Understanding Ext4 Disk Layout, Part 1

Srivathsa Dara ⋮ 27-34 minutes

Overview

This blog is the first in a series of blogs in which we are going to look at the disk layout of the ext4 filesystem. We will be looking at different types of on-disk structures in newly created ext4 filesystems and understand their importantance.

To create an ext4 filesystem on a device, let's say the device be sdb1, run the following command:

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
mkfs.ext4 /dev/sdb1
```

The `mkfs.ext4` creates an ext4 filesystem on device `sdb1`. `mkfs.ext4` can take many arguments which facilitate the specification and customizaton of ext4 features. We can either enable or disable a feature depending upon our specific requirements. The default values for ext4 features for newly created ext4 filesystems is contained in the configuration file `/etc/mke2fs.conf`. These default feature values can be overridden by specifying command line arguments to `mkfs.ext4`.

Lets take a look at the contents of `/etc/mke2fs.conf`:

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
[defaults]
        base_features =
sparse_super,large_file,filetype,resize_inode,dir_index,ext_attr
        default_mntopts = acl,user_xattr
        enable_periodic_fsck = 0
        blocksize = 4096
        inode_size = 256
        inode_ratio = 16384

[fs_types]
        ...
        ext4 = {
                features =
has_journal,extent,huge_file,flex_bg,metadata_csum,64bit,dir_nlink,extra_isize
```

```
                inode_size = 256
        }
        ...
```

Note: *Only contents related to ext4 filesystem are displayed.*

The configuration settings for `base_features` and `features` are enabled by default. If we want to disable any of these or enable any other feature, we can do so using `mkfs.ext4`. Some other default settings of interest might be that periodic fsck is disabled, the default blocksize being set to 4096 bytes, the inode size is 256 bytes and the inode_ratio is 16384.

Knowing the default values of `blocksize`, `inode_size` and `inode_ratio` is important because these parameters decide the number of blocks, the number of block groups and the number of inodes in a filesystem. As already mentioned, these values can be changed with `mkfs.ext4`. Among these, `blocksize` and `inode_size` are self explanatory, however `inode_ratio` needs some explanation.

inode_ratio: This gives the number of bytes for which, an inode is created. That means an inode is created for every `inode_ratio` bytes.

In our case, we have `inode_ratio = 16384`, so `mkfs.ext4` creates a inode for every 16384 bytes. If we run `mkfs.ext4` on a 1GiB device, it creates 65536 inodes in the filesystem - as 1Gib/16364bytes = 65536. Therefore, the number of inodes in a filesystem is given by (filesystem size)/(inode_ratio).

Layout

The following are important on-disk structures of an ext4 filesystem: - Superblock - Block Group Descriptor, Group Descriptor Table - Inode Bitmap - Block Bitmap - Inode Table - Extent Tree - Hash Tree - Journal

In this blog, we will cover up to the Inode Table, the rest will be covered in subsequent blogs.

An ext4 filesystem is divided into block groups. Each block group has 8*blocksize (in bytes) number of blocks in it. If we consider the default blocksize, which is 4096 bytes, the number of blocks in blockgroup is given by 8*4096 = 32768 blocks in each group.

Size of each block group = (Number of blocks in each group) * blocksize

The number of block groups can be obtained by dividing the filesystem size by the size of each block group. (filesystem size)/(size of each block group).

Generally the default block size is 4KiB, using the above formulae the number of blocks in each block group will be 32768 and the block group size will be of length 128MiB. For a 1GiB filesystem we will have 8 (1 GiB/128 MiB) block groups.

Each block group has its associated block group descriptor, block bitmap, inode bitmap and inode table.

So, a filesystem with 8 block groups has 8 block group descriptors, 8 block bitmaps, 8 inode bitmaps and 8 inode tables.

