **optimal block size** (in the range of 1 KB to 4 KB), depending on the **expected average file size**.
- When creating a system, the administrator can set the **number of inodes** for a particular partition size, depending on the **number of files** expected.
- Disk blocks are divided into **groups** including adjacent tracks, thanks to which reading a file located within a single group is associated with a **short seek time**.
- The file system **preallocates** disk blocks for **regular files**, so as the file grows, blocks are already reserved for it in physically adjacent areas, which **reduces file fragmentation**.
- Thanks to the careful implementation, it is stable and flexible.
- Defined in **/fs/ext2**.

Directories in ext2

Directory – consists of records of type **ext2_dir_entry_2**.

```
#define EXT2_NAME_LEN 255

struct ext2_dir_entry_2 {
        __le32  inode;          /* Inode number */
        __le16  rec_len;        /* Directory entry length */
        __u8    name_len;       /* Name length */
        __u8    file_type;
        char    name[];         /* File name, up to EXT2_NAME_LEN */
};
```

The **file name** is limited to **255 characters** (constant **EXT2_NAME_LEN**). Depending on the setting of the **NO_TRUNCATE**, flag, longer names may be truncated or treated as incorrect.

The structure has **variable length** because the last field is an array of variable length. Each entry is supplemented with \0 to multiples of 4. The **name_len** field stores the actual length of the file name.

Directories in ext2

The **rec_len** field can be interpreted as a pointer to the next correct directory entry.

To remove an entry, just **reset** the **inode field** and **increase** the **rec_len** value.

The file is added by means of a linear search to the first structure, in which the **inode** number is **0** and there is **enough space**. If one is not found, the new file will be appended at the **end**.

User processes can **read** the directory as a file, but only the **kernel** can **write** the directory, which guarantees the correctness of the data.

Example directory (source: Bovet, Cesati, Understanding the Linux Kernel)

file_type
name_len

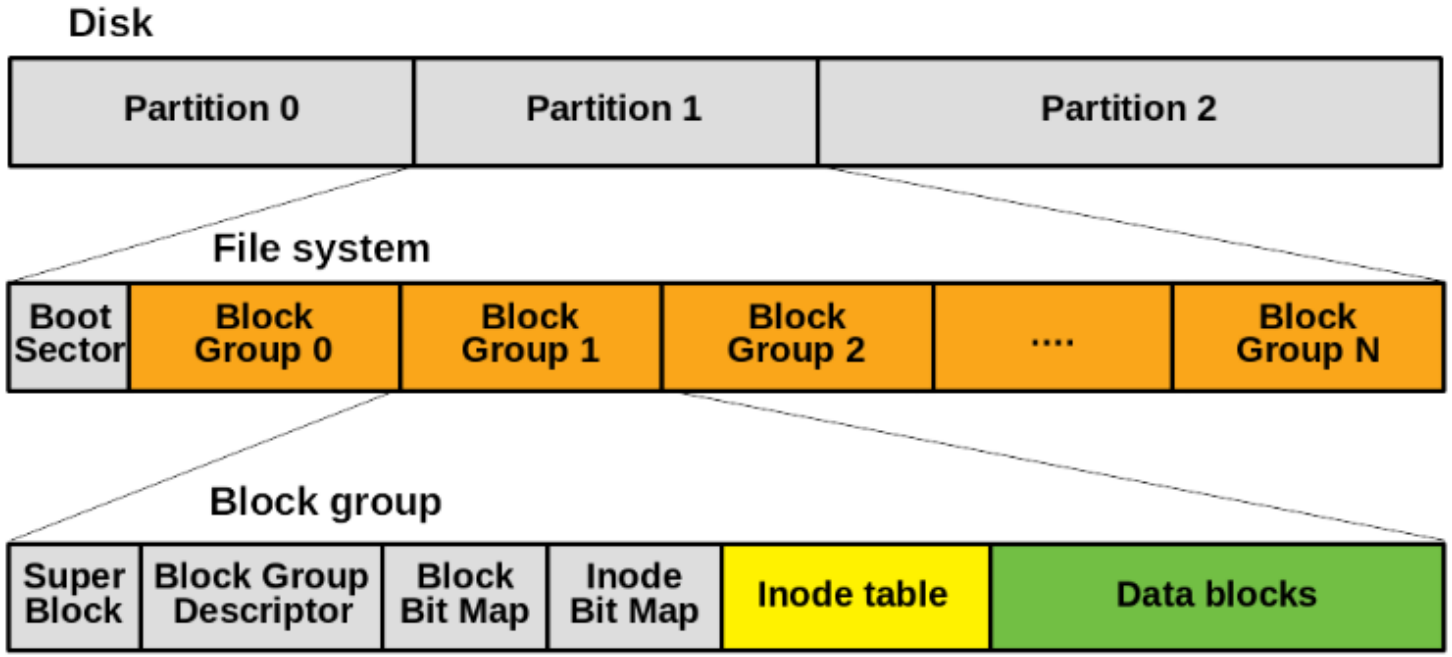
	inode	rec_len	name_len	file_type	name							
0	21	12	1	2	.	\0	\0	\0				
12	22	12	2	2	.	.	\0	\0				
24	53	16	5	2	h	o	m	e	1	\0	\0	\0
40	67	28	3	2	u	s	r	\0				
52	0	16	7	1	o	l	d	f	i	l	e	\0
68	34	12	4	2	s	b	i	n				

Ext2 file system data structures on disk

The basic **physical unit** of data on a disk is a **block**. The block size is constant throughout the entire file system. These constants limit the block size:

```
#define  EXT2_MIN_BLOCK_SIZE  1024
#define  EXT2_MAX_BLOCK_SIZE  4096   /* in bytes*/
```

Ext2 on a disk consists of many **groups of disk blocks** (of the same size, located sequentially one after the other). Block groups **increase security** and **optimize data writing to disk**.



Source: https://computing.ece.vt.edu/~changwoo/ECE-LKP-2019F/l/lec21-fs.pdf

Ext2 file system data structures on disk

Superblock — Superblocks in all groups have the same content*.

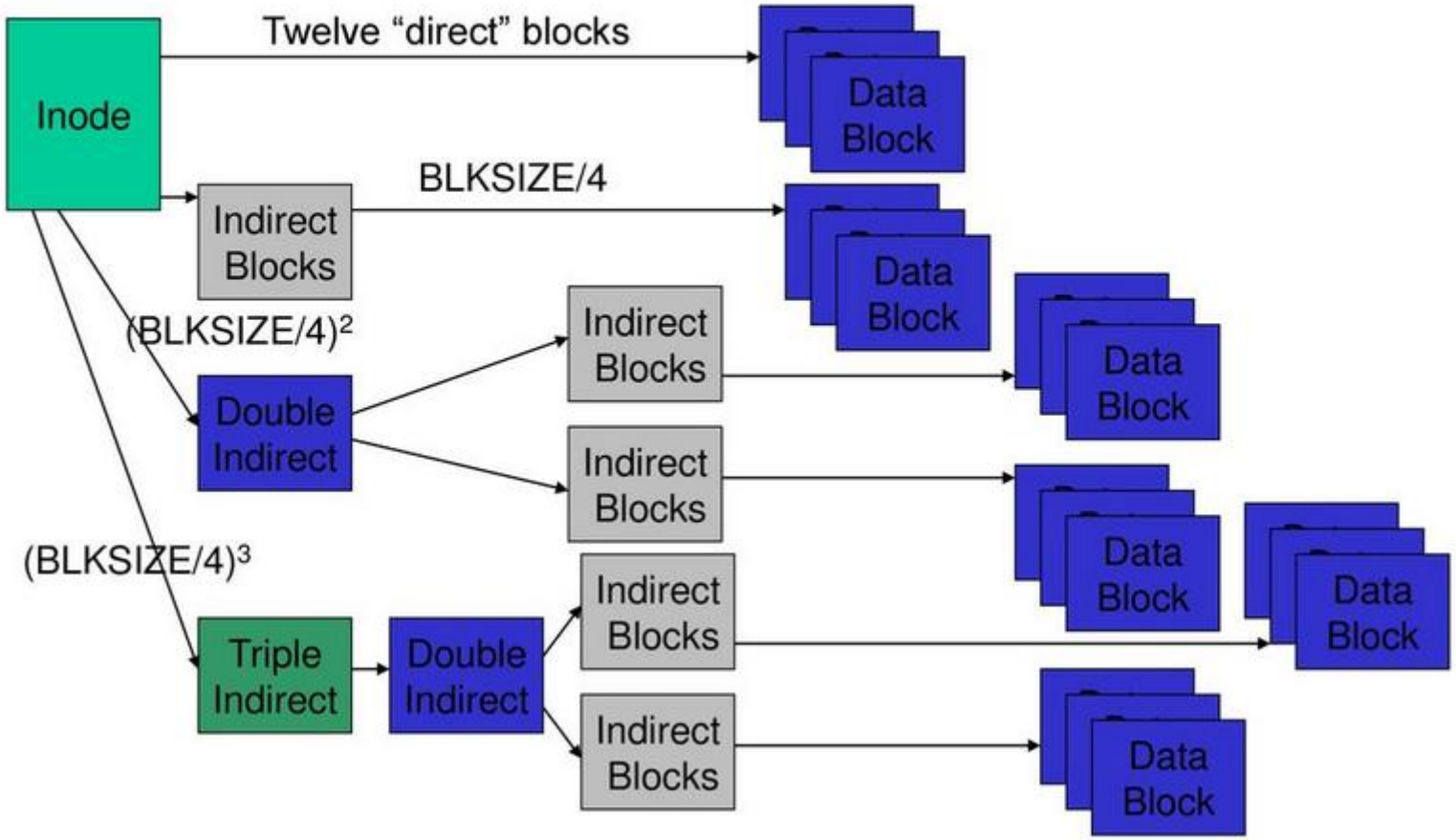
Group descriptors — As with superblocks, their content is copied to all groups*.

