Understanding Ext4 Disk Layout, Part 2

Srivathsa Dara ⋮ 23-29 minutes

Welcome to the second blog post of the Understanding Ext4 Disk Layout series. In the previous blog, we explored the concepts of the Superblock, GDT, Bitmaps, and Inode Table. In this blog, our focus will shift toward the on-disk structures of the extent tree, hash tree, and their corresponding data structures.

i_block

The `i_block` field, present in the `struct ext4_inode`, occupies 60 bytes (`EXT4_N_BLOCKS == 15`). This field serves as a container for block-related information associated with the inode.

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
struct ext4_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size_lo;      /* Size in bytes */
        __le32  i_atime;        /* Access time */
        __le32  i_ctime;        /* Inode Change time */
        __le32  i_mtime;        /* Modification time */
        __le32  i_dtime;        /* Deletion Time */
        __le16  i_gid;          /* Low 16 bits of Group Id */
        __le16  i_links_count;  /* Links count */
        __le32  i_blocks_lo;    /* Blocks count */
        __le32  i_flags;        /* File flags */
        union {
                struct {
                        __le32  l_i_version;
                } linux1;
                struct {
                        __u32  h_i_translator;
                } hurd1;
                struct {
                        __u32  m_i_reserved1;
                } masix1;
        } osd1;                         /* OS dependent 1 */
        __le32  i_block[EXT4_N_BLOCKS];/* Pointers to blocks */
        __le32  i_generation;   /* File version (for NFS) */
```

```
        __le32  i_file_acl_lo;  /* File ACL */
        __le32  i_size_high;
        __le32  i_obso_faddr;   /* Obsoleted fragment address */
        union {
                struct {
                        __le16  l_i_blocks_high; /* were l_i_reserved1
*/
                        __le16  l_i_file_acl_high;
                        __le16  l_i_uid_high;   /* these 2 fields */
                        __le16  l_i_gid_high;   /* were reserved2[0]
*/
                        __le16  l_i_checksum_lo;/*
crc32c(uuid+inum+inode) LE */
                        __le16  l_i_reserved;
                } linux2;
                struct {
                        __le16  h_i_reserved1;  /* Obsoleted fragment
number/size which are removed in ext4 */
                        __u16   h_i_mode_high;
                        __u16   h_i_uid_high;
                        __u16   h_i_gid_high;
                        __u32   h_i_author;
                } hurd2;
                struct {
                        __le16  h_i_reserved1;  /* Obsoleted fragment
number/size which are removed in ext4 */
                        __le16  m_i_file_acl_high;
                        __u32   m_i_reserved2[2];
                } masix2;
        } osd2;                         /* OS dependent 2 */
        __le16  i_extra_isize;
        __le16  i_checksum_hi;  /* crc32c(uuid+inum+inode) BE */
        __le32  i_ctime_extra;  /* extra Change time      (nsec << 2 |
epoch) */
        __le32  i_mtime_extra;  /* extra Modification time(nsec << 2 |
epoch) */
        __le32  i_atime_extra;  /* extra Access time      (nsec << 2 |
epoch) */
        __le32  i_crtime;       /* File Creation time */
        __le32  i_crtime_extra; /* extra FileCreationtime (nsec << 2 |
epoch) */
        __le32  i_version_hi;   /* high 32 bits for 64-bit version */
        __le32  i_projid;       /* Project ID */
};
```

Extent Tree

An extent refers to a group of physically contiguous blocks. This grouping reduces the need for direct block mapping between logical and physical blocks, which, in turn, reduces the amount of metadata to be maintained. That results in improved performance and efficiency.

An extent tree is a data structure that maintains the extents associated with an inode. The extent tree helps in faster traversal and retrieval of data.

The `i_block` field stores structures such as `struct ext4_extent_header`, `struct ext4_extent_idx`, and `struct ext4_extent`, which are integral components of the extent tree. Each of these structures have a size of 12 bytes.

As previously stated, the `i_block` field has a size of 60 bytes. Within these 60 bytes, the initial 12 bytes are allocated for the extent header. The remaining 48 bytes can be utilized by either `ext4_extent_idx` or `ext4_extent`, depending on the depth of the extent tree. Consequently, the `i_block` field can store an extent header along with a maximum of 4 extents or 4 extent indices, depending on the configuration of the extent tree.


```
struct ext4_extent_header {
        __le16  eh_magic;       /* probably will support different
formats */
        __le16  eh_entries;     /* number of valid entries */
        __le16  eh_max;         /* capacity of store in entries */
        __le16  eh_depth;       /* has tree real underlying blocks? */
        __le32  eh_generation;  /* generation of the tree */
};
```



```
struct ext4_extent_idx {
        __le32  ei_block;       /* index covers logical blocks from
'block' */
        __le32  ei_leaf_lo;     /* pointer to the physical block of
the next *
                                 * level. leaf or next index could be
there */
        __le16  ei_leaf_hi;     /* high 16 bits of physical block */
```