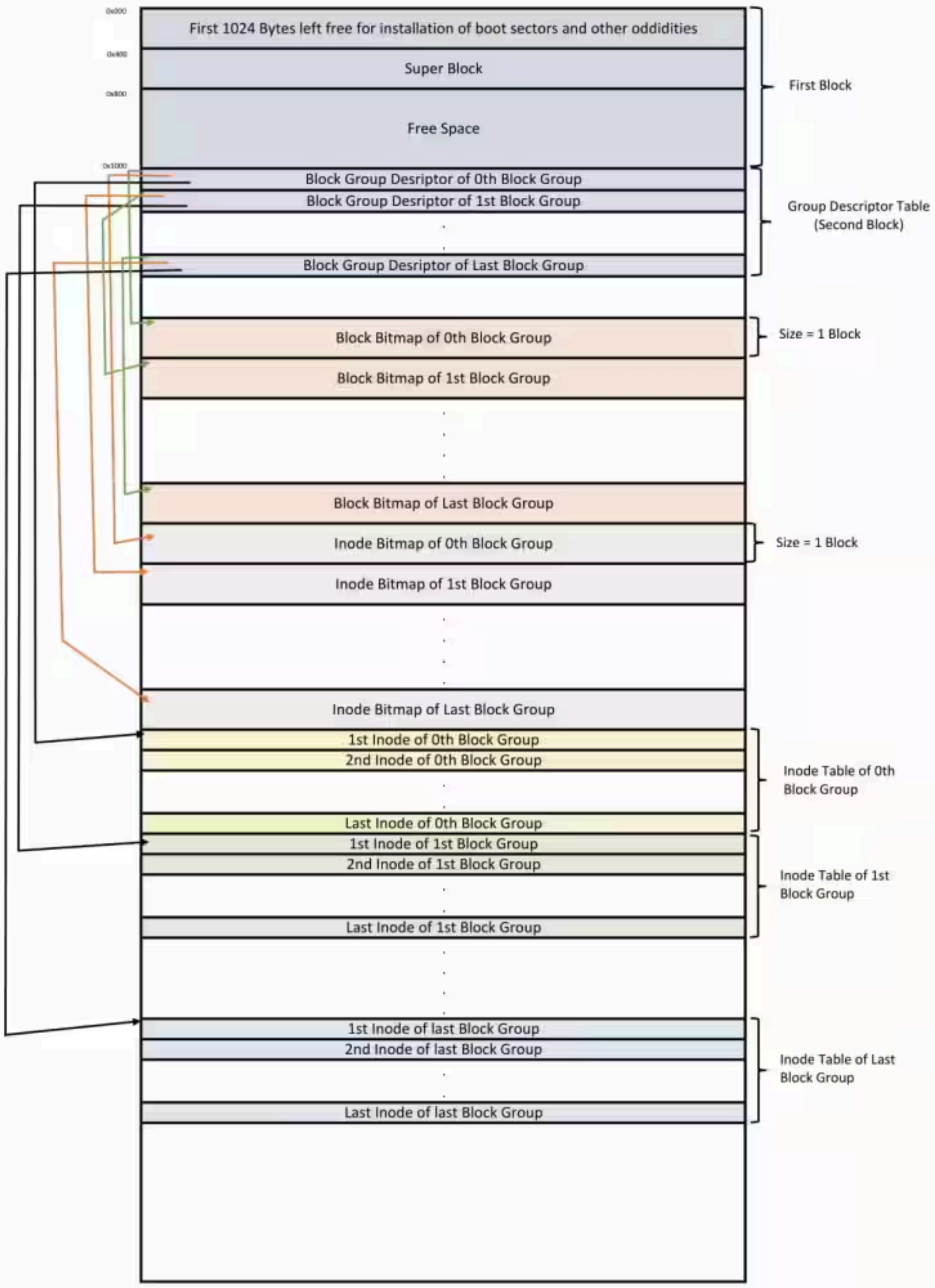
Block group descriptors for all groups together is called the Group Descriptor Table (GDT).

No. of inodes in each group = (Total Number of Inodes) / (No. block groups)

Inode Table Size = No. of Inodes in each Group * inode_size

Note: *For simplicity, we have used 1 GiB filesystem with blocksize = 4096, inode_size = 256, inode_ratio = 16384 as an example through out the document.*

Graphical view of Disk Layout



Note: When flex groups are used (see below for more info), the metadata of the entire flex group is contained in the first block group of the flex group. In the above example the filesystem has only 1 flex group. So, the metadata of all block groups are contained in only the first block group.

Superblock

On an ext4 filesystem, the first block contains the Superblock. `struct ext4_super_block` is the on-disk structure of the Superblock. The size of `struct ext4_super_block` is 1024 Bytes. In the first block of the filesystem, the first 1024 bytes are left for the installation of boot sectors and other oddities. And the next 1024 bytes are used for the Superblock. the remaining 2048 bytes in first block remain unused.

On disk ext4 Superblock structure

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

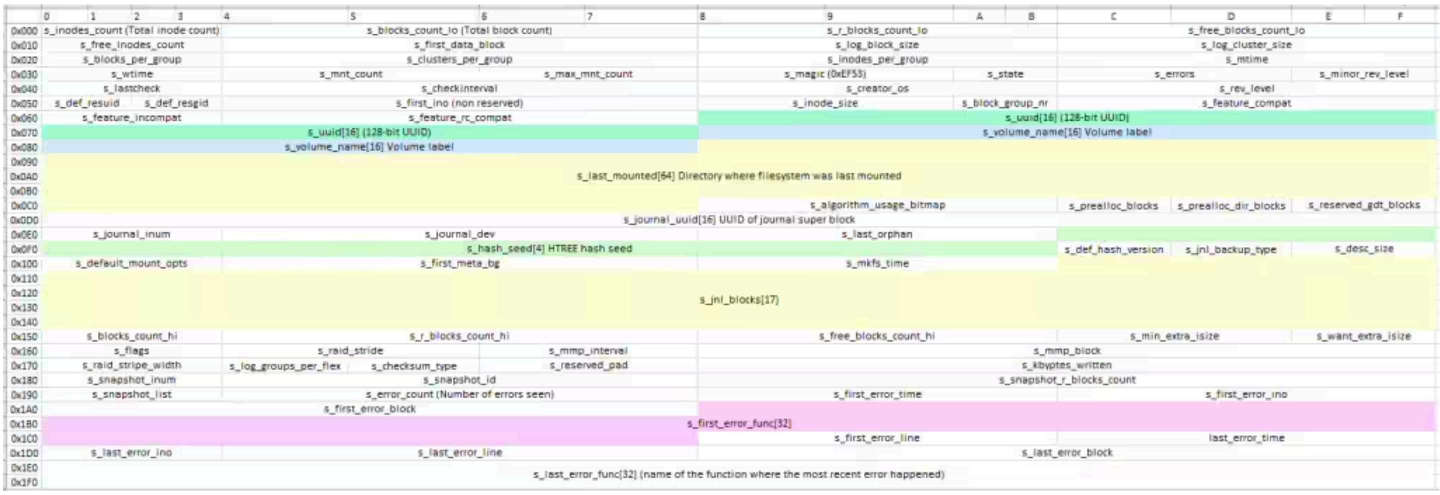
```
struct ext4_super_block {
/*00*/  __le32  s_inodes_count;         /* Inodes count */
        __le32  s_blocks_count_lo;      /* Blocks count */
        __le32  s_r_blocks_count_lo;    /* Reserved blocks count */
        __le32  s_free_blocks_count_lo; /* Free blocks count */
/*10*/  __le32  s_free_inodes_count;    /* Free inodes count */