** Originally, the superblock and group descriptors were replicated in every block group with those located in block group 0 designated as the primary copies. This is no longer common practice due to the **Sparse SuperBlock Option**, which replicates the file system superblock and group descriptors in only a **fraction of the block groups**.*

The kernel only uses the superblock and group descriptors from **group 0**. When **e2fsck** checks the **consistency** of the file system, it reaches into the superblock and descriptors from group 0 and copies them to other groups. If as a result of the **failure** the structure data stored in block 0 are **unusable**, the administrator can order e2fsck to reach older copies in the **other groups**.

During system initialization, blocks with group descriptors from **group 0** are **read into memory**. Unless there are exceptional situations, the system does not use blocks with descriptors and a superblock from other groups

Block Addressing in Ext2



Some **block numbers** may be **zero**. This means that nothing has been saved to a certain space in the file (this is possible thanks to the **lseek ()** function).

Allocating an ext2 disk data block

The allocation of disk blocks is performed by the function **ext2_new_blocks()**.

The **inode** parameter indicates the inode for which we allocate the block, **count** indicates the desired number of blocks, **goal** gives the number of the block we would like to allocate (this is related to **pre-allocation**).

If it is not possible to allocate a block with this number, the function will try to allocate any other free block.

ext2_fsblk_t **ext2_new_blocks** (struct **inode** *inode, ext2_fsblk_t **goal**, unsigned long ***count**, int *errp)

void **ext2_free_blocks** (struct inode * **inode**, unsigned long **block**, unsigned long **count**)

If the **goal** block is free, it will be allocated. If the request cannot be completed within the current group, it tries in the others. If it still fails, **preallocation** is turned off.

Searching for a new block to be allocated first in the **immediate vicinity** of the given block makes sense for the **speed** of the file system.

The ext2 file system owes it its extremely **low file fragmentation** rate.

The **blocks** of a **given file** are almost always **close** together and loading the file is fast.

File systems of the ext* family

Ext3 (third extended file system)

- Introduced in **2001**, withdrawn in **2015**.
- The main developer is **Stephen Tweedie**.
- Available since kernel version **2.4.15**.
- The main benefit of ext3 is that it allows **journaling**. Journaling has a dedicated area in the file system, where all the changes are tracked. When the system crashes, the possibility of file system corruption is lower because of journaling.
- Maximum individual file size can be from **16 GB** to **2 TB**.
- Overall ext3 **file system size** can be from **4 TB** to **32 TB**.
- A directory can contain **32,000** subdirectories
- You can **convert ext2** file system to **ext3** file system directly (without backup/restore).

SKIP

Ext3 – journaling

The ext3 file system was first mentioned in Journaling the Linux ext2fs Filesystem (Stephen Tweedie, 1998).

If you shut down your computer without **unmounting** the ext2 file system, you must **examine the integrity** of this partition **before mounting** it again. The **larger** this partition is, the **longer** it takes, for large file systems it can take **hours**.

In the case of ext3 with the **has_journal** option enabled, the consistency test is replaced by **playing** a **journal**, which is much **faster**, in the order of **seconds**. After incorrect unmounting, playing the journal restores the correct state of the data or even the metadata.

Information about **pending file system updates** is written to the **journal**.

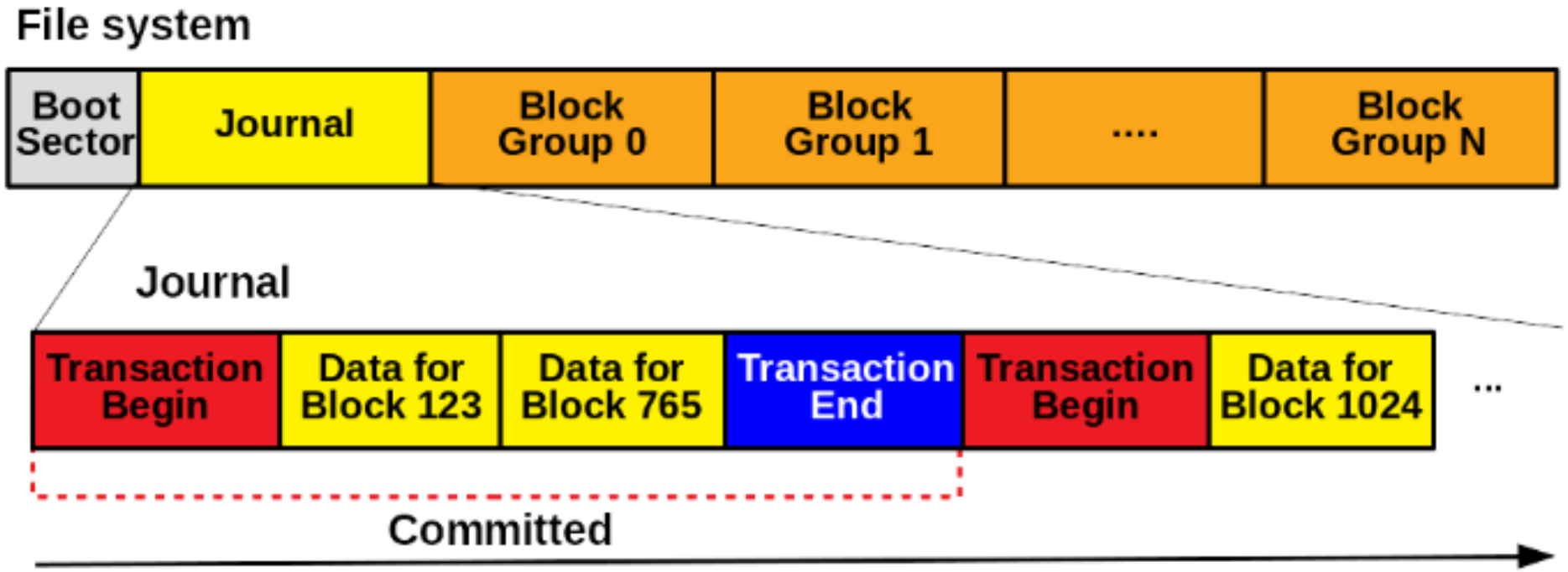
Regardless of the mode of operation, the journal ensures consistency only at the level of the **system function call**.

There are **six types of metadata** in **ext2** and **ext3**: superblocks, block group descriptors, inodes, intermediate index blocks, data block bitmaps and inode bitmaps.

Ext3 – journaling

Journal is logically a **fixed-size**, **circular array**.

- Implemented as a **special file** with a **hard-coded inode** number.
- Each journal **transaction** is composed of a **begin** marker, **log**, and **end** marker.



Source: https://computing.ece.vt.edu/~changwoo/ECE-LKP-2019F/l/lec21-fs.pdf

Ext3 – journaling

Transactions

Instead of considering each file system update as a **separate transaction**, ext3 groups many updates into a single **compound transaction** that is periodically committed to disk. Compound transactions may have better performance than more fine-grained transactions when the same structure is **frequently updated** in a short period of time (e.g., a **free space bitmap** or an **inode** of a **file** that is constantly being **extended**).

