Ext4 Setup

The COS default filesystem is a basic implementation of the Ext4 FileSystem. A minimal implementation of an Ext4 formatter called cos_mkfs was created for this purpose.

Superblock Definition

Writing the Superblock to Disk

Purpose of the Superblock

In a filesystem, the superblock is a critical data structure that contains metadata describing the overall layout and state of the filesystem.
It includes counts of inodes and blocks, block size information, timestamps, feature flags, and more.
This function builds the superblock structure for an EXT4-like filesystem and writes it to a specific disk location.

```
void write_superblock_to_disk(char *disk_address, size_t
        filesystem_size, size_t *offset, uint64_t unix_timestamp)
```

- **disk_address**: A pointer representing the start of the disk (or the disk image) in memory.
- **filesystem_size**: Total size of the filesystem in bytes.
- **offset**: A pointer to an offset variable where the superblock will be written. It is set within the function.
- **unix_timestamp**: Timestamp representing the current or last write time, used for superblock metadata.

The function assembles a superblock_t structure with many fields initialized based on the filesystem size and constants defined elsewhere.
Finally, it copies this structure to the disk image at the appropriate offset.

Core Calculations and Constants

```
uint32_t number_of_blocks_per_group = 8 * DEFAULT_BLOCK_SIZE;
size_t total_number_of_inodes = filesystem_size /
        DEFAULT_INODE_RATIO;
size_t number_of_block_groups = filesystem_size / (number_of_blocks_per_group
        * DEFAULT_BLOCK_SIZE);
uint32_t number_of_inodes_per_group = total_number_of_inodes /
        number_of_block_groups;
size_t total_number_of_blocks = number_of_block_groups *
        number_of_blocks_per_group;
size_t number_of_blocks_required_to_store_inodes = (DEFAULT_INODE_SIZE
        * total_number_of_inodes) / DEFAULT_BLOCK_SIZE;
```

- **number_of_blocks_per_group**: Defines how many blocks are grouped together in a block group.
  Calculated as 8 times the default block size (in bytes).
- **total_number_of_inodes**: Determines total number of inodes based on a predefined inode-to-bytes ratio (DEFAULT_INODE_RATIO).
- **number_of_block_groups**: Total block groups in the filesystem. Calculated by dividing the total filesystem size by the size of a block group.
- **number_of_inodes_per_group**: How many inodes each block group contains.
- **total_number_of_blocks**: Overall count of blocks in all block groups combined.
- **number_of_blocks_required_to_store_inodes**: Blocks needed to store all inodes, based on inode size and block size.

Offset Initialization

```
*offset = 0x400;
```

The superblock starts at offset 0x400 (1024 decimal), which is standard for EXT2/EXT3/EXT4 filesystems, reserving the first 1024 bytes for the boot sector or other bootloader data.

Superblock Field Assignments

```
superblock_header.s_inodes_count = filesystem_size /
        DEFAULT_INODE_RATIO;
superblock_header.s_blocks_count_lo = filesystem_size /
        DEFAULT_BLOCK_SIZE;
superblock_header.s_r_blocks_count_lo = 0;
superblock_header.s_free_blocks_count_lo = (total_number_of_blocks
        - number_of_blocks_required_to_store_inodes - 2 *
        number_of_block_groups - 2) & 0xFFFFFFFF;
```

```
superblock_header.s_free_inodes_count = total_number_of_inodes -
        11;
superblock_header.s_first_data_block = 0;
superblock_header.s_log_block_size = 2;
superblock_header.s_blocks_per_group = number_of_blocks_per_group;
superblock_header.s_inodes_per_group = number_of_inodes_per_group;
```

- **s_inodes_count**: Total inodes in the filesystem.
- **s_blocks_count_lo**: Total number of blocks (lower 32 bits).
- **s_r_blocks_count_lo**: Reserved blocks count, set to 0 initially.
- **s_free_blocks_count_lo**: Free blocks available, calculated by subtracting space taken by inodes, block groups metadata, and other reserved blocks.
- **s_free_inodes_count**: Free inodes available, subtracting 11 for reserved inodes.
- **s_first_data_block**: Starting block of data (0 here, meaning block 0 is data block).
- **s_log_block_size**: Logarithm of the block size (block size = 1024 << log_block_size).
  Here set to 2 (block size = 4KiB).
- **s_blocks_per_group**: Blocks in each group.
- **s_inodes_per_group**: Inodes in each group.