        __le32  s_first_data_block;     /* First Data Block */
        __le32  s_log_block_size;       /* Block size */
        __le32  s_log_cluster_size;     /* Allocation cluster size */
/*20*/  __le32  s_blocks_per_group;     /* # Blocks per group */
        __le32  s_clusters_per_group;   /* # Clusters per group */
        __le32  s_inodes_per_group;     /* # Inodes per group */
        __le32  s_mtime;                /* Mount time */
/*30*/  __le32  s_wtime;                /* Write time */
        __le16  s_mnt_count;            /* Mount count */
        __le16  s_max_mnt_count;        /* Maximal mount count */
        __le16  s_magic;                /* Magic signature */
        __le16  s_state;                /* File system state */
        __le16  s_errors;               /* Behaviour when detecting errors
*/
        __le16  s_minor_rev_level;      /* minor revision level */
/*40*/  __le32  s_lastcheck;            /* time of last check */
        __le32  s_checkinterval;        /* max. time between checks */
        __le32  s_creator_os;           /* OS */
        __le32  s_rev_level;            /* Revision level */
/*50*/  __le16  s_def_resuid;           /* Default uid for reserved blocks
*/
        __le16  s_def_resgid;           /* Default gid for reserved blocks
*/
        /*
         * These fields are for EXT4_DYNAMIC_REV Superblocks only.
         *
         * Note: the difference between the compatible feature set and
         * the incompatible feature set is that if there is a bit set
         * in the incompatible feature set that the kernel doesn't
         * know about, it should refuse to mount the filesystem.
         *
```