Checkpointing

It is the process of writing **journaled metadata** and **data** to their **fixed-locations**. Checkpointing is triggered when various thresholds are crossed, e.g., when file system **buffer space** is **low**, when there is **little free space left in the journal,** or when a **timer expires**.

https://www.usenix.org/legacy/publications/library/proceedings/usenix05/tech/general/full_papers/prabhakaran/prabhakaran_html/main.html

Ext3 – journaling

At some point we will wish to **commit** our outstanding **filesystem updates** to the **journal** as a new **compound transaction**.

When we **commit** a transaction, the new **updated filesystem blocks** are sitting in the **journal** but have **not yet been synced** back to their **permanent home blocks** on disk (we need to keep the old blocks unsynced in case we crash before committing the journal).

Once the journal has been **committed**, the **old version** on the disk is **no longer important** and we can **write** back the **buffers** to their **home locations** at our leisure. **Until** we have **finished syncing** those buffers, we **cannot delete** the copy of the **data** in the **journal**.

The ext3 uses **checkpoints** at which a check is made to ascertain whether the changes in the journal have been written to the filesystem. If they have, the data in the journal are **no longer needed** and can be **removed**.

During **recovery**, the file system scans the log for **committed complete** transactions; **incomplete** transactions are **discarded**. Each update in a completed transaction is simply replayed into the **fixed-place ext3 structures**.

Ext3 – journaling modes

1. **Writeback** (highest risk)

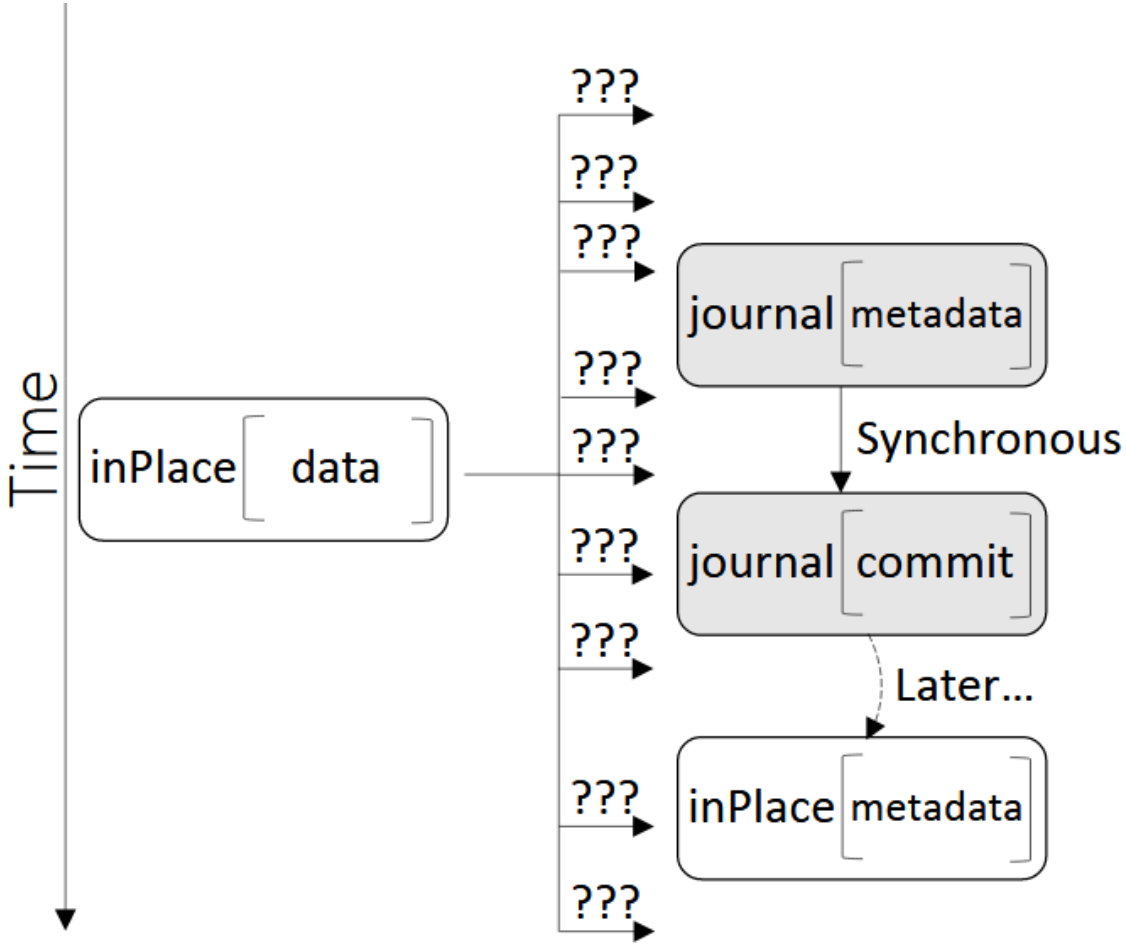
Only **metadata** is journaled (as in XFS, JFS and ReiserFS); file contents are not.

The contents might be written **before** or **after** the journal is updated (the system **does not wait** for associated changes to file data to be written before updating metadata).

Files modified right before a crash can become corrupted. For example, a **file being appended to** may be marked in the journal as **being larger** than it actually is, causing **garbage** at the **end**. **Older versions** of files could also appear unexpectedly **after** a journal **recovery**.

Usually causes the smallest overhead.

Source: https://en.wikipedia.org/wiki/Ext3



Source: https://www.eecs.harvard.edu/~cs161/notes/journaling.pdf

Ext3 – journaling modes

2. Ordered (medium risk)

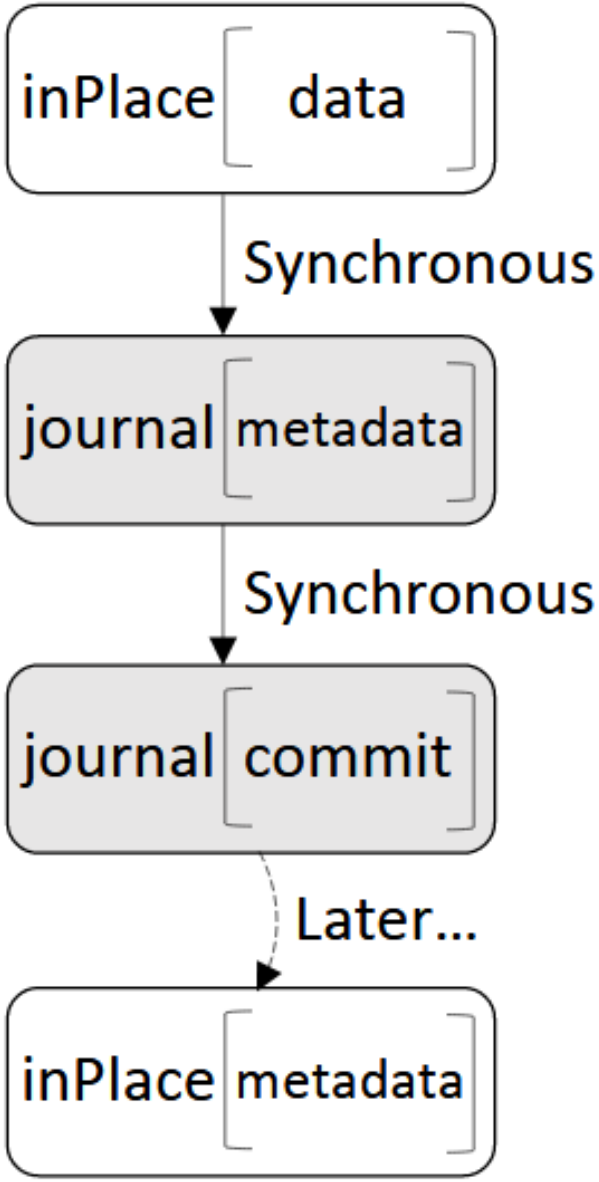
Only **metadata** is saved in the journal. **Metadata** are journaled only **after** writing **data** to disk. This is the **default** on many Linux distributions.

If there is a power outage while a file is being **written** or **appended to**, the journal will indicate that the new file or appended data has not been committed, so it will be purged by the cleanup process.

Files being **overwritten** can be corrupted because the **original** version of the file is **not stored**. It's possible to end up with a file in an intermediate state **between new** and **old**, without enough information to restore either one or the other (the new data never made it to disk completely, and the old data is not stored anywhere). Even worse, the intermediate state might intersperse old and new data, because the order of the write is left up to the disk's hardware.

This mode is generally slightly slower than writeback and much faster than journal.