Cluster Size Handling

```
if (BIGALLOC_FLAG) {
    superblock_header.s_log_cluster_size =
        DEFAULT_LOG_CLUSTER_SIZE;
    superblock_header.s_clusters_per_group =
        DEFAULT_CLUSTERS_PER_GROUP_NUMBER;
} else {
    superblock_header.s_log_cluster_size =
        superblock_header.s_log_block_size;
    superblock_header.s_clusters_per_group =
        superblock_header.s_blocks_per_group;
}
```

If the filesystem uses bigalloc (allocation by clusters rather than blocks), cluster size and number per group are set differently. Otherwise, cluster size equals block size, and clusters per group equal blocks per group.

Time and Version Fields

```
superblock_header.s_mtime = 0;
superblock_header.s_wtime = unix_timestamp;
superblock_header.s_mnt_count = 0;
```

```
superblock_header.s_max_mnt_count = 0xFFFF;
superblock_header.s_magic = SUPERBLOCK_MAGIC;
superblock_header.s_state = 1;
superblock_header.s_errors = 1;
superblock_header.s_minor_rev_level = 0;
superblock_header.s_lastcheck = unix_timestamp;
superblock_header.s_checkinterval = DEFAULT_TIME_CHECKS;
superblock_header.s_creator_os = 0;
superblock_header.s_rev_level = 1;
superblock_header.s_def_resuid = DEFAULT_UID_RESERVED;
superblock_header.s_def_resgid = DEFAULT_GID_RESERVED;
```

These fields track timestamps of last write (s_wtime), last consistency check (s_lastcheck), mount counts, filesystem state, and error behaviors.
s_magic identifies the filesystem type (EXT4).
Revision and compatibility flags are also set here.

Inode and Feature Flags

```
superblock_header.s_first_ino = 11;
superblock_header.s_inode_size = DEFAULT_INODE_SIZE;
superblock_header.s_block_group_nr = 0;
superblock_header.s_feature_compat = 0;
superblock_header.s_feature_incompat = 0x40; // extents feature
        flag
superblock_header.s_feature_ro_compat = 0;
```

According to the EXT4 specification, the first valid inode is inode number 11.
The inode size and block group number are initialized.
Compatibility flags indicate that this filesystem supports extents (a modern way of tracking file data blocks).

UUID and Labels

```
cos_memcpy(superblock_header.s_uuid, (uint8_t[16])
        {DEFAULT_UUID_VOLUME}, sizeof(uint8_t) * 16);
cos_memcpy(superblock_header.s_volume_name, (char[16]){},
        sizeof(char) * 16);
cos_memcpy(superblock_header.s_last_mounted, (char[64]){},
        sizeof(char) * 64);
```

The volume UUID uniquely identifies this filesystem.
Volume name and last mounted path are initialized as empty strings.

Journaling and Reserved Fields

Many fields related to journaling, error handling, quotas, and encryption are zeroed or initialized to default values because they are either unimplemented or unused in this context.

Examples include:

```
superblock_header.s_journal_inum = 0; // journal inode number, 0
        means no journal yet
superblock_header.s_journal_dev = 0;
superblock_header.s_last_orphan = 0;
cos_memcpy(superblock_header.s_journal_uuid, (uint8_t[16]){},
        sizeof(uint8_t) * 16);
```

64-bit and High Fields

To support very large filesystems, fields like block counts are split into low and high 32-bit parts:

```
superblock_header.s_blocks_count_hi = (uint32_t)(((uint64_t)
        (filesystem_size / DEFAULT_BLOCK_SIZE)) >> 4);
superblock_header.s_free_blocks_count_hi =
        superblock_header.s_blocks_count_hi;
superblock_header.s_min_extra_isize = 32;
superblock_header.s_want_extra_isize = 32;
```

Error Tracking and Mount Options

The superblock contains fields for error counts, timestamps, and function names related to the last errors, which are initialized to zero or empty buffers.
It also has fields for mount options and quota inodes.