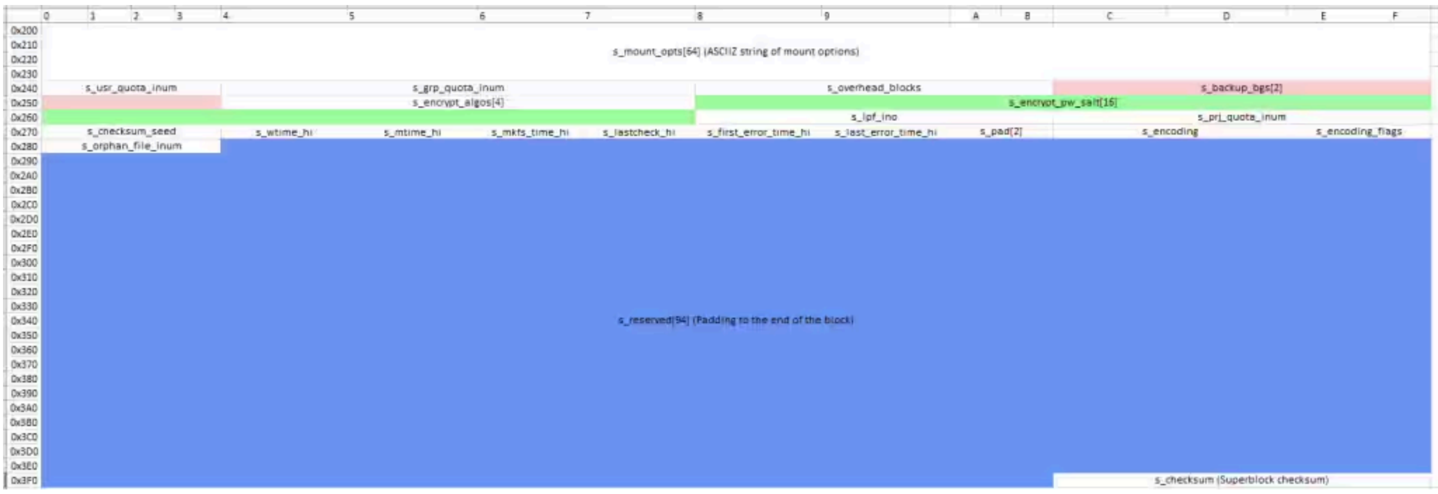
```
         * e2fsck's requirements are more strict; if it doesn't know
         * about a feature in either the compatible or incompatible
         * feature set, it must abort and not try to meddle with
         * things it doesn't understand...
         */
        __le32  s_first_ino;            /* First non-reserved inode */
        __le16  s_inode_size;           /* size of inode structure */
        __le16  s_block_group_nr;       /* block group # of this Superblock
*/
        __le32  s_feature_compat;       /* compatible feature set */
/*60*/  __le32  s_feature_incompat;     /* incompatible feature set */
        __le32  s_feature_ro_compat;    /* readonly-compatible feature set
*/
/*68*/  __u8    s_uuid[16];             /* 128-bit uuid for volume */
/*78*/  char    s_volume_name[16];      /* volume name */
/*88*/  char    s_last_mounted[64] __nonstring; /* directory where last
mounted */
/*C8*/  __le32  s_algorithm_usage_bitmap; /* For compression */
        /*
         * Performance hints.  Directory preallocation should only
         * happen if the EXT4_FEATURE_COMPAT_DIR_PREALLOC flag is on.
         */
        __u8    s_prealloc_blocks;      /* Nr of blocks to try to
preallocate*/
        __u8    s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */
        __le16  s_reserved_gdt_blocks;  /* Per group desc for online growth
*/
        /*
         * Journaling support valid if EXT4_FEATURE_COMPAT_HAS_JOURNAL set.
         */
/*D0*/  __u8    s_journal_uuid[16];     /* uuid of journal Superblock */
/*E0*/  __le32  s_journal_inum;         /* inode number of journal file */
        __le32  s_journal_dev;          /* device number of journal file */
        __le32  s_last_orphan;          /* start of list of inodes to
delete */
        __le32  s_hash_seed[4];         /* HTREE hash seed */
        __u8    s_def_hash_version;     /* Default hash version to use */
        __u8    s_jnl_backup_type;
        __le16  s_desc_size;            /* size of group descriptor */
/*100*/ __le32  s_default_mount_opts;
        __le32  s_first_meta_bg;        /* First metablock block group */
        __le32  s_mkfs_time;            /* When the filesystem was created
*/
        __le32  s_jnl_blocks[17];       /* Backup of the journal inode */
        /* 64bit support valid if EXT4_FEATURE_COMPAT_64BIT */
/*150*/ __le32  s_blocks_count_hi;      /* Blocks count */
        __le32  s_r_blocks_count_hi;    /* Reserved blocks count */
        __le32  s_free_blocks_count_hi; /* Free blocks count */
        __le16  s_min_extra_isize;      /* All inodes have at least # bytes
*/
        __le16  s_want_extra_isize;     /* New inodes should reserve #
```

```
bytes */
        __le32  s_flags;                /* Miscellaneous flags */
        __le16  s_raid_stride;          /* RAID stride */
        __le16  s_mmp_update_interval;  /* # seconds to wait in MMP
checking */
        __le64  s_mmp_block;            /* Block for multi-mount protection
*/
        __le32  s_raid_stripe_width;    /* blocks on all data disks
(N*stride)*/
        __u8    s_log_groups_per_flex;  /* FLEX_BG group size */
        __u8    s_checksum_type;        /* metadata checksum algorithm used
*/
        __u8    s_encryption_level;     /* versioning level for encryption
*/
        __u8    s_reserved_pad;         /* Padding to next 32bits */
        __le64  s_kbytes_written;       /* nr of lifetime kilobytes written
*/
        __le32  s_snapshot_inum;        /* Inode number of active snapshot
*/
        __le32  s_snapshot_id;          /* sequential ID of active snapshot
*/
        __le64  s_snapshot_r_blocks_count; /* reserved blocks for active
                                              snapshot's future use */
        __le32  s_snapshot_list;        /* inode number of the head of the
                                           on-disk snapshot list */
#define EXT4_S_ERR_START offsetof(struct ext4_super_block, s_error_count)
        __le32  s_error_count;          /* number of fs errors */
        __le32  s_first_error_time;     /* first time an error happened */
        __le32  s_first_error_ino;      /* inode involved in first error */
        __le64  s_first_error_block;    /* block involved of first error */
        __u8    s_first_error_func[32] __nonstring;     /* function where
the error happened */
        __le32  s_first_error_line;     /* line number where error happened
*/
        __le32  s_last_error_time;      /* most recent time of an error */
        __le32  s_last_error_ino;       /* inode involved in last error */
        __le32  s_last_error_line;      /* line number where error happened
*/
        __le64  s_last_error_block;     /* block involved of last error */
        __u8    s_last_error_func[32] __nonstring;      /* function where
the error happened */
#define EXT4_S_ERR_END offsetof(struct ext4_super_block, s_mount_opts)
        __u8    s_mount_opts[64];
        __le32  s_usr_quota_inum;       /* inode for tracking user quota */
        __le32  s_grp_quota_inum;       /* inode for tracking group quota
*/
        __le32  s_overhead_clusters;    /* overhead blocks/clusters in fs
*/
        __le32  s_backup_bgs[2];        /* groups with sparse_super2 SBs */
        __u8    s_encrypt_algos[4];     /* Encryption algorithms in use  */
        __u8    s_encrypt_pw_salt[16];  /* Salt used for string2key
```

```
algorithm */
        __le32  s_lpf_ino;              /* Location of the lost+found inode
*/
        __le32  s_prj_quota_inum;       /* inode for tracking project quota
*/
        __le32  s_checksum_seed;        /* crc32c(uuid) if csum_seed set */
        __u8    s_wtime_hi;
        __u8    s_mtime_hi;
        __u8    s_mkfs_time_hi;
        __u8    s_lastcheck_hi;
        __u8    s_first_error_time_hi;
        __u8    s_last_error_time_hi;
        __u8    s_pad[2];
        __le16  s_encoding;             /* Filename charset encoding */
        __le16  s_encoding_flags;       /* Filename charset encoding flags
*/
        __le32  s_reserved[95];         /* Padding to the end of the block
*/
        __le32  s_checksum;             /* crc32c(Superblock) */
};
```