Source: https://en.wikipedia.org/wiki/Ext3



Source: https://www.eecs.harvard.edu/~cs161/notes/journaling.pdf

Ext3 – journaling modes

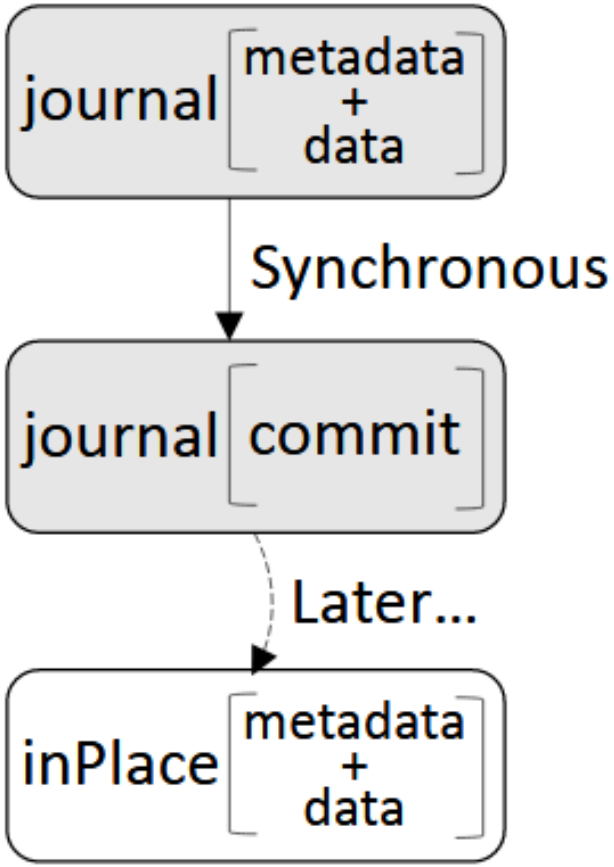
3. Journal (lowest risk)

Both **metadata** and **file contents** are written to the journal before being committed to the main file system. Data consistency is guaranteed: in case of failure, the file will contain **old data** or **new data**.

Because the journal is **relatively continuous** on disk, this can **improve performance**, if the journal has enough space. In other cases, **performance** gets **worse**, because the data must be **written twice—once** to the journal, and once to the main part of the filesystem.

In all modes, ext3 logs **full blocks**, as opposed to **differences** from old versions; thus, even a single bit change in a bitmap results in the entire bitmap block being logged.

Source: https://en.wikipedia.org/wiki/Ext3



Source: https://www.eecs.harvard.edu/~cs161/notes/journaling.pdf

Ext3 – directories in H-trees

In ext2, the directory is a **list** of variable size directory entries. Searching for the inode number takes **O (n)**, when **n** is the number of entries in the directory. The ext3 partition with the **dir_index** option enabled can reduce the search time of the inode **several times**.

H-trees (**htree**, **hashed binary tree**) used in ext3 directory indexes are trees of **height 2 or 3** with equal depth of all nodes. The **root** of an **h-tree** index is the **first block** of a **directory file**. The **leaves** are **normal ext2 directory blocks**, referenced by the root or indirectly through intermediate h-tree index blocks. References within the directory file are by means of **logical block offsets** within the file.

The nodes other than leaves, as in B-trees, contain **key values** that separate the keys in the subtrees attached to subsequent child nodes.

The keys are the hash function values for the file names.

Ext3 supports several hash functions.

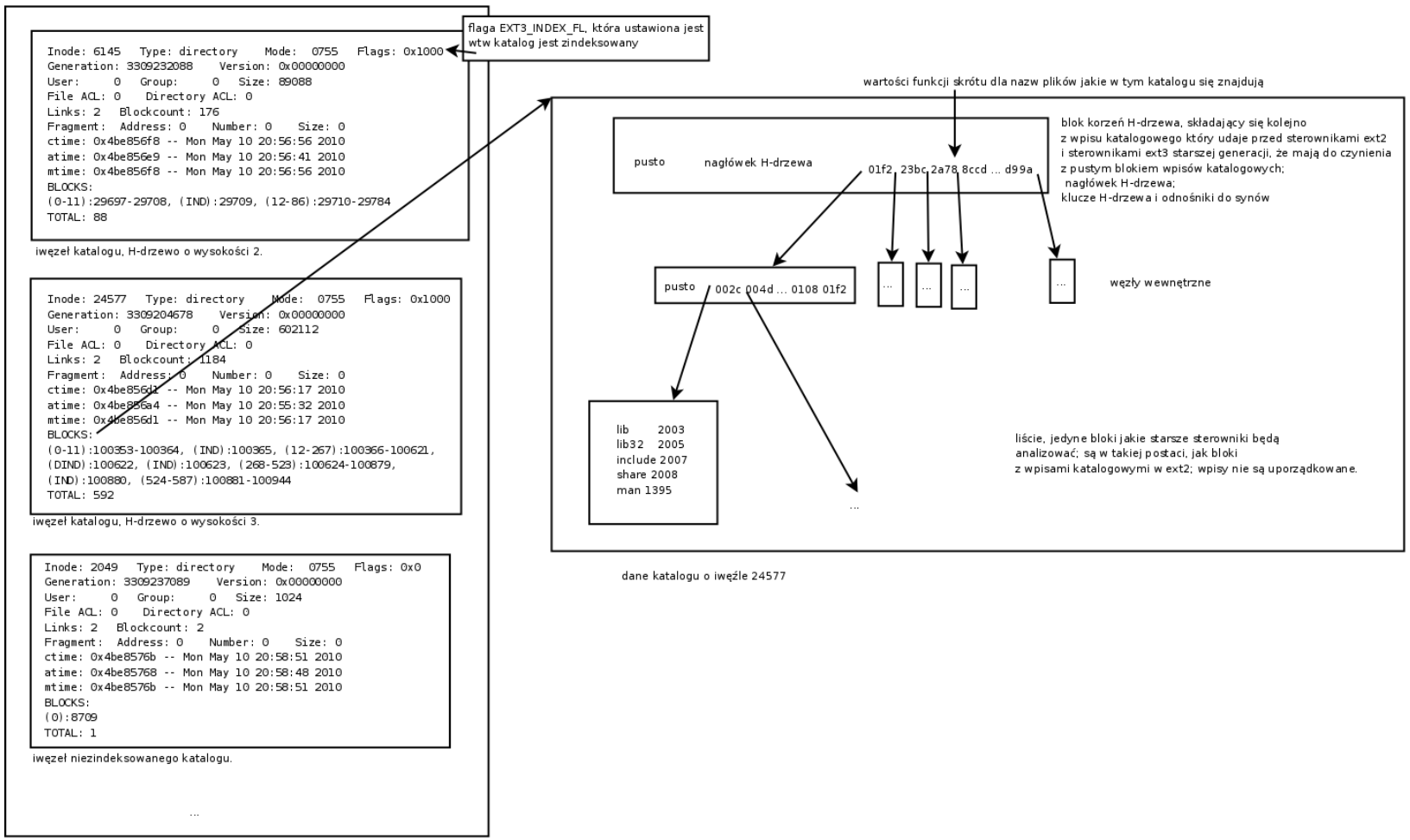
A Directory Index for Ext2, Daniel Phillips, 2001

Add ext3 indexed directory (htree) support (October 2002, ver. 2.5.40)

*This patch significantly increases the speed of using large directories in ext3, in a completely backwards and forwards compatible fashion. Creating **100,000 files** in a single directory took **38 minutes without directory indexing**... and **11 seconds with** the **directory indexing** turned on.*

```
Filesystem volume name:   <none>
Last mounted on:          <not available>
Filesystem UUID:          7973fd20-a477-4a97-9386-bdaa28b01509
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
    Filesystem features:      has_journal ext_attr resize_inode dir_index filetype needs_recovery sparse_super
    Filesystem flags:         signed_directory_hash
    Default mount options:    (none)
    ...
    Default directory hash:   half_md4
    Directory Hash Seed:      3bd9302a-1f0a-4e7e-b543-beb4c97554d8
    ...
```