Writing to Disk

```
cos_memcpy(&disk_address[*offset], &superblock_header,
        SUPERBLOCK_SIZE);
```

The fully initialized superblock_header is copied into the disk memory at offset 0x400.

Block Descriptor Definition

Writing block descriptors to disk

Purpose of the Block Descriptors

This function initializes and writes the block group descriptors and their associated bitmaps (block bitmap and inode bitmap) to the disk image.
It sets up the data structures necessary for managing block groups in an EXT4-like filesystem and writes these structures directly into the provided disk buffer (disk_address).

```
void write_block_descriptor(char *disk_address, size_t
        filesystem_size, size_t *offset)
```

Variables and Initial Setup

```
block_descriptor_t desc;
block_descriptor_t temp_desc_cpy;
uint8_t block_bitstream[84] = {};
uint8_t bitmap_bitstream[16 + DEFAULT_BLOCK_SIZE] = {};
uint32_t checksum = 0;
```

- **desc**: Temporary structure to hold metadata for a block group descriptor before writing to disk.
- **temp_desc_cpy**: Copy of desc used for checksum calculation.
- **block_bitstream**: Buffer holding data over which the CRC32C checksum for the block descriptor is computed (includes a 16-byte UUID, 4-byte block count, and 64-byte descriptor).
- **bitmap_bitstream**: Buffer holding data over which the CRC32C checksum for block/inode bitmaps is computed (includes 16-byte UUID prefix + bitmap data).
- **checksum**: Variable to hold the computed checksum values.

Calculations of Filesystem Parameters

```
uint32_t number_of_blocks_per_group = 8 * DEFAULT_BLOCK_SIZE;
size_t total_number_of_inodes = filesystem_size /
        DEFAULT_INODE_RATIO;
size_t number_of_block_groups = filesystem_size / (number_of_blocks_per_group
        * DEFAULT_BLOCK_SIZE);
uint32_t number_of_inodes_per_group = total_number_of_inodes /
        number_of_block_groups;
```

```
size_t total_number_of_blocks = number_of_block_groups *
        number_of_blocks_per_group;
size_t number_of_blocks_required_to_store_inodes = (DEFAULT_INODE_SIZE
        * total_number_of_inodes) / DEFAULT_BLOCK_SIZE;

size_t used_blocks = number_of_blocks_required_to_store_inodes + 2
        * number_of_block_groups + 2 + 5;
size_t used_inodes = 11;
size_t count = 2;
```

- **number_of_blocks_per_group**: Defines how many blocks are grouped together in a block group.
  Calculated as 8 times the default block size (in bytes).
- **total_number_of_inodes**: Determines total number of inodes based on a predefined inode-to-bytes ratio (DEFAULT_INODE_RATIO).
- **number_of_block_groups**: Total block groups in the filesystem. Calculated by dividing the total filesystem size by the size of a block group.
- **number_of_inodes_per_group**: How many inodes each block group contains.
- **total_number_of_blocks**: Overall count of blocks in all block groups combined.
- **number_of_blocks_required_to_store_inodes**: Blocks needed to store all inodes, based on inode size and block size.
- **used_blocks**: The count of blocks already reserved or used by metadata and system structures (inodes, block groups descriptors, and others).
- **used_inodes**: Number of reserved inodes by EXT4 filesystem standards (usually 11).
- **count**: Loop counter initialized to 2, used to calculate block locations inside groups.

Setting Initial Offset:

```
*offset = DEFAULT_BLOCK_SIZE;
```

This sets the starting point for writing block descriptors to the disk. It places descriptors at the beginning of the next block as defined by the EXT4 format.

UUID Initialization in Buffers

```
cos_memcpy(&bitmap_bitstream[0], (uint8_t[16])
        {DEFAULT_UUID_VOLUME}, 16);
cos_memcpy(&block_bitstream[0], (uint8_t[16])
        {DEFAULT_UUID_VOLUME}, 16);
```

The first 16 bytes of both bitmap_bitstream and block_bitstream are initialized with a UUID representing the volume.
This UUID is included in checksum computations for consistency and integrity verification.