Byte level view of an Ext4 Superblock





Superblock state (0x03A to 0x03B): * 0001 - Cleanly umounted (This is the case in above hexdump) * 0002 - Errors detected * 0004 - Orphans being recovered

Filesystem creator OS (0x048 to 0x049) * 0 - Linux (Linux the creator OS in the above hexdump) * 1 - Hurd * 2 - Masix * 3 - FreeBSD * 4 - Lites

Superblock encrypt algorithms (0x254 to 0x257) * 0 - Invalid Algorithm (ENCRYPTION_MODE_INVALID) * 1 - 256-bit in XTS mode (ENCRYPTION_MODE_AES_256_XTS) * 2 - 256-bit AES in GCM mode (ENCRYPTION_MODE_AES_256_GCM) * 3 - 256-bit AES in CBC mode (ENCRYPTION_MODE_AES_256_CBC)

Superblock Hexdump

Note: *Ext4 uses little endian notation.*

```
          00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000400  00 00 01 00 00 00 04 00  33 33 00 00 65 cd 03 00  |........33..e...|
00000410  f5 ff 00 00 00 00 00 00  02 00 00 00 02 00 00 00  |................|
00000420  00 80 00 00 00 80 00 00  00 20 00 00 00 00 00 00  |......... ......|
00000430  3e ac 4f 63 00 00 ff ff  53 ef 01 00 01 00 00 00  |>.Oc....S.......|
00000440  3e ac 4f 63 00 00 00 00  00 00 00 00 01 00 00 00  |>.Oc............|
00000450  00 00 00 00 0b 00 00 00  00 01 00 00 3c 00 00 00  |............<...|
00000460  c2 02 00 00 6b 04 00 00  0f 63 64 99 7a 63 40 77  |....k....cd.zc@w|
00000470  8d c5 45 f4 ac c3 1f 7c  00 00 00 00 00 00 00 00  |..E....|........|
00000480  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000004c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 7f 00  |................|
000004d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000004e0  08 00 00 00 00 00 00 00  00 00 00 00 f8 79 95 8b  |.............y..|
000004f0  f1 2f 47 37 85 eb 54 ed  e6 26 87 82 01 01 40 00  |./G7..T..&....@.|
00000500  0c 00 00 00 00 00 00 00  3e ac 4f 63 0a f3 01 00  |........>.Oc....|
00000510  04 00 00 00 00 00 00 00  00 00 00 00 00 20 00 00  |............. ..|
00000520  00 00 02 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000530  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000540  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 02  |................|
00000550  00 00 00 00 00 00 00 00  00 00 00 00 20 00 20 00  |............ . .|
00000560  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000570  00 01 00 00 04 01 00 00  15 02 00 00 00 00 00 00  |................|
00000580  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000007f0  00 00 00 00 00 00 00 00  00 00 00 00 e1 d7 51 0e  |..............Q.|
00000800
```

```
Block size of 1024 Bytes unused for Boot Sector Purposes
Total inode count (00010000) = 65536 inodes
Total Block Count (00040000) = 262144 Blocks
Free Block Count (0003cd65) = 249189 Free Blocks
Free Inode Count (0000fff5) = 65525 Free inodes
First Data Block (00000000) is at zeroth Block
Blocks per Group (00008000) = 32768 Blocks
Inodes per Group (00002000) = 8192 inodes
Magic Number (ef53)
128 bit UUID of Volume
128 bit UUID of Journal Super Block
S_mkfs_time since the epoch (634fac3e) = 1666165822 seconds
Flags
Raid stripe width (00000100) = 256
Super Block Checksum
```

A few important fields of the Superblock are highlighted in the hexdump. The total size of the Superblock is 1024 bytes, which spans from 0000400 to 00007ff.

Endianness: Ext4 uses little endian notation. In little endian notation the rightmost byte will be the most significant byte.

For Example, from the above hexdump image, if we observe the magic number in the hexdump is `53 ef`, but the actual magic number of an ext4 filesystem is `0xef53`. As ext4 uses little endian, the most significant byte will be the right most byte. So, `ef` is most significant byte and `53` is least significant byte.

Primary and Backup Superblocks Backup copies of the Superblock are written to some of the block groups across the disk - in case the beginning of the disk gets trashed, backup Superblocks can be used for recovery. Not all blocks groups will have a Superblock copy. In our case (1GiB ext4 filesystem) apart from the primary Superblock, other copies are present in the first, third, fifth and seventh block groups. The 0th block group will host the primary Superblock.

The Superblock contains a lot of information regarding the filesystem, so if the primary Superblock gets corrupted then vital information pertaining to the filesystem is compromised. To overcome such situations, backup Superblocks are stored at different places in the filesystem.

In earlier versions such as ext2, backup Superblocks were created in every block group. But the ext4 filesystem has a `sparse_super` feature, which when enabled results in backup Superblocks only being created in blockgroups 0, 1 and powers of 3, 5 and 7 (e.g. powers of 3 = 9, 27, 81 …..; powers of 5 = 25, 125, 625 ….; powers of 7 = 49, 343, 2401 …).

Backup Superblocks and backup GDTs are never updated by the kernel. They will get updated only if any fundamental parameter of the filesystem is amended, for example, resizing the filesystem. Programs like `resize2fs` and `tune2fs` will change structural parameters and cause updates to the backup Superblocks and GDTs.

Block Group Descriptors

A block group is a logical grouping of contigous blocks, whose size is equal to the number of bits in one block. For example, in a filesystem with a blocksize of 4096 bytes, a block group will have 4096*8 = 32768 blocks. Each block group is represented by its block group descriptor, which stores information such as free inodes, free blocks and the location of the block bitmap, inode bitmap and the inode table of that particular block group.

The Group Descriptor Table (GDT) contains all block group descriptors, we can call the GDT a table of block group descriptors of all block groups of the filesystem.

The GDT will be present immediately after the Superblock block, and backup copies of the GDT are stored similarly to the backup copies of the Superblock. If `sparse_super` is set then just like Superblock backups, GDT backups will be in block groups 0, 1 and powers of 3, 5, 7 (e.g. 27, 125, 343 etc), if `sparse_super` is not set, then these backup copies will be present in all block groups.

GDTs will follow Superblocks, the primary GDT will be present in second block. Each block group descriptor is of 64 Bytes, so for a filesystem with 8 block groups there will be a group decriptor table occupying 512 Bytes (8*64) in the second block, and the remaining block will be unused.