```
        __u16   ei_unused;
};
```

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

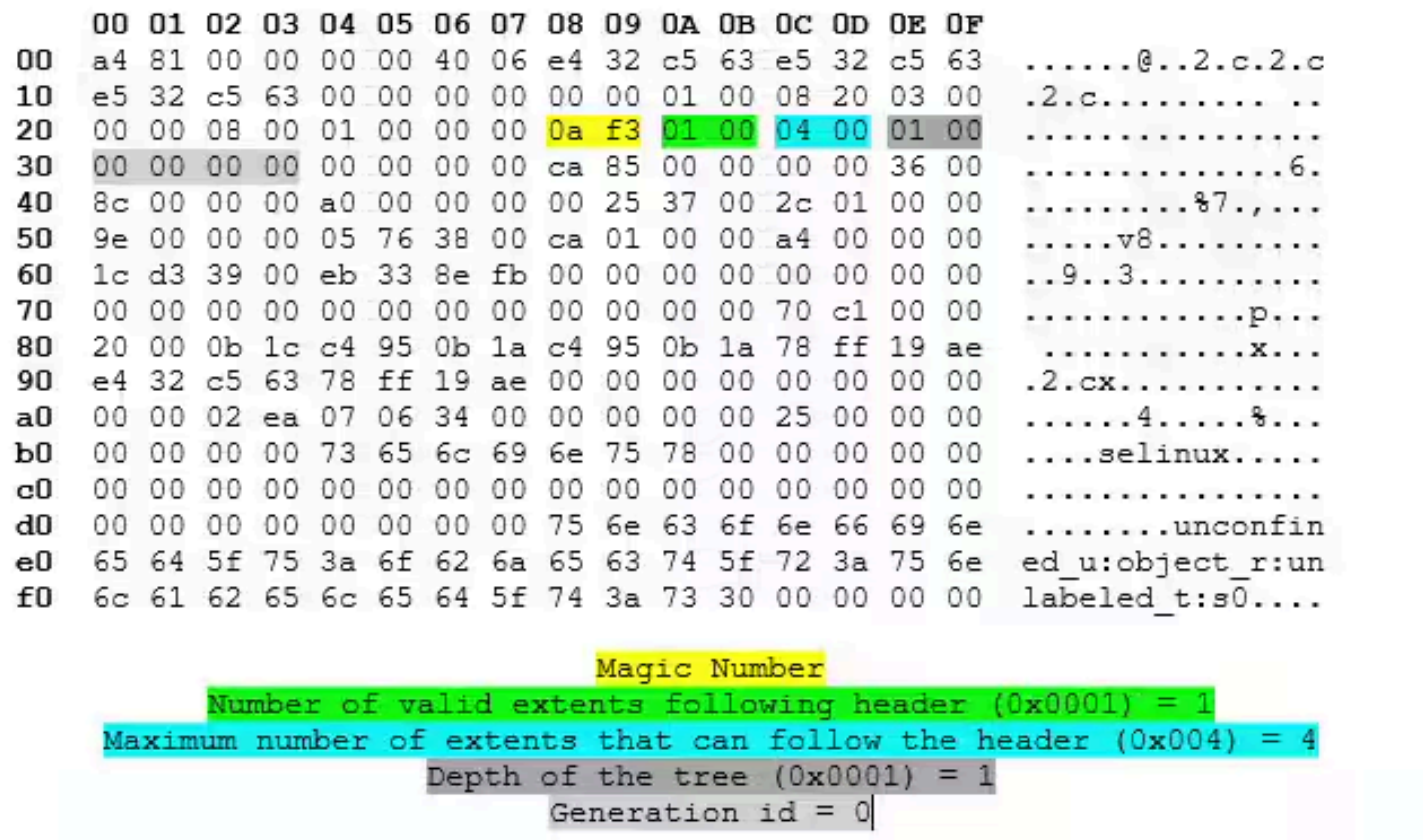
Error: Could not Copy