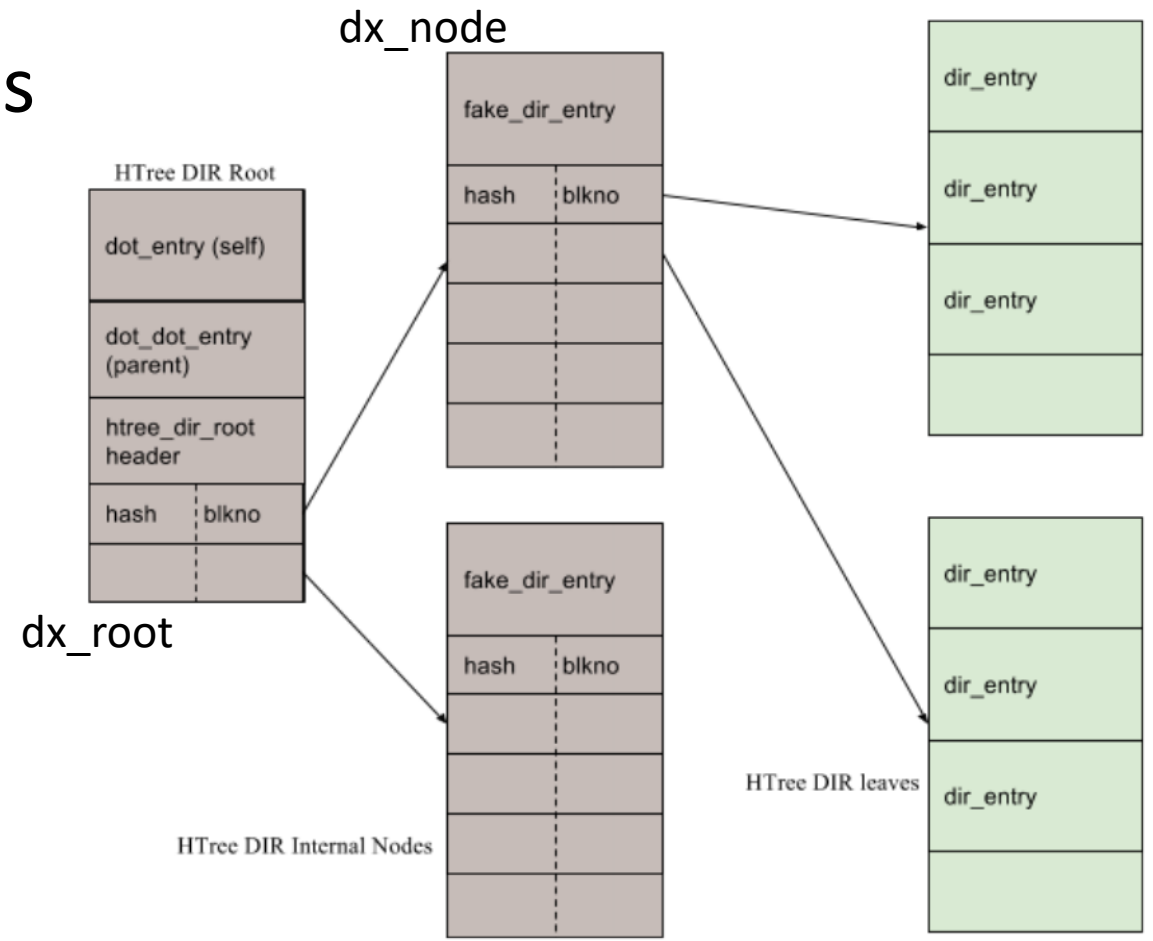
superblok partycji ext3 z włączoną cechą dir_index, o funkcji skrótu half_md4 użytej w H-drzewach i sekrecie 3bd9302a-1f0a-4e7e-b543-beb4c97554d8



SKIP

Ext3 – directories in H-trees

```
struct dx_root
{
   struct fake_dirent   dot;
   char                 dot_name[4];
   struct fake_dirent   dotdot;
   char                 dotdot_name[4];
   struct   dx_root_info
   {
      __le32   reserved_zero;
      u8       hash_version;
      u8       info_length; /* 8 */
      u8       indirect_levels;
      u8       unused_flags;
   }
   info;
   struct dx_entry  entries[0];
};
 struct dx_node
{
   struct fake_dirent   fake;
   struct dx_entry      entries[0];
};
```




```
struct dx_entry
{
   __le32    hash;
   __le32    block;
};
```

```
struct fake_dirent
{
   __le32    inode;
   __le16    rec_len;
   u8        name_len;
   u8        file_type;
};
```

SKIP

Ext3 – directories in H-trees

The **search** for a **file name** in the H-tree begins with a binary search of the leaf in which the directory entry for the name is found.

Directory entries within the **leaf** are **not ordered**, the leaf should be **searched linearly**.

There may be a **collision** of hash function values.

An important case is when of the two directory entries for conflicting names, one has the **largest hash value** in its leaf and the other has **the smallest** in the successor of this node.

If we are looking for the second name, then we get to this first block and there we recognize, that the **searched name is missing**.

At this point we need to search the **successor** node (and perhaps more nodes).

The **youngest bit** of the **hash** in the **parent node** indicates whether such a **collision** on the border occurred; thanks to this information we can skip searching the successor.

SKIP

Ext3 – directories in H-trees

Adding an entry consists in adding a new directory entry to the appropriate leaf.

If the leaf is **full** and there is only **one level** of index nodes, we perform the **split** operation as in a B-tree.

If the leaf is **full** and there are **two levels** of index nodes, then there are several tens of millions of entries in the directory, or because of the fragmentation too many other nodes are not full – in this case the **inability** to **create** the **file** is reported.

The directory entry is **deleted** only in the leaf.

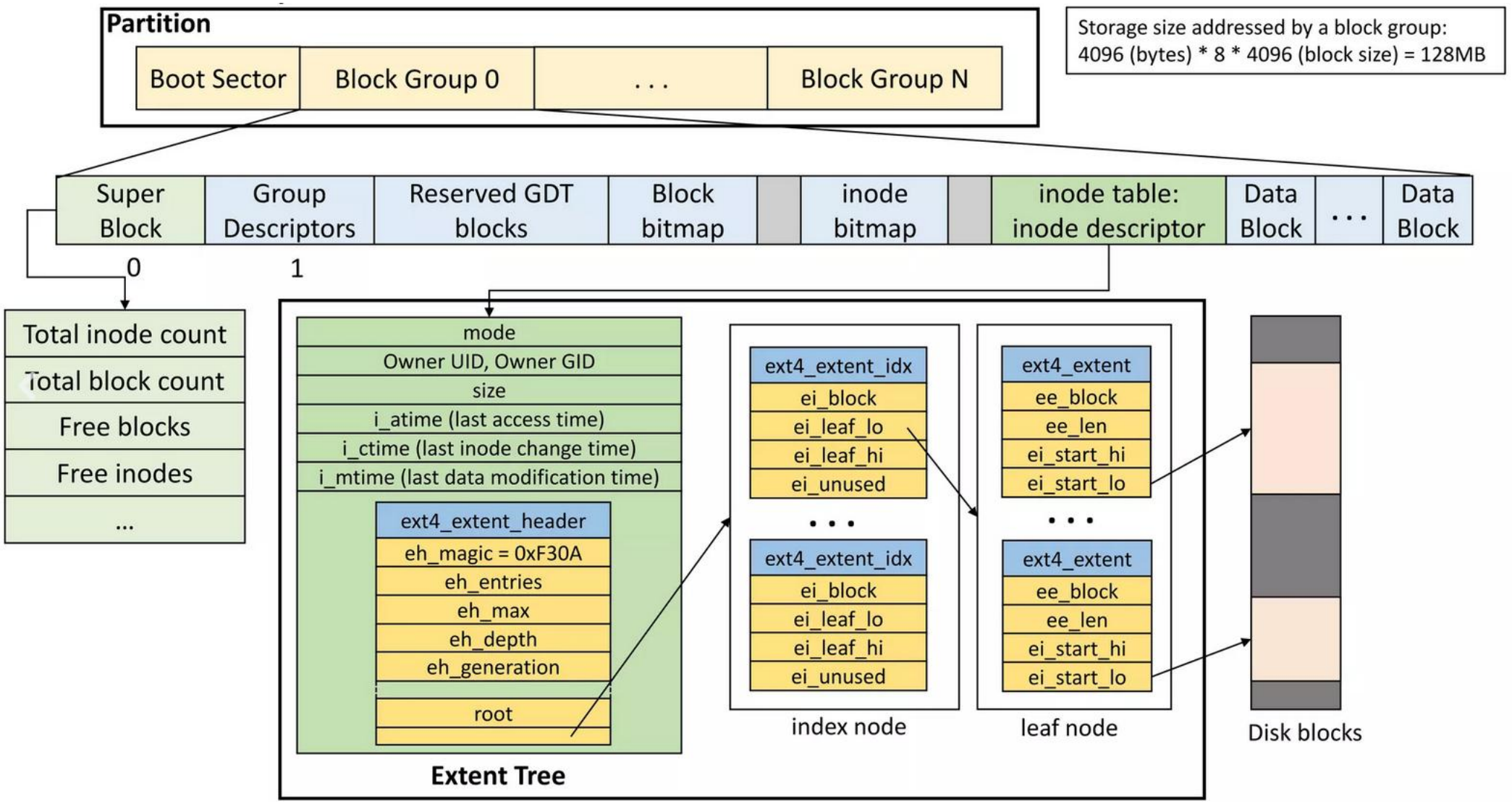
If the **leaf** becomes **empty**, we do **nothing** about it – which simplifies the implementation, but potentially prevents the operation of splitting another node