On disk block group descriptor is represented by **struct ext4_group_desc**

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
/*
 * Structure of a blocks group descriptor
 */
struct ext4_group_desc
{
        __le32  bg_block_bitmap_lo;     /* Blocks bitmap block */
        __le32  bg_inode_bitmap_lo;     /* Inodes bitmap block */
        __le32  bg_inode_table_lo;      /* Inodes table block */
        __le16  bg_free_blocks_count_lo;/* Free blocks count */
        __le16  bg_free_inodes_count_lo;/* Free inodes count */
        __le16  bg_used_dirs_count_lo;  /* Directories count */
        __le16  bg_flags;               /* EXT4_BG_flags (INODE_UNINIT,
etc) */
        __le32  bg_exclude_bitmap_lo;   /* Exclude bitmap for snapshots */
        __le16  bg_block_bitmap_csum_lo;/* crc32c(s_uuid+grp_num+bbitmap)
LE */
        __le16  bg_inode_bitmap_csum_lo;/* crc32c(s_uuid+grp_num+ibitmap)
LE */
        __le16  bg_itable_unused_lo;    /* Unused inodes count */
        __le16  bg_checksum;            /* crc16(sb_uuid+group+desc) */
        __le32  bg_block_bitmap_hi;     /* Blocks bitmap block MSB */
        __le32  bg_inode_bitmap_hi;     /* Inodes bitmap block MSB */
        __le32  bg_inode_table_hi;      /* Inodes table block MSB */
        __le16  bg_free_blocks_count_hi;/* Free blocks count MSB */
        __le16  bg_free_inodes_count_hi;/* Free inodes count MSB */
        __le16  bg_used_dirs_count_hi;  /* Directories count MSB */
        __le16  bg_itable_unused_hi;    /* Unused inodes count MSB */
        __le32  bg_exclude_bitmap_hi;   /* Exclude bitmap block MSB */
        __le16  bg_block_bitmap_csum_hi;/* crc32c(s_uuid+grp_num+bbitmap)
BE */
        __le16  bg_inode_bitmap_csum_hi;/* crc32c(s_uuid+grp_num+ibitmap)
BE */
        __u32   bg_reserved;
};
```

Byte level view of block group descriptor

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	bg_block_bitmap_lo				bg_inode_bitmap_lo				bg_inode_table_lo				free_block_count_lo		free_block_inode_lo	
0x10	bg_used_dirs_count_lo		bg_flags		bg_exclude_bitmap_lo				bg_block_bitmap_csum_lo		bg_inode_bitmap_csum_lo		bg_itable_unused_lo		bg_checksum	
0x20	bg_block_bitmap_hi				bg_inode_bitmap_hi				bg_inode_table_hi				bg_free_blocks_count_hi		bg_free_inodes_count_hi	
0x30	bg_used_dirs_count_hi		bg_itable_unused_hi		bg_exclude_bitmap_hi				bg_block_bitmap_csum_hi		bg_inode_bitmap_csum_hi		bg_reserved			

Hexdump of a Group Descriptor Table

The following is a GDT hexdump of a newly created ext4 filesystem of 1GiB size. As the GDT follows the Superblock’s block, the GDT will start at 4096th byte (0x1000).

Filesystem’s block size is 4096 Bytes

No. of blocks in each block group = 8 * blocksize = 32768 blocks.

Size of each block group = no. of blocks * blocksize = 32768 * 4096 = 134,217,728 = 128MiB

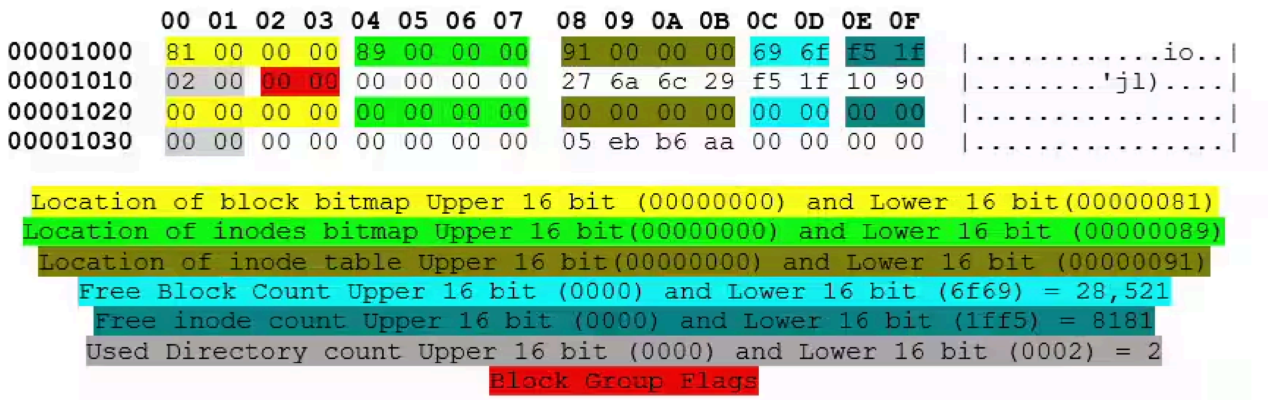
No. of block groups = (Size of File System)/(Size of each block group) = 1GiB/128MiB = 8.

Therefore a GDT will have 8 block group descriptors in it.