```
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 */
}
```

Now the question comes, what is the purpose of these structures? * `ext4_extent_header` - The header provides vital information about the extent tree. It holds the magic number, depth of the extent tree, and the number of valid entries (these entries can be either `ext4_extent_idx` or `ext4_extent`), etc. When the depth is zero, then the entries are `ext4_extents` otherwise `ext4_extent_idxs`. The header occupies the first 12 bytes in all blocks of the extent tree. * `ext4_extent_idx` - This structure serves as an index, it holds references to further levels of the tree. * `ext4_extent` - Represents an extent. These structures are exclusively present in the leaf blocks of the extent tree and provides details regarding the initial block and the length of the extent.

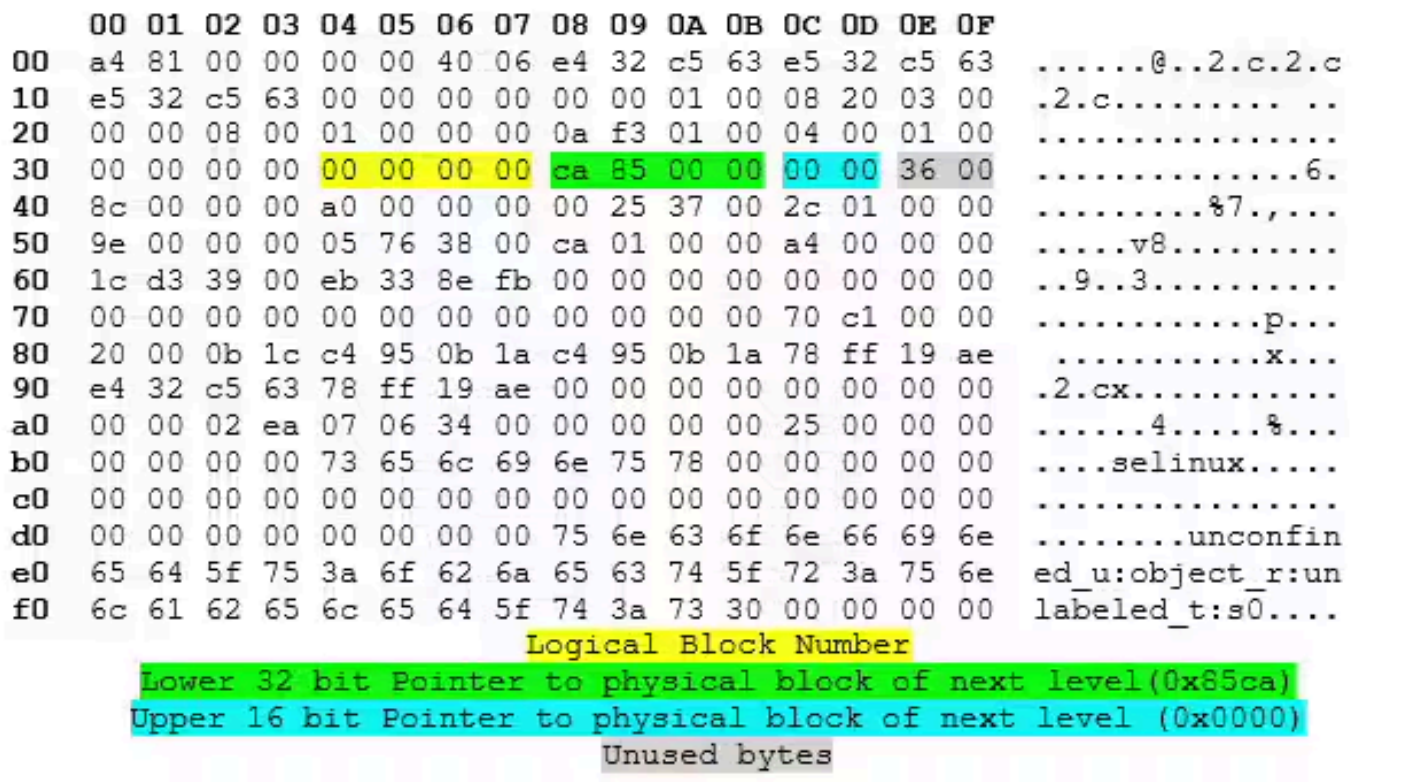
Let’s take an example by analyzing the hexdump of an inode.

Note: *Ext4 uses little endian notation. That means the least significant byte is stored first and the most significant byte is stored last. For example, if `0x1234` is written on disk, then its value in memory is `0x3412`.*



Above is the hexdump of an inode, with the highlighted region representing the extent header. From the extent header, it is evident that the extent tree has a depth of 1. Consequently, the extent header is followed by `struct ext4_extent_idx` instances.

The image below showcases the highlighted `struct ext4_extent_idx`. The extent index is pointing to the block `0x85ca`.



Let’s analyze the dump of that particular block for further insights.

```
debugfs:  bd 0x85ca
0000   0a f3 b6 00 54 01 00 00 00 00 00 00 00 00 00 00   ....T...........
0010   8c 00 00 00 34 92 36 00 8c 00 00 00 a0 00 00 00   ....4.6.........
0020   00 25 37 00 2c 01 00 00 9e 00 00 00 05 76 38 00   .%7.,.........v8.
0030   ca 01 00 00 a4 00 00 00 1c d3 39 00 6e 02 00 00   ..........9.n...
0040   90 00 00 00 4a 83 39 00 fe 02 00 00 8e 00 00 00   ....J.9.........
0050   32 d7 3b 00 8c 03 00 00 8d 00 00 00 f3 96 3b 00   2.;...........;.
0060   19 04 00 00 8a 00 00 00 76 19 3c 00 a3 04 00 00   ........v.<.....
0070   92 00 00 00 ee e6 3d 00 35 05 00 00 91 00 00 00   ......=.5.......
0080   66 ca 3f 00 c6 05 00 00 91 00 00 00 00 ef 3f 00   f.?...........?.
0090   57 06 00 00 80 00 00 00 80 b1 3f 00 d7 06 00 00   W.........?.....
00A0   7d 00 00 00 80 21 41 00 54 07 00 00 7e 00 00 00   }....!A.T...~...
00B0   c8 a7 42 00 d2 07 00 00 2e 00 00 00 ad 81 45 00   ..B...........E.
00C0   00 08 00 00 cd 00 00 00 00 f7 46 00 cd 08 00 00   ..........F.....
00D0   c6 00 00 00 80 6c 47 00 93 09 00 00 f8 00 00 00   .....lG.........
00E0   4f 58 48 00 8b 0a 00 00 ec 00 00 00 ad 32 48 00   OXH..........2H.
00F0   77 0b 00 00 c0 00 00 00 00 6e 49 00 37 0c 00 00   w........nI.7...
```

Magic Number

Number of valid extents following extent header (0xb6) = 182

Maximum number of extents that can follow extent header (0x154) = 340

Depth of the tree = 0

Generation id

Upon inspection, we observe that the first 12 bytes correspond to the extent header. Notably, the maximum number of extents that can follow the extent header is 340. This calculation is derived by considering that each block has a size of 4k bytes, with the first 12 bytes allocated to the extent header and the final 4 bytes utilized by `struct ext4_extent_tail`. Consequently, this leaves 4080 bytes available. Since each extent occupies 12 bytes, dividing 4080 by 12 results in 340 extents. Additionally, it is worth noting that the depth of this block is zero, indicating that we reached leaf block of the extent tree. As previously mentioned, in a leaf block, the entries following the extent header are `ext4_extent` structures.

The number of valid extents in the given context is 182, which means 182 extents follow the extent header. Each of these extents is associated with a specific range of data blocks and contains a pointer that indicates the location of the first data block within its respective extent.

Here is the hexdump of the same block, but this time `ext4_extent` are highlighted:

```
debugfs:  bd 0x85ca
0000  0a f3 b6 00 54 01 00 00 00 00 00 00 00 00 00 00  ....T...........
0010  8c 00 00 00 34 92 36 00 8c 00 00 00 a0 00 00 00  ....4.6.........
0020  00 25 37 00 2c 01 00 00 9e 00 00 00 05 76 38 00  .%7.,........v8.
0030  ca 01 00 00 a4 00 00 00 1c d3 39 00 6e 02 00 00  ..........9.n...
0040  90 00 00 00 4a 83 39 00 fe 02 00 00 8e 00 00 00  ....J.9.........
0050  32 d7 3b 00 8c 03 00 00 8d 00 00 00 f3 96 3b 00  2.;...........;.
0060  19 04 00 00 8a 00 00 00 76 19 3c 00 a3 04 00 00  ........v.<.....
0070  92 00 00 00 ee e6 3d 00 35 05 00 00 91 00 00 00  ......=.5.......
0080  66 ca 3f 00 c6 05 00 00 91 00 00 00 00 ef 3f 00  f.?...........?.
0090  57 06 00 00 80 00 00 00 80 b1 3f 00 d7 06 00 00  W.........?.....
00A0  7d 00 00 00 80 21 41 00 54 07 00 00 7e 00 00 00  }....!A.T...~...
00B0  c8 a7 42 00 d2 07 00 00 2e 00 00 00 ad 81 45 00  ..B...........E.
00C0  00 08 00 00 cd 00 00 00 00 f7 46 00 cd 08 00 00  ..........F.....
00D0  c6 00 00 00 80 6c 47 00 93 09 00 00 f8 00 00 00  .....lG.........
00E0  4f 58 48 00 8b 0a 00 00 ec 00 00 00 ad 32 48 00  OXH..........2H.
00F0  77 0b 00 00 c0 00 00 00 00 6e 49 00 37 0c 00 00  w........nI.7...

1st Extent:
Logical Block Number = 0
No. of blocks in the extent (0x8c) = 140 blocks in the extent
Upper 16bit address of 1st block = 0
Lower 32bit address of 1st block (0x369234) = 3576372

2nd Extent:
Logical Block Number (0x8c) = 140
No. of blocks in the extent (0xa0) = 160 blocks in the extent
Upper 16bit address of 1st block = 0
Lower 32bit address of 1st block (0x372500) = 3613952

3rd Extent:
Logical Block Number (0x12c) = 300
No. of blocks in the extent (0x9e) = 158 blocks in the extent
Upper 16bit address of 1st block = 0
Lower 32bit address of 1st block (0x387605) = 3700229
```

Upon analyzing the provided hexdump, we can observe the following information from the first three `struct ext4_extent` entries:

The first extent has a logical block value of 0. This extent encompasses 140 blocks, with the first block address pointing to the 3,576,372nd block. Therefore, the 140 consecutive blocks starting from the 3,576,372nd block belong to this extent, and they represent actual data blocks.

In the second extent, the logical block value is 140, indicating that the first 140 blocks are part of the previous extent. This extent comprises 160 blocks, and the first block of this extent is located at the 36,139,252nd block.

The third extent has a logical block value of 300, which is the sum of the blocks present in the previous two extents (140 + 160). This extent encompasses 158 blocks, and its first block is situated at the 3700229th block.

Similarly, the remaining 179 `ext4_extent` entries provide information about the extents associated with the inode, allowing for the determination of the corresponding data blocks and their respective ranges.

Debugfs also provides extent tree information of an inode. `ex` command serves this purpose. Here are the first few lines of the output obtained from the `debugfs ex` command:

```
debugfs:  ex <13>
Level Entries       Logical          Physical Length Flags
 0/ 1   1/  1     0 - 25599   34250              25600
 1/ 1   1/182     0 -   139 3576372 - 3576511    140
 1/ 1   2/182   140 -   299 3613952 - 3614111    160
 1/ 1   3/182   300 -   457 3700229 - 3700386    158
 1/ 1   4/182   458 -   621 3789596 - 3789759    164
 1/ 1   5/182   622 -   765 3769162 - 3769305    144
 1/ 1   6/182   766 -   907 3921714 - 3921855    142
 1/ 1   7/182   908 -  1048 3905267 - 3905407    141
 1/ 1   8/182  1049 -  1186 3938678 - 3938815    138
 1/ 1   9/182  1187 -  1332 4056814 - 4056959    146
 1/ 1  10/182  1333 -  1477 4180582 - 4180726    145
 1/ 1  11/182  1478 -  1622 4189952 - 4190096    145
 1/ 1  12/182  1623 -  1750 4174208 - 4174335    128
 1/ 1  13/182  1751 -  1875 4268416 - 4268540    125
 1/ 1  14/182  1876 -  2001 4368328 - 4368453    126
 1/ 1  15/182  2002 -  2047 4555181 - 4555226     46
 1/ 1  16/182  2048 -  2252 4650752 - 4650956    205
 1/ 1  17/182  2253 -  2450 4680832 - 4681029    198
 1/ 1  18/182  2451 -  2698 4741199 - 4741446    248
 1/ 1  19/182  2699 -  2934 4731565 - 4731800    236
 1/ 1  20/182  2935 -  3126 4812288 - 4812479    192
```

Looking at the 2nd, 3rd, and 4th rows of the provided output, we can observe that the Logical, Physical, and Length columns align with the logical block, first physical block, and length of the corresponding extents discussed earlier. By cross-referencing these values, we can confirm the accurate representation of the extent tree information for the given inode.

We have seen the case of an extent tree with a depth of 1, there is a single intermediate block between the `i_block` field and the actual data blocks. This means that there is one level of indirection between the inode and the data blocks.

To further illustrate, if the depth of the extent tree were 2, then there would be two intermediate blocks between the `i_block` field and the actual data blocks.

However, in the special case of a zero depth extent tree, the `i_block` field directly holds the `ext4_extent` structures. There are no `ext4_extent_idx` structures in this case, as there is no need for intermediate blocks. The `ext4_extent` structures within the `i_block` directly point to the extents that represent the data blocks. In this case of zero depth, the `i_block` has a maximum capacity of holding four `ext4_extents` and an `ext4_extent_header` whose size adds up to 60 bytes. Since the `i_block` itself is 60 bytes, when a 5th extent is allocated, the extent tree's depth is increased to accommodate the new extent.

This mechanism allows for efficient navigation and retrieval of data blocks within the Ext4 file system, providing different levels of indirection based on the depth of the extent tree.

Hash Tree

A directory stores the dirents of all its files in its data blocks. In the context of a directory that contains a large number of files, performing a linear search for a specific directory entry within its data blocks can become time-consuming and inefficient. To address this issue, the Ext4 file system implements a hash tree.

The hash tree provides a more efficient way to locate, insert or delete a directory entry within a directory. The hierarchical structure of hash trees allows for faster lookup than linear searchs. A specific hash algorithm is used to hash the directory entries, and the resulting hash values are used to navigate through the hash tree.

Typically, a directory in the Ext4 file system initially follows an unindexed approach for storing its directory entries. In this unindexed mode, directory entries are stored sequentially within a single data block of the directory. This approach is efficient when the number of directory entries are contained within a single data block.

However, once the number of directory entries exceeds the capacity of a single data block, the directory can be converted to an indexed tree structure, i.e. a hash tree. This conversion occurs when the hash tree feature is enabled.

The hash tree structure includes root node, intermediate nodes and leaf nodes.

The root node holds essential metadata about the structure and characteristics of the tree. This metadata includes information like the depth of the tree, hash algorithm used, etc. The intermediate nodes act as branches in the tree, guiding the search process based on hashed values derived from the directory entries. These intermediate nodes direct the search algorithm to the appropriate subtree that potentially contains the desired directory entry. On the other hand, the leaf nodes hold the actual directory entries themselves.

Below is the dirent structure which is stored by the directory in its data block:

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
struct ext4_dir_entry_2 {
        __le32  inode;                  /* Inode number */
        __le16  rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[EXT4_NAME_LEN];    /* File name */
};
```

Note: *EXT4_NAME_LEN is 255, so the maximum size for a file name in EXT4 is 255.*

Let’s examine a directory with an inode number of 131007, which contains a total of 2500 files.

```
[opc@sridara-s mnt]$ ls dir/dir2/ | wc -l
2500
[opc@sridara-s mnt]$ ls -i dir/ | grep dir2
131077 dir2
[opc@sridara-s mnt]$
```