File systems of the ext* family

Ext4 (fourth extended file system)

- Introduced in **2008** (not entirely new filesystem, rather fork of ext3).
- Main maintainers: Theodore Ts'o, Andreas Dilger.
- Available since kernel version **2.6.19**.
- Supports **huge individual file size** and overall **file system size**.
- Maximum **individual file size** can be from **16 GB** to **16 TB**.
- Overall **file system size** can be from **1 EB** (exabyte).
  1 EB = 1024 PB (petabyte), 1 PB = 1024 TB (terabyte).
- A directory can contain **64,000** subdirectories.
- Several other new features are introduced in ext4: **multiple block allocation**, **delayed allocation**, **persistant preallocation**, **journal checksum**, **fast fsck**, etc.
- There is an option of **turning** the **journaling** feature **off**.
- An existing ext3 can be mounted as ext4 (without having to upgrade it).

EXT4 file system format



(source: Adrian Huang, Virtual File System, 2022)

Extent

The most important feature that distinguishes ext4 from the ext2 and ext3 is the **extents** mechanism, which replaces **indirect block addressing**.

Instead of addressing individual blocks, ext4 tries to **map** as much data as possible to a **continuous block area** on the disk. To get this ext4 mapping needs 3 values:

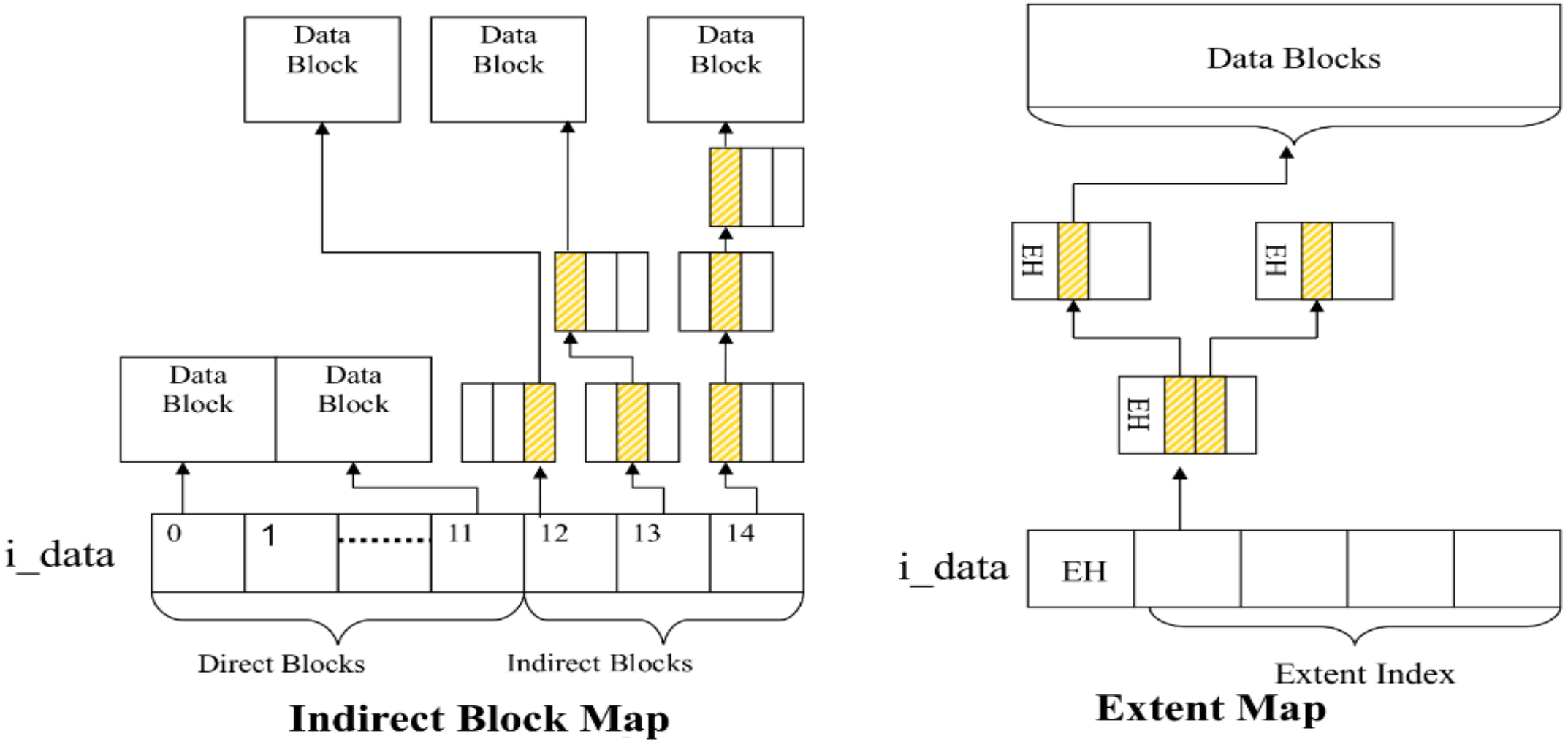
- the **initial** mapping **block** in the **file**,
- the **size** of the mapped area (in blocks) and
- the **initial block** of data saved on the **disk**.

The structure that stores these values is called **extent**.

```
#define  EXT4_MIN_BLOCK_SIZE    1024
#define  EXT4_MAX_BLOCK_SIZE    65536      /* in bytes*/

struct ext4_extent {
      __le32  ee_block;      /* first logical block extent covers */
      __le16  ee_len;        /* number of blocks covered by extent */
      __le16  ee_start_hi;   /* high 16 bits of physical block */
      __le32  ee_start_lo;   /* low 32 bits of physical block */
};
```

Indirect Block Map vs Extent map



Source: https://computing.ece.vt.edu/~changwoo/ECE-LKP-2019F/l/lec21-fs.pdf

SKIP

File, volume, extent size

File blocks in the ext4 system are **numbered** using **32 bits**, which limits their number to 2^{32} 4 KB blocks. This gives a maximum file size of $2^{32}*2^{12}=2^{4}*2^{40}$=**16 TiB**. In standard ext3, the file can have a maximum of **2 TiB**.

The volume size, in turn, is limited by the **48-bit block identifier** on the disk, which for a 4 KB block size gives $2^{48}*2^{12}=2^{60}$=**1 EiB**. For comparison, ext3 with a 32-bit number and a 4 KB block size offered a maximum partition size of **16 TiB**.

The **size of the extent** is limited by **2^{15}** blocks, i.e. for a 4 KB block it gives $2^{15}*2^{12}=2^{17}$=**128 MB**. This limitation results from the division into block groups, and a single block group can have a maximum size of **128 MB**. Due to this limitation, the last bit of the 16-bit extent size can be used in the **preallocation** mechanism.

The extents mechanism reduces the size of metadata, which means that operations on large files are much faster. The **500 MB file** in ext4 uses **four 12-byte extents**, while the block addresses of the same file need more than **0.5 MB metadata** in ext2. The advantage of the new solution can be seen especially in operations requiring many operations on metadata (e.g. file deletion).

1 exbibyte = 2^{60} bytes = 1152921504606846976 bytes = 1,024 pebibytes
1 EiB is approximately 1.15 EB, where exabyte (EB) to 10^{18} bytes.