```
         00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F
00001000 81 00 00 00 89 00 00 00  91 00 00 00 69 6f f5 1f  |............io..|
00001010 02 00 00 00 00 00 00 00  27 6a 6c 29 f5 1f 10 90  |........'jl)....|
00001020 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001030 00 00 00 00 00 00 00 00  05 eb b6 aa 00 00 00 00  |................|
00001040 82 00 00 00 8a 00 00 00  91 02 00 00 7f 7f 00 20  |............... |
00001050 00 00 03 00 00 00 00 00  00 00 00 00 00 20 0d 36  |............. .6|
00001060 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001080 83 00 00 00 8b 00 00 00  91 04 00 00 00 80 00 20  |............... |
00001090 00 00 03 00 00 00 00 00  00 00 00 00 00 20 d1 e8  |................|
000010a0 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000010c0 84 00 00 00 8c 00 00 00  91 06 00 00 7f 7f 00 20  |............... |
000010d0 00 00 03 00 00 00 00 00  00 00 00 00 00 20 ce 51  |............. .Q|
000010e0 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001100 85 00 00 00 8d 00 00 00  91 08 00 00 00 60 00 20  |.............`. |
00001110 00 00 01 00 00 00 00 00  62 00 00 00 00 20 e1 f0  |........b.... ..|
00001120 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001130 00 00 00 00 00 00 00 00  76 8a 00 00 00 00 00 00  |........v.......|
00001140 86 00 00 00 8e 00 00 00  91 0a 00 00 7f 7f 00 20  |............... |
00001150 00 00 03 00 00 00 00 00  00 00 00 00 00 20 43 a3  |............. C.|
00001160 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001180 87 00 00 00 8f 00 00 00  91 0c 00 00 00 80 00 20  |............... |
00001190 00 00 03 00 00 00 00 00  00 00 00 00 00 20 9f 7d  |............. .}|
000011a0 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000011c0 88 00 00 00 90 00 00 00  91 0e 00 00 7f 7f 00 20  |............... |
000011d0 00 00 01 00 00 00 00 00  d6 a7 00 00 00 20 b1 97  |................|
000011e0 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000011f0 00 00 00 00 00 00 00 00  6d 91 00 00 00 00 00 00  |........m.......|
```

1st Group's Block Group Descriptor
2nd Group's Block Group Descriptor
3rd Group's Block Group Descriptor
4th Group's Block Group Descriptor
5th Group's Block Group Descriptor
6th Group's Block Group Descriptor
7th Group's Block Group Descriptor
8th Group's Block Group Descriptor

Hexdump of first group descriptor of explained:



```
          00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F
00001000  81 00 00 00 89 00 00 00  91 00 00 00 69 6f f5 1f  |............io..|
00001010  02 00 00 00 00 00 00 00  27 6a 6c 29 f5 1f 10 90  |........'jl)....|
00001020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001030  00 00 00 00 00 00 00 00  05 eb b6 aa 00 00 00 00  |................|
```

Location of block bitmap Upper 16 bit (00000000) and Lower 16 bit(00000081)
Location of inodes bitmap Upper 16 bit(00000000) and Lower 16 bit (00000089)
Location of inode table Upper 16 bit(00000000) and Lower 16 bit (00000091)
Free Block Count Upper 16 bit (0000) and Lower 16 bit (6f69) = 28,521
Free inode count Upper 16 bit (0000) and Lower 16 bit (1ff5) = 8181
Used Directory count Upper 16 bit (0000) and Lower 16 bit (0002) = 2
Block Group Flags

Block Bitmap

The Block bitmap tracks the block usage status of all blocks of a block group. Each bit in the bitmap represents a block. If a block is in use, its corresponding bit will be set, otherwise it will be unset. The location of the block bitmap is not fixed, so its position is stored in respective block group descriptors.

From the above image, the location of the block bitmap is given as 0x81 (129 in decimal). That means the block bitmap is at 129th block. As each block is 4096 bytes, the block bitmap will start at 528384th byte, which is 81000 in hex. Below is a hexdump from that offset. The size of the block bitmap is 1 block (4096 bytes here).

Hexdump of block bitmap

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
00081000  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
|................|
*
00081210  ff ff 7f 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
00081220  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
*
00082000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
```

Inode Bitmap

Similar to the Block bitmap, the Inode bitmap's location is also not fixed, therefore the group descriptor points to the location of the Inode bitmap. The Inode bitmap tracks usage of inodes. Each bit in the bitmap represents an inode. If an inode is in use then its corresponding bit in Inode bitmap will be set, otherwise it will be unset.

Each Inode bitmap uses a block. As in this case we have only 8192 inodes in each group, only the first 1024 bytes are used as a bitmap. From the above hexdump of a block group descriptor, the location of the inode bitmap is given as 0x00000089 (137 in decimal). Therefore the inode bitmap is present at 137th block. As each block is 4096 bytes, the offset of the 137th block is 0x89000 (137 * 4096). Below is the hexdump of an inode bitmap at that offset.

Hexdump of inode bitmap

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
00089000  ff 07 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
00089010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
*
00089400  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff
|................|
*
0008a000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
```

From the above hexdump, we can see that starting from 0x00089400 every bit is set until 0x0008a000 because only the first 1024 bytes are used as a bitmap, these remaining bits are not part of bitmap.

Inode Table

The Inode Table is a table of all the inodes in a block group. In our example, the inode size is 256 bytes, and there has to be one inode for every 16384 bytes (`inode_ratio`). For a filesystem of 1 GiB, having 8 block groups, the total number of inodes will be 65536 (1Gib/16Kib = 65536), and 8192 inodes per block group (65536/8 = 8192). The size of the inode table will be 2Mib per block group (256*8192=2097152 = 2MiB). For 8 block groups, a space of 16 Mib is required for all inode tables, and 4096 blocks are used for this purpose (16MiB/4Kib).

The location of the inode table as given by the block group descriptor of the first block group is 0x00000091(145 in decimal). The offset of the 145th block is 0x00091000.

Hexdump of 1st inode in inode table

Copied to Clipboard

Error: Could not Copy

Copied to Clipboard

Error: Could not Copy

```
00091000  00 00 00 00 00 00 00 00  8a 45 7b 63 8a 45 7b 63
|.........E{c.E{c|
00091010  8a 45 7b 63 00 00 00 00  00 00 00 00 00 00 00 00
|.E{c............|
00091020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
```