Below is the inode dump of the directory:

```
debugfs:  id <131077>
0000  ed41 0000 00e0 0100 57ca d363 f1e7 c463  .A......W..c...c
0020  f1e7 c463 0000 0000 0000 0200 f000 0000  ...c............
0040  0010 0800 4d1d 0000 0af3 0100 0400 0000  ....M...........
0060  0000 0000 0000 0000 1e00 0000 6020 0800  ............` ..
0100  0000 0000 0000 0000 0000 0000 0000 0000  ................
*
0140  0000 0000 75d7 a14c 0000 0000 0000 0000  ....u..L........
0160  0000 0000 0000 0000 0000 0000 b572 0000  ............r..
0200  2000 6ea4 5447 f31f 5447 f31f ec4c 0a74   .n.TG..TG...L.t
0220  8be7 c463 2c34 dddb 0000 0000 0000 0000  ...c,4..........
0240  0000 02ea 0706 3400 0000 0000 2500 0000  ......4.....%...
0260  0000 0000 7365 6c69 6e75 7800 0000 0000  ....selinux.....
0300  0000 0000 0000 0000 0000 0000 0000 0000  ................
0320  0000 0000 0000 0000 756e 636f 6e66 696e  ........unconfin
0340  6564 5f75 3a6f 626a 6563 745f 723a 756e  ed_u:object_r:un
0360  6c61 6265 6c65 645f 743a 7330 0000 0000  labeled_t:s0....
```

Number of extents (0x0001) = 1

Depth (0x0) = 0

Underlined is 1st and only extent

Logical Block Number (0x0) = 0

No. of blocks in the Extent (0x001e) = 30

Upper 16 bit address of 1st block of the extent (0x0) = 0

Lower 32 bit address of 1st block of the extent (0x00082060) = 532576

If a directory is unindexed, its data blocks will store dirents (directory entries). However, if a directory is indexed (i.e is using hash tree), the first few data blocks will store information related to the hash tree structure, while the remaining data blocks will store the dirents. In the case of a hash tree, the 1st data block serves as the host for the hash tree root, represented by `struct dx_root`, followed by `struct dx_entry` instances. These `dx_entry` structures provide the addresses of the next level blocks within the hash tree hierarchy.

Copied to Clipboard

Error: Could not Copy

```
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];
};
```

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

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

The `struct fake_dirent` serves as the directory entry for the dot (.) and dotdot (..) directories within the `dx_root` structure. It is similar to the `struct ext4_dir_entry_2`, but it lacks a name field since the name field is incorporated within the `dx_root` structure.

Copied to Clipboard

Error: Could not Copy

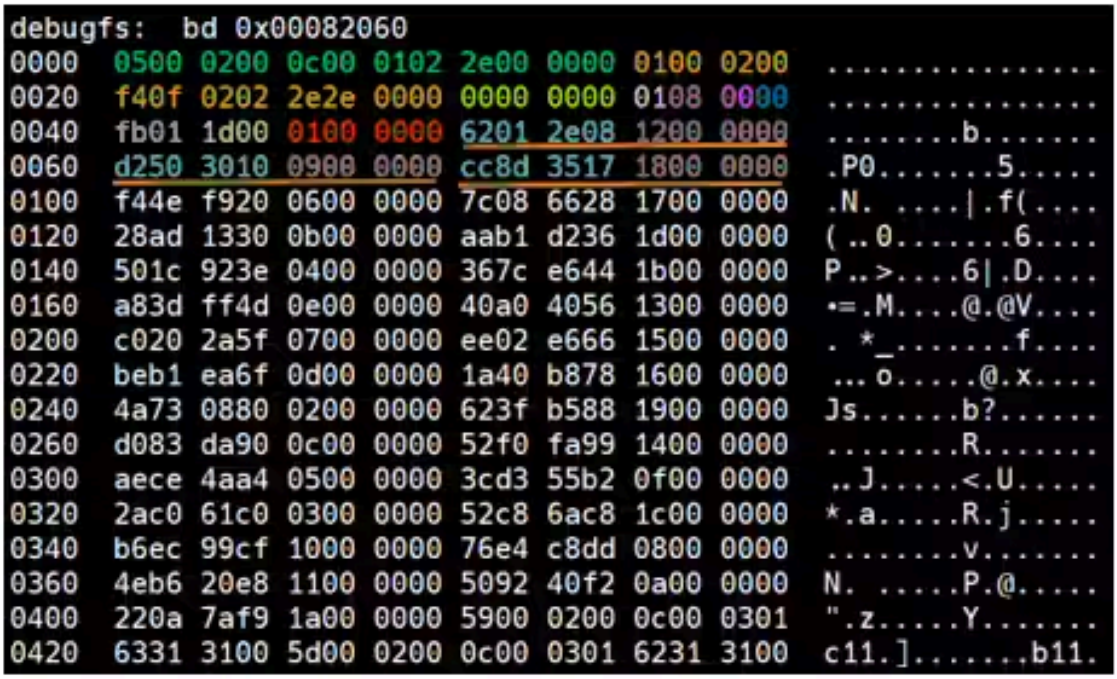
Copied to Clipboard

Error: Could not Copy

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

From the above image, it is evident that the depth of the extent tree is zero. Furthermore, there is only a single extent present in the extent tree. The first block of this extent is the 532,576th block, and it is followed by 29 consecutive blocks, collectively representing the extent associated with the inode.

Here is the dump of the initial few bytes of the first block (Block number: 532,576):



The first 36 bytes are used by dx_root, and following dx_root we have 28 dx_entrys. We can observe that the number of valid entries in dx_root is 29, considering the 28 dx_entrys and the dx_root itself. The first three dx_entrys are highlighted in the provided hexdump. Additionally, the depth of the tree is indicated as 0, indicating that it is a leaf level in the hash tree structure.