Writing Block Group Descriptors

```
for (size_t i = 0; i < number_of_block_groups; i +=
        BLOCK_DESCRIPTOR_SIZE, ++count) {
    ...
}
```

For each block group:

Assign block locations

```
desc.bg_block_bitmap_lo = (uint32_t)(count & 0xFFFFFFFF);
desc.bg_inode_bitmap_lo = (uint32_t)((count +
        number_of_block_groups) & 0xFFFFFFFF);
desc.bg_inode_table_lo = (uint32_t)((count + 2 *
        number_of_block_groups) & 0xFFFFFFFF);


desc.bg_block_bitmap_hi = (uint32_t)(count >> 32);
desc.bg_inode_bitmap_hi = (uint32_t)((count +
        number_of_block_groups) >> 32);
desc.bg_inode_table_hi = (uint32_t)((count + 2 *
        number_of_block_groups) >> 32);
```

- **bg_block_bitmap_lo**: Block index of block bitmap for this group.
- **bg_inode_bitmap_lo**: Block index of inode bitmap.
- **bg_inode_table_lo**: Block index of inode table.

Since block pointers can be 64-bit, high 32 bits for block bitmaps, inode bitmaps, and inode tables are set by right-shifting counts.

Calculate free blocks and inodes

```
desc.bg_free_blocks_count_lo = (count == 2) ? (uint16_t)((number_of_blocks_per_gro
        - used_blocks) & 0xFFFF) : (uint16_t)(number_of_blocks_per_group
        & 0xFFFF);
desc.bg_free_inodes_count_lo = (count == 2) ? (uint16_t)((number_of_inodes_per_gro
        - 11) & 0xFFFF) : (uint16_t)(number_of_inodes_per_group &
        0xFFFF);

desc.bg_free_blocks_count_hi = (uint16_t)(number_of_blocks_per_group
        >> 16);
desc.bg_free_inodes_count_hi = (count == 2) ? (uint16_t)((number_of_inodes_per_gro
        - 12) >> 16) : (uint16_t)(number_of_inodes_per_group >>
        16);
```

For the first descriptor (count == 2), calculate the free blocks by subtracting used blocks from blocks per group.
For others, free blocks and inodes are set to total per group since they are assumed unused at this point.

Set other block group metadata

```
desc.bg_used_dirs_count_lo = 0; /*TBD*/
desc.bg_flags = 0;
desc.bg_exclude_bitmap_lo = 0; /*TBD*/
desc.bg_itable_unused_lo = 0;

desc.bg_used_dirs_count_hi =
        0; /*This is to be updated when a new inode is added.*/
desc.bg_itable_unused_hi = (uint16_t)0; /*This value shouldn't be
        set for the same reason.*/
desc.bg_exclude_bitmap_hi = 0;
desc.bg_reserved = 0;
```

Directory count, flags, excluded bitmap, unused inode table count, and reserved fields are initialized (mostly zero).
If an unused count is set, the system treats it as unavailable space instead of free space.

Copy descriptor and prepare checksum

```
temp_desc_cpy = desc;
cos_memcpy(&block_bitstream[16], (uint8_t[4]){(count >> 24) &
        0xFF, (count >> 16) & 0xFF, (count >> 8) & 0xFF, count &
        0xFF}, 4);
cos_memcpy(&block_bitstream[20], &temp_desc_cpy, 64);
desc.bg_checksum = cos_generate_crc32c_checksum(block_bitstream,
        0, 84) & 0xFFFF;
```

The block_bitstream buffer is populated with: - 16-byte UUID (already set), - 4 bytes representing the count in big-endian order, - 64 bytes of the block descriptor structure.

It then computes a 16-bit CRC32C checksum for the descriptor data to ensure integrity.

Compute checksums for block bitmap and inode bitmap

```
cos_memcpy(&bitmap_bitstream[16], &disk_address[count *
        DEFAULT_BLOCK_SIZE], DEFAULT_BLOCK_SIZE);
checksum = cos_generate_crc32c_checksum(bitmap_bitstream, 0,
        DEFAULT_BLOCK_SIZE + 16);
desc.bg_block_bitmap_csum_lo = (checksum & 0xFFFF);
desc.bg_block_bitmap_csum_hi = (checksum >> 16) & 0xFFFF;

cos_memcpy(&bitmap_bitstream[16], &disk_address[(count +
        number_of_block_groups) * DEFAULT_BLOCK_SIZE],
        DEFAULT_BLOCK_SIZE);
checksum = cos_generate_crc32c_checksum(bitmap_bitstream, 0,
        DEFAULT_BLOCK_SIZE + 16);
desc.bg_inode_bitmap_csum_lo = checksum & 0xFFFF;
desc.bg_inode_bitmap_csum_hi = (checksum >> 16) & 0xFFFF;
```

For both bitmaps:

- Copy 16-byte UUID prefix plus bitmap data from the disk buffer into bitmap_bitstream.
- Generate CRC32C checksum over bitmap_bitstream.
- Store checksum's low and high 16 bits in the descriptor.