```
*
00091070  00 00 00 00 00 00 00 00  00 00 00 00 df 46 00 00
|.............F..|
00091080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
*
00091100  ed 41 00 00 00 b0 1d 00  45 14 93 63 18 14 93 63
|.A......E..c...c|
```

Inode Structure

Copied to Clipboard

Error: Could not Copy

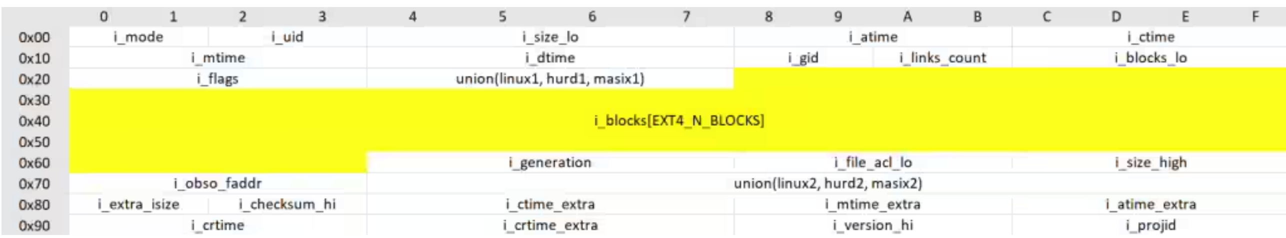
Copied to Clipboard

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 */
};
```

Byte level view of on disk inode



Inode hexdump

```
          00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F
00091d00  a4 81 00 00 00 00 00 00  18 14 93 63 18 14 93 63  |.........|.c...c|
00091d10  18 14 93 63 00 00 00 00  00 00 00 00 00 00 00 00  |.|.c...c........|
00091d20  00 00 08 00 01 00 00 00  0a f3 00 00 04 00 00 00  |................|
00091d30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00091d60  00 00 00 00 ef 80 fd 16  00 00 00 00 00 00 00 00  |................|
00091d70  00 00 00 00 00 00 00 00  00 00 00 00 09 c4 00 00  |................|
00091d80  20 00 ea 59 50 96 d5 1e  60 01 34 da 60 01 34 da  | ..YP...`.4.`.4.|
00091d90  4c 56 90 63 70 f6 aa 1c  00 00 00 00 00 00 00 00  |LV.cp...........|
00091da0  00 00 02 ea 07 06 34 00  00 00 00 00 25 00 00 00  |......4.....%...|
00091db0  00 00 00 00 73 65 6c 69  6e 75 78 00 00 00 00 00  |....selinux.....|
00091dc0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00091dd0  00 00 00 00 00 00 00 00  75 6e 63 6f 6e 66 69 6e  |........unconfin|
00091de0  65 64 5f 75 3a 6f 62 6a  65 63 74 5f 72 3a 75 6e  |ed_u:object_r:un|
00091df0  6c 61 62 65 6c 65 64 5f  74 3a 73 30 00 00 00 00  |labeled_t:s0....|
```

Access Time (i_atime)

Inode Change Time (i_ctime)

Modification Time (i_mtime)

Links Count (i_links_count)

Block Count (i_blocks_lo)

Extra Change Time (i_ctime_extra)

Extra Modification time (i_mtime_extra)

Extra Access time (i_atime_extra)

Above is the hexdump of the fourteenth inode in the inode table. The size of the on-disk inode structure `struct ext4_inode` is 160 bytes. The extra 96 bytes at the end are used to store the extended attributes.

From the above hexdump access time (i_atime) is 0x63931418 which is 1,670,583,320 in decimal. The following is `stat` output for the same inode.

```
[opc@sridara-s temper]$ stat f40mb
  File: f40mb
  Size: 41943040        Blocks: 81920      IO Block: 4096   regular file
Device: 813h/2067d      Inode: 14          Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Context: unconfined_u:object_r:unlabeled_t:s0
Access: 2022-12-09 10:55:20.417447861 +0000
Modify: 2022-12-09 10:55:20.569450558 +0000
Change: 2022-12-09 10:55:20.569450558 +0000
 Birth: 2022-12-09 10:55:20.417447861 +0000
```

If we convert our time stamp from hexdump to a readable date format, we see that it will match with the access time from the stat image.

```
[opc@sridara-s temper]$ date -d @1670583320
Fri Dec  9 10:55:20 GMT 2022
```

Flex Block Groups

In ext4 filesystems, we have a feature called `flex_bg`. If this feature is enabled, then the number of block groups given by `2^s_log_groups_per_flex` (`s_log_groups_per_flex` is a Superblock field) are grouped as a single flex group and the inode bitmaps, block bitmaps, and inode tables of all the groups in flex group are stored in the first block group of the flex group.

For example, if `s_log_groups_per_flex` is 4 then the size of flex group is 16 (2^4) and assume that the filesystem has 48 block groups in toteal. In this case, the filesystem will have 3 flex groups, each containing 16 block groups. All the inode bitmaps, block bitmaps and inode tables of all the groups of the first flext group (0-15 block groups), second flex group (16-31 block groups), and the last flext group (32-47 block groups) will be stored in 0th, 16th, and 32nd blockgroups respectively.

Note: This feature does not change the location of the backup Superblocks and backup GDTs.

Conclusion

On an ext4 filesystem, the on-disk data structures are present in the same order as discussed above. The Superblock, followed by the GDT, with the GDT pointing to the locations of corresponding block bitmaps, inode bitmaps and inode tables. When creating a filesystem, `mkfs.ext4` creates some important structures, we have only covered up to inode tables in this blog, and future blogs will cover the rest of the data structures.

References

- e2fsprogs code - 1.46.6-rc1
- Kernel code- 5.4.17-2136.310.7.1.el8uek.x86_64
- https://www.kernel.org/doc/html/latest/filesystems/ext4/index.html
- https://www.spinics.net/lists/linux-ext4/msg62155.html