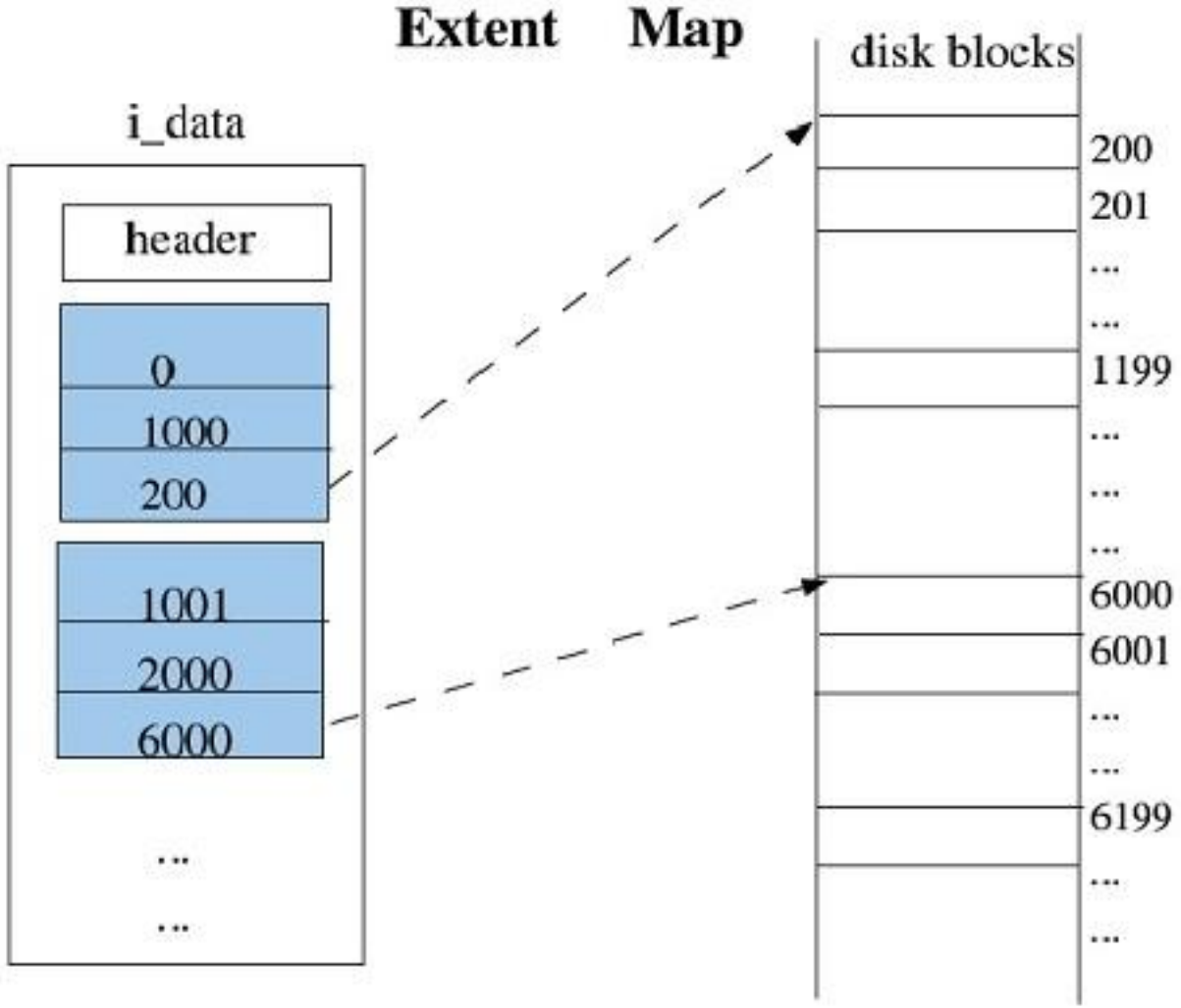
Storing files up to 512 MB

The metadata for storing information about extents is saved in the **inode** in the **60-bytes area** used in ext2 for storing **block addresses**.

There are **4 extents** in this area and **1 header** describing them (12 bytes for each structure).

struct **ext4_extent_header**

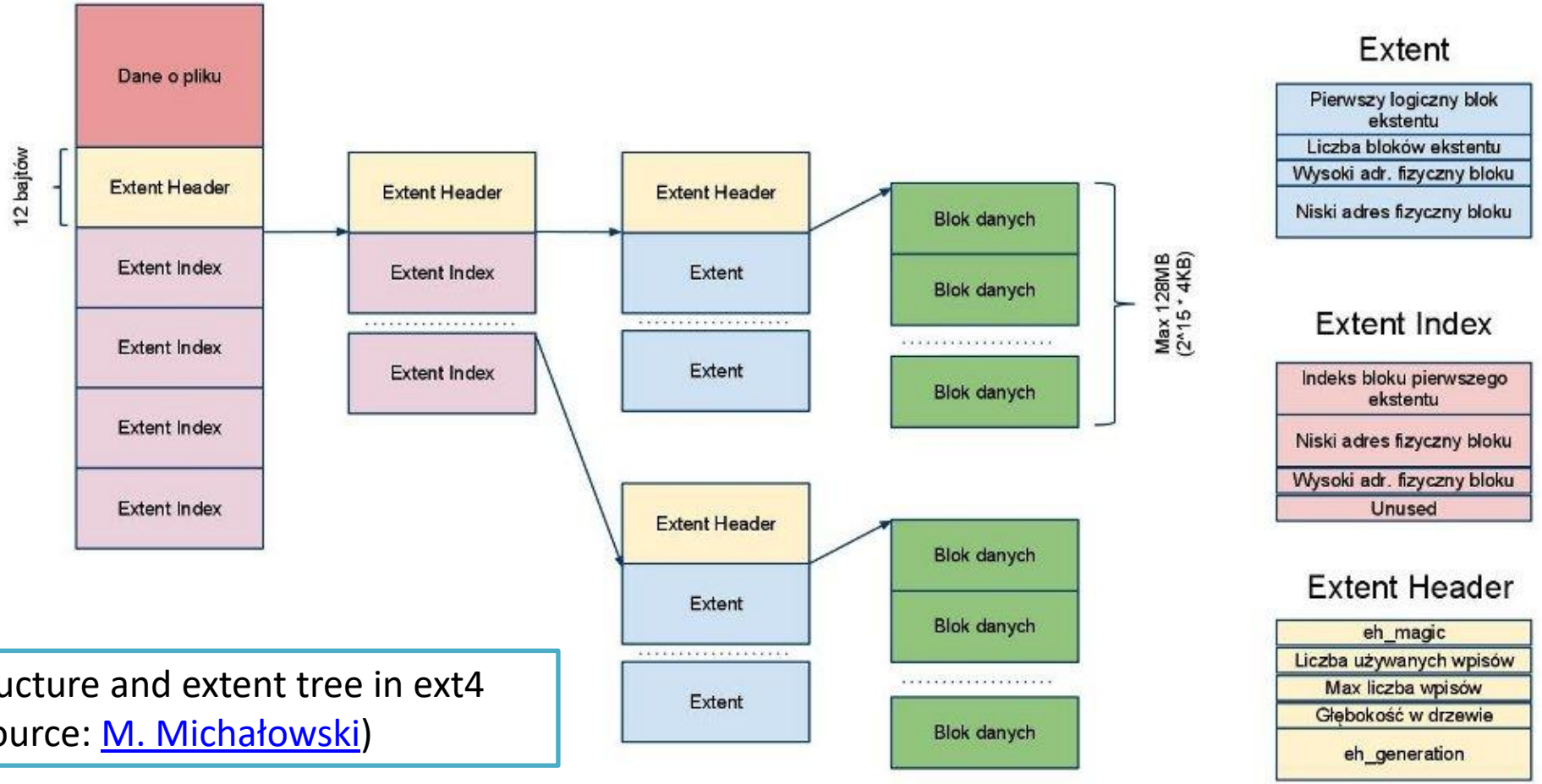
struct **ext4_extent_idx**



Extent map (source: Ext4: The Next Generation of Ext2/3 Filesystem, Mingming Cao, Suparna Bhattacharya, Ted Tso)

Storing files over 512 MB

A tree is built for larger chunks of data. For this purpose, an additional structure is used – an **index** containing the initial position of the extent in the file and the block number of the data on the disk. This block always contains a header describing the data and may contain further indexes or extents with the data.



Inode structure and extent tree in ext4
(source: M. Michałowski)

Block allocator changes

Needed to best suport extents:

- **Extents** work best if files are **contiguous**.
- **Delayed allocation** and allocating **multiple blocks** at a time makes this much more likely.
- Responsible for most of ext4's performance improvements.

Multiple block allocation

In ext2, as well as ext3, **each block** of the file had to be **allocated separately**, which in the case of large files resulted in a large number of calls to the allocation function. In addition to performance issues, this made the file system more susceptible to **fragmentation**.

Ext4 has a **multiple block allocation** mechanism (**mballoc**) that is necessary to ensure a **continuous block area** for **extents**.

Depending on the file size, the **allocator** uses **different strategies**

- for **small files** (<16 blocks) it tries to keep them close together, which will speed up their reading;
- **large files** are allocated so that they are in the **most continuous memory area possible**.