Write the descriptor to the disk

```
cos_memcpy(&disk_address[*offset], &desc, BLOCK_DESCRIPTOR_SIZE);
*offset += BLOCK_DESCRIPTOR_SIZE;
```

Copy the fully populated descriptor into the disk image at the current offset.
Increment offset by the size of one block descriptor.

Update Offset to Point to Block Bitmap Area

```
*offset = 2 * DEFAULT_BLOCK_SIZE;
```

This sets the offset to start writing block bitmaps, just after the superblock and block group descriptors.

Write Block Bitmap to Disk

```
count = (used_blocks >> 3) + *offset;
for (size_t i = *offset; i < count; ++i)
    disk_address[i] = 0xFF;
```

Calculate count as the byte index where used blocks end.
Fill bytes with 0xFF for reserved inodes.

```
disk_address[(used_blocks >> 3) + *offset] = ~(0xFF << (used_blocks
        % 8));
*offset += number_of_block_groups * DEFAULT_BLOCK_SIZE;
```

For the final byte in the bitmap, this sets bits for used blocks and clears bits for free blocks within the last partially-used byte.
Increment offset to skip all block bitmaps for all groups.

Write Inode Bitmap to Disk:

Similarly, calculate the byte where used inodes end.

```
count = (used_inodes >> 3) + *offset;
for (size_t i = *offset; i < count; ++i)
    disk_address[i] = 0xFF;
```

Calculate count as the byte index where used inodes end.
Fill bytes with 0xFF for reserved inodes.

```
disk_address[(used_inodes >> 3) + *offset] = ~(0xFF << (used_inodes
        % 8));
*offset += number_of_block_groups * DEFAULT_BLOCK_SIZE;
```

Set the last byte to mask used inodes bits only.
Increment offset by size of all inode bitmaps.

Inode Header Definition

Writing the reserved inodes to disk

Purpose of the Inode Headers

This function initializes and writes inodes, including special inodes for directories, and their associated directory entries into the disk image buffer (disk_address).

It sets up file metadata, extent structures for data blocks, and directory entries necessary for filesystem operation, and writes these structures with correct checksums and offsets.

```
void write_inode_to_disk(char *disk_address, size_t
        filesystem_size, size_t *offset, uint64_t unix_timestamp)
```

Variables and Buffers

Inode and extent structures

```
inode_t inode;
extent_head_t head;
extent_leaf_t leaf;
dirent2_t dir_entry;
dirent2_tail_t tail;
```

- **inode**: Represents the inode structure for a file or directory.
- **head**: Extent tree header, describing the extent structure used for mapping file blocks.
- **leaf**: Leaf node in extent tree, representing a contiguous block range on disk.
- **dir_entry**: Directory entry structure, representing files or directories inside a directory.
- **tail**: Tail structure of directory entries used for integrity checks.

Buffers for checksum calculation

```
uint8_t checksum_bitstream[280] = {};
uint8_t dir_bitstream[16 + DEFAULT_BLOCK_SIZE] = {};
uint32_t checksum;
```

- **checksum_bitstream**: Buffer used to generate inode checksum, which includes UUID, inode index, generation, and inode data.
- **dir_bitstream**: Buffer used for directory block checksum, including UUID prefix + directory block data.
- **checksum**: Stores the computed CRC32C checksums.