In a dx_entry, the first 4 bytes represent the hash value, and the next 4 bytes represent the logical block number. This logical block corresponds to a data block that stores directory entries (dirents) whose hash values fall between the hash value of the current dx_entry and the hash value of the next dx_entry in the hash tree structure.

Below is a table of hash values and logical blocks of the first 3 dx_entrys:

Hash value

Logical block number

Let's consider an example where we are trying to add a dirent with a hash value of 0x092e0111. By comparing the hash value with the hash values in the table above, we find that it is greater than 0x082e0162 and less than 0x103050d2. Therefore, the directory entry will be inserted into the logical block 12, which corresponds to the data block that stores dirents whose hash values are between 0x082e0162 and 0x103050d2.

Similarly, if we are searching for a dirent with a hash value greater than 0 and less than 0x082e0162, we need to search block 1 to find the target directory entry. On the other hand, if the target dirent hash value is greater than 0x103050d2 and less than 0x17358dcc, we need to search block 9 to find the desired dirent. This process allows us to locate the directory entry based on its hash value within the hash tree structure.

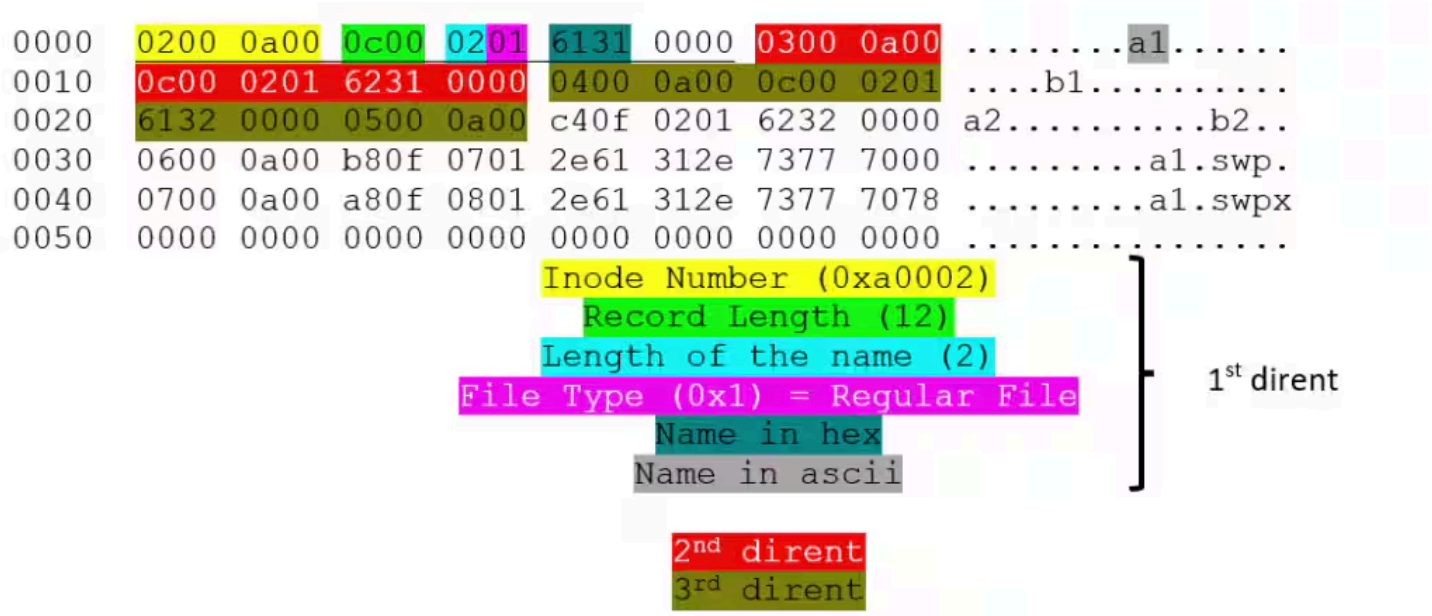
When depth is > 0:

In the case where the hash tree depth is 2, there will be two levels of intermediate blocks between the leaf block (which stores directory entries) and the dx_root block. These intermediate blocks exclusively store dx_entry blocks.

During the search for a file in the hash tree, we only examine a single block at each level until we reach the leaf block. At the leaf block, we perform a linear search to find the desired dirent. This hierarchical search process significantly improves efficiency compared to an unindexed directory, where a linear search is conducted across all blocks of the directory.

Different Hash algorithms:

Below is the hexdump of one of the leaf blocks in the hash tree:



`a1`, `b1`, `a2`, `b2` etc are the files in the directory, so their dirents are stored in the data block of the directory.

Let's take a closer look at the dirent of file a1 in the directory. The structure of the dirent is as follows:

1. The first four bytes represent the inode number of the file.
2. The next two bytes indicate the record length of the dirent.
3. The subsequent byte represents the length of the file name.
4. Following that, we have the file type, which provides information about the type of the file (e.g., regular file, directory, symbolic link, etc.).
5. Finally, we encounter the actual name of the file.

The `rec_len` of the last dirent will be such that it spans till the end of the block.