This solves the performance and fragmentation issues that occur in ext2.

Regardless of which strategy the ext4 allocator uses – it **first checks** if there are **free preallocated blocks**, only in the **next step** uses the **buddy cache**.

Description of the allocator:

Mballoc.c @ LXR – 300 line comment on the operation of the multiblock allocator.

Using this mechanism does not affect the format of the data stored on the disk.

Delayed allocation

Delayed allocation (**allocate-on-flush**) is a technique used in many modern file systems, consisting in **maximum delay** in **block allocation** (in contrast to traditional file systems, in which **blocks are allocated as soon as possible**) .

If the process writes to a file, the **file system** immediately **allocates** the **blocks** where the **data** will be **written**, even if it does not happen immediately and the data is cached for some time.

In the case of delayed allocation, **blocks** are **not allocated immediately** upon **writing**, but only when **disk writing** is **actually** to **take place**. This allows the block allocator to optimize allocation.

Delayed writing works very well with two other techniques: **extents** and **multiple block allocation**, because in many situations when the file is finally saved to disk, it will be placed in the extents allocated using the **mballoc** allocator. This improves performance and reduces fragmentation.

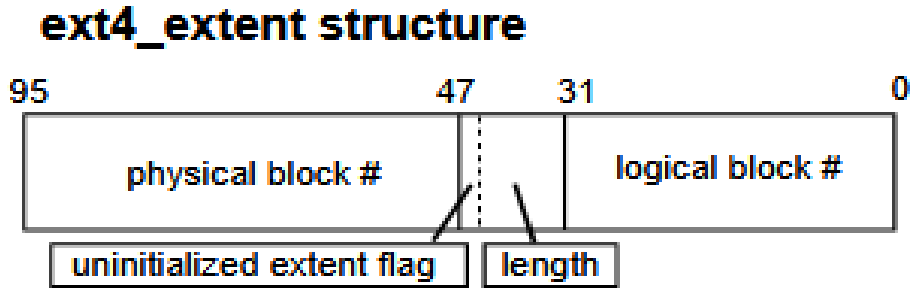
In the case of **temporary files**, there is a chance that you will not need to save them to disk at all.

Disadvantages: Increases the risk of **data loss** during a failure. Many assumptions about writing to a file, true for ext2, become false for ext4.

Persistent preallocation

Persistent preallocation allows blocks to be assigned to files without initializing first:

- Most useful for databases and video files.
- Also useful for files that **grow gradually** via small append operations (i.e. Unix mail files and log files).
- Protects against the lack of disk space for **file extension** and allows to **reduce data fragmentation**.
- The **fallocate()** system call allows to reserve a specific area for a file that does not initially use all space.
- Information that the file is **pre-allocated** and **extent** contains **uninitialized data** is contained in bit **16** of the field describing the **size** of the **extent** (**ee_len**).
- During **reads**, an uninitialized extent is treated just like a **hole**, so that the VFS returns zero-filled blocks.
- Upon writes, the extent must be split into **initialized** and **uninitialized extents**, merging the initialized portion with an adjacent initialized extent if contiguous.



LWN: fallocate()

Layout of the large inode

Ext3 supports different **inode sizes**. The inode size can be set to any power-of-two larger than **128 bytes** size up to the filesystem block size. This can be done by the **mke2fs -I [inode size]** command at format time. The default inode size is **128 bytes**, which is already crowded with data and has little space for new fields.

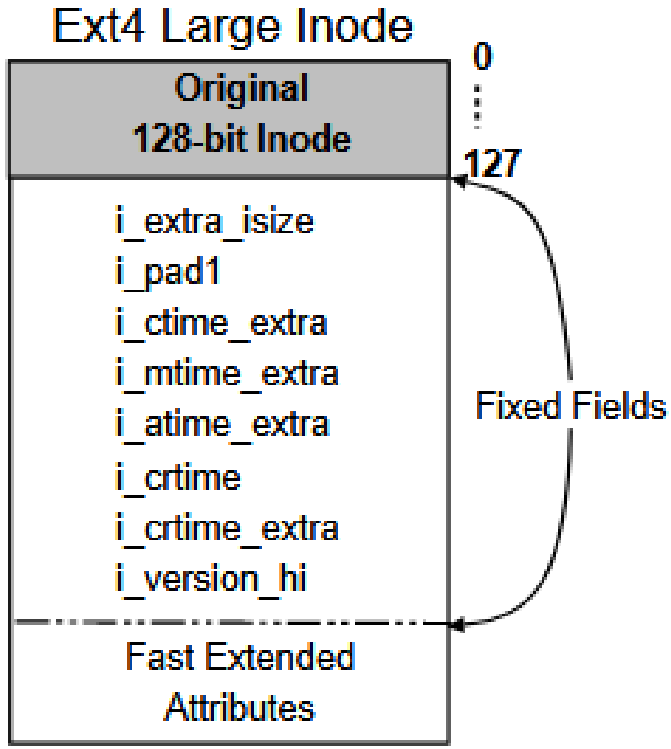
In ext4, the **default inode** structure size is **256 bytes**.

In order to avoid duplicating a lot of code in the kernel and e2fsck, the large **inodes** keep the same **fixed layout** for the **first 128-bytes**. The rest of the inode is split into two parts:

1. A fixed-field section that allows addition of fields common to all inodes, such as **nanosecond timestamps**.

2. A section for **Fast Extended Attributes** (**EAs**) that consumes the rest of the inode (support for fast EAs in large inodes has been available in Linux kernels since 2.6.12).

The first use of EAs was to store file **ACLs** and other **security data** (selinux).



Source : [The new ext4 filesystem: current status and future plans]

SKIP

Journal

Journal checksums

The journal is one of the **most intensively used** areas of the disk, therefore data corruption in this area is particularly severe for the entire file system.

Ext4 introduces the **journal checksumming**.

The **checksum** is calculated for each **transaction** and each **block group descriptor**.

Larger overhead for I/O operations, but allows to transform two journal writing phases known from ext3 to one, which improves both reliability and performance.

Fast commits for ext4, Marta Rybczyńska, January 2021.

Barriers

This mechanism allows to **order** the **disk driver** to **save data** in a **specific order**, which prevents data from being split in case of an emergency.

The file system must explicitly **instruct** the **disk** that **writing** data to the **journal** must **precede writing** the **record** with a **commit**.

Barriers are used for this.

Nanosecond timestamps

Ext4 introduces **time stamps** with **nanosecond** precision.

The new time stamps also shift the problem of 2038 by another **204 years**.

Additional reading

- Documentation/filesystems/ext2.txt.
- State of the Art: Where we are with the Ext3 filesystem, M. Cao, T. Y. Ts'o, B. Pulavarty, S. Bhattacharya, IBM.
- A Directory Index for Ext2, Daniel Phillips, 2001.
- Journaling the Linux ext2fs Filesystem, LinuxExpo, Stephen C. Tweedie, 1998.
- Anatomy of Linux journaling file systems, M. Tim Jones, IBM.
- Ext3, Wikipedia, the free encyclopedia, 7 maja 2010.
- Ext3 removal, quota & udf fixes (Linus Torwalds, September 2015)

  So the thing I'm happy to see is that the ext4 developers seem to unanimously agree that maintaining ext3 compatibility is part of their job, and nobody seems to be arguing for keeping ext3 around.

  Assuming no major objections come up, the EXT3 file-system driver will be dropped for the Linux 4.3 kernel.
- File system design case studies (Paul Krzyzanowski, March 2012)

Additional reading

- Documentation/filesystems/ext4.txt.
- Ext4 wiki.
- Ext4 Howto.
- Ext4 Disk Layout.
- Ext4, FOSDEM, Theodore Ts’o, 2009.
- Ted Ts'o on the ext4 filesystem, NYLUG, Theodore Ts’o, 2013.
- Ext4 block and inode allocator improvements, A. Kumar, M. Cao, J. Santos, A. Diliger, 2008 Linux Symposium.
- Case-insensitive ext4, Jake Edge, March 2019.
- The new ext4 filesystem: current status and future plans, A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, L. Vivier, 2007 Linux Symposium.
- Ext4: The Next Generation of Ext2/3 Filesystem, M. Cao, S. Bhattacharya, T. Tso, IBM, 2007.
- A Minimum Complete Tutorial of Linux ext4 File System, Mete Balci, 2017.
- Understanding Linux filesystems: ext4 and beyond, Jim Salter, April 2018
- How do SSDs work?, Joel Hruska, ExtremeTech, 2024.