Calculated Filesystem Layout Parameters

```
uint32_t number_of_blocks_per_group = 8 * DEFAULT_BLOCK_SIZE;
size_t total_number_of_inodes = filesystem_size /
        DEFAULT_INODE_RATIO;
size_t number_of_block_groups = filesystem_size / (number_of_blocks_per_group
        * DEFAULT_BLOCK_SIZE);
```

```
size_t number_of_blocks_required_to_store_inodes = (DEFAULT_INODE_SIZE
        * total_number_of_inodes) / DEFAULT_BLOCK_SIZE;

size_t used_blocks = number_of_blocks_required_to_store_inodes + 2
        * number_of_block_groups + 2;
size_t number_sector = 0;
```

- **number_of_blocks_per_group**: Number of blocks in one block group.
- **total_number_of_inodes**: Total inodes in the filesystem, calculated by dividing filesystem size by inode ratio.
- **number_of_block_groups**: Total block groups in the filesystem.
- **number_of_blocks_required_to_store_inodes**: Number of blocks needed to store all inodes.
- **used_blocks**: Number of blocks reserved for metadata (inode storage, block groups, superblock, etc.).
- **number_sector**: Number of 512-byte sectors required for the file described by the inode (calculated later).

Extent Header Initialization

```
head.eh_magic = EXTENT_MAGIC;
head.eh_entries = 1;
head.eh_max = 4;
head.eh_depth = 0;   // Only leaf nodes for this proof of concept
        (PoC)
head.eh_generation = 0;
```

Sets up the extent tree header with magic number, number of entries, max entries, tree depth (flat leaf nodes only), and generation.

Initialize UUID Prefix for Checksums:

```
cos_memcpy(&checksum_bitstream[0], (uint8_t[16])
        {DEFAULT_UUID_VOLUME}, 16);
cos_memcpy(&dir_bitstream[0], (uint8_t[16]){DEFAULT_UUID_VOLUME},
        16);
```

Copies the 16-byte volume UUID to the start of both checksum buffers for checksum integrity.

Write Reserved Inodes

```
for (size_t i = 1; i < 12; ++i) {
    ...
}
```

Writes 11 inodes, where:

- Inodes 2 and 11 are special directory inodes.
- Other reserved inodes are initialized empty.

Root Directory Inode (i == 2) and Lost+Found Directory Inode (i == 11)

```
inode.i_mode = 0x4000 | 0x800 | 0x400 | 0x100 | 0x80 | 0x40 | 0x20
        | 0x8 | 0x4 | 0x1;
inode.i_flags = 0x80000;
inode.i_links_count = 2;
```

Set mode: Marks inode as directory with read/write/execute permissions for user/group/others.
i_flags set to 0x80000 (immutable or other EXT4 flag).
Links count starts at 2 (for “.” and “..”).

```
if (i == 2) {
    ++inode.i_links_count;
    leaf.ee_block = 0;
    leaf.ee_len = 1;
    leaf.ee_start_lo = used_blocks & 0xFFFFFFFF;
    leaf.ee_start_hi = (used_blocks >> 32) & 0xFFFF;
} else {
    leaf.ee_block = 0;
    leaf.ee_len = 4;
    leaf.ee_start_lo = (used_blocks + 1) & 0xFFFFFFFF;
    leaf.ee_start_hi = ((used_blocks + 1) >> 32) & 0xFFFF;
}
```

Increment links count by one more (because inode 11 will be referenced).

- **ee_block**: Logical block number within file (start).
- **ee_len**: Length in blocks (1 block for inode 2, 4 blocks for inode 11).
- **ee_start_lo** and **ee_start_hi**: Starting physical block on disk, split into lower and upper bits, set to used_blocks and used_blocks + 1 respectively.

```
number_sector = (leaf.ee_len * 4096) / 512;
```

Converts extent length in 4KB blocks to 512-byte sectors.

```
inode.i_blocks_lo = (uint32_t)(number_sector & 0xFFFFFFFF);
inode.osd2.linux2.l_i_blocks_high = (uint16_t)((number_sector >>
        32) & 0xFFFF);
cos_memcpy(&inode.i_block[0], &head, sizeof(uint32_t) * 3);
cos_memcpy(&inode.i_block[3], &leaf, sizeof(uint32_t) * 3);
```

These fields represent the number of 512 byte blocks used by the file.
i_blocks_lo and **l_i_blocks_high** store total sectors used.
i_block array stores extent header and leaf structures (each 3 x 32-bit words).

```
inode.i_size_lo = (uint32_t)((512 * number_sector) & 0xFFFFFFFF);
inode.i_size_high = (uint32_t)(((512 * number_sector) >> 32) &
        0xFFFFFFFF);
```

Set file size (low and high 32 bits).
Size is number_sector * 512 bytes.

```
inode.i_extra_isize = 32;
inode.i_generation = head.eh_generation;
```

Set extra inode fields.