This structure allows us to retrieve important details about the file, such as its inode number, length, type, and name, from the dirent stored in the directory's data block.

The table below provides a list of file types:

0x3

Character device file

Tree size

As the number of dirents increases in a directory, the size and depth of the tree used to store them will also increase.

Initially, a directory is in an unindexed state, where dirents are stored in a single block. However, if the number of dirents exceeds the capacity of a single block, the directory is converted to an indexed tree structure. During this conversion process, two new blocks are created. One of these blocks becomes the root block and is initialized with a `dx_root` structure. The dirents are then sorted in ascending order, and the first half of the dirents remain in the original block while the remaining dirents are moved to the other new block. Two `dx_entry` structures are added to the `dx_root` to maintain the directory's indexed state.

Suppose the directory was initially unindexed and contained 40 dirents, with the data block of the directory being completely full. When the 41st dirent is added, the directory undergoes a conversion to an indexed structure.

During the conversion, two new blocks are created. One of these blocks becomes the root block, and the 40 dirents are sorted. The first 20 dirents remain in the original block, while the other 20 dirents are moved to the new block. The `dx_entry` of the new block will have a hash value equal to the hash of the first dirent in that block. On the other hand, the `dx_entry` of the initial block will have a hash value of zero.

With this reorganization, the addition of the 41st dirent can now be accommodated in the directory's indexed structure.

After the conversion of the directory to an indexed structure, if any leaf block becomes filled with dirents, a new block will be created to accommodate additional dirents. The dirents from the filled leaf block will be redistributed equally between the filled block and the new block.

This redistribution ensures that the dirents are evenly distributed among the leaf blocks, preventing any single block from becoming overloaded with dirents. By creating a new block and redistributing the dirents, the indexed structure of the directory can efficiently store and manage a larger number of dirents while maintaining balanced block utilization.

Example: The image below depicts the initial configuration of a directory's hash tree. At this point, the hash tree has a depth of 0 and consists of three leaf blocks (1, 2, and 3).

```
debugfs:  htree dir2
Root node dump:
        Reserved zero: 0
        Hash Version: 1
        Info length: 8
        Indirect levels: 0
        Flags: 0
Number of entries (count): 3
Number of entries (limit): 507
Checksum: 0x26e53965
Entry #0: Hash 0x00000000, block 1
Entry #1: Hash 0x7d9c8fa2, block 2
Entry #2: Hash 0xbfe55cc0, block 3
```

Now, let's consider a scenario where 20 new files are created within the same directory. After adding these new dirents, the hash tree configuration is updated as follows:

```
debugfs:  htree dir2
Root node dump:
        Reserved zero: 0
        Hash Version: 1
        Info length: 8
        Indirect levels: 0
        Flags: 0
Number of entries (count): 4
Number of entries (limit): 507
Checksum: 0x1e7a9fd0
Entry #0: Hash 0x00000000, block 1
Entry #1: Hash 0x46aa34e4, block 4
Entry #2: Hash 0x7d9c8fa2, block 2
Entry #3: Hash 0xbfe55cc0, block 3
```

The depth of the tree remains unchanged, indicating that no additional levels were required. However, a new leaf block (block 4) is created and inserted between block 1 and block 2 in the hash tree. This is necessary because some of the newly added entries filled up block 1 completely. To accommodate the additional dirents, a new block (block 4) is created, and the dirents are evenly redistributed between block 1 and block 4 to maintain a balanced hash tree structure.

Collision

In the event of a hash collision, it is important to note that the hash value is primarily used to locate the block that holds dirents within a specific hash value range. If a collision occurs and multiple dirents share the same hash value, they will be stored together in the same block. Once the target block is located, a linear search is performed within that block to find the specific dirent.

When adding a new dirent that experiences a hash collision with a dirent residing in a completely filled block, a new block will be created. The dirents will be redistributed equally between the two blocks based on their hash values. However, it is worth noting that dirents with the same hash values will still remain together in the same block. This ensures that despite the redistribution, dirents with colliding hash values are kept in the same block, allowing for efficient retrieval during linear search operations.

Conclusion

In this blog, we explored two important concepts: extent tree and hash tree, along with the supporting structures associated with them. We learned that filesystems often allocate non-contiguous blocks for storing data, and the extent tree provides a mechanism to link these scattered blocks together, enabling efficient data retrieval and management.

Additionally, we delved into the hash tree, which serves as a powerful tool for directories with a significant number of files. The hash tree scheme facilitates quick and efficient search, addition, and deletion of directory entries (dirents). This feature becomes particularly valuable in directories that house a large volume of files, as it minimizes the need for linear searches and ensures swift access to the desired dirents.

In summary, the extent tree and hash tree play vital roles in optimizing filesystem performance and organization. They enable efficient allocation of non-contiguous blocks and streamline operations within directories, leading to improved overall system efficiency and effectiveness.

References

- Kernel code (fs/ext4/) - 5.4.17-2136.310.7.1.el8uek.x86_64
- https://www.kernel.org/doc/html/latest/filesystems/ext4/
- https://www.sans.org/blog/understanding-ext4-part-3-extent-trees/
- https://www.sans.org/blog/understanding-ext4-part-6-directories/

Previous Chapter Next Chapter