Other Reserved Inodes (non-directory)

```
inode.i_mode = 0;
inode.i_flags = 0;
inode.i_links_count = 0;

inode.i_blocks_lo = 0;
inode.osd2.linux2.l_i_blocks_high = 0;
cos_memcpy(inode.i_block, (uint32_t[15]){}, sizeof(uint32_t) *
        15);

inode.i_size_lo = 0;
inode.i_size_high = 0;

inode.i_extra_isize = 0;
inode.i_generation = 0;
```

All fields are zeroed to mark unused or empty inodes.

Set Common Inode Fields (All Inodes)

```
inode.i_gid = (uint16_t)0;
inode.osd2.linux2.l_i_gid_high = (uint16_t)0;
inode.i_uid = (uint16_t)0;
inode.osd2.linux2.l_i_uid_high = (uint16_t)0;
```

Group ID and User ID fields set to 0 (with high 16 bits).

```
inode.i_atime = unix_timestamp;
inode.i_ctime = unix_timestamp;
inode.i_mtime = unix_timestamp;
inode.i_crtime = unix_timestamp;
```

Set timestamps (i_atime, i_ctime, i_mtime, i_crtime) to unix_timestamp (time of creation/modification/access).

```
inode.i_ctime_extra = 0;
inode.i_mtime_extra = 0;
inode.i_atime_extra = 0;
inode.i_crtime_extra = 0;
```

Sub-second timestamps set to 0.

```
inode.i_dtime = 0;
```

Deletion time is 0.

```
inode.i_file_acl_lo = (uint32_t) 0;
inode.osd2.linux2.l_i_file_acl_high = (uint16_t) 0;

inode.osd2.linux2.l_i_reserved = 0;
inode.i_obso_faddr = 0;
inode.osd1.linux1.l_i_version = (uint32_t)0;
inode.i_version_hi = (uint32_t)0;
inode.i_projid = 0;
cos_memcpy(inode.i_cos_reserved, (uint8_t[96]){}, sizeof(uint8_t)
        * 96);
```

File ACL, reserved fields, version, project ID, and reserved bytes zeroed.

Calculate and Write Checksums

```
cos_memcpy(&checksum_bitstream[16], (uint8_t[4]){(i >> 24) & 0xFF,
        (i >> 16) & 0xFF, (i >> 8) & 0xFF, i & 0xFF}, 4);
cos_memcpy(&checksum_bitstream[20], (uint8_t[4]){(inode.i_generation
        >> 24) & 0xFF, (inode.i_generation >> 16) & 0xFF, (inode.i_generation
        >> 8) & 0xFF, inode.i_generation & 0xFF}, 4);
cos_memcpy(&checksum_bitstream[24], &disk_address[*offset],
        DEFAULT_INODE_SIZE);
checksum = cos_generate_crc32c_checksum(checksum_bitstream, 0,
        280);
inode.osd2.linux2.l_i_checksum_lo = (uint16_t)(checksum & 0xFFFF);
inode.i_checksum_hi = (uint16_t)((checksum >> 16) & 0xFFFF);
```

Prepare checksum_bitstream for CRC32C:

- Bytes 0-15: Volume UUID (already copied).
- Bytes 16-19: Inode number (i).
- Bytes 20-23: Inode generation.
- Bytes 24-...: Current inode data copied from disk buffer.

Compute checksum over 280 bytes.
Store checksum low and high parts into inode structure **l_i_checksum_lo** and **i_checksum_hi** fields.

```
cos_memcpy(&disk_address[*offset], &inode, DEFAULT_INODE_SIZE);
*offset += DEFAULT_INODE_SIZE;
```

Copy fully prepared inode structure to disk at the current offset.
Increment offset by the size of one inode.

Setup Directory Entries

```
tail.det_rec_len = 12;
tail.det_reserved_zero = 0;
tail.det_reserved_zero2 = 0;
tail.det_reserved_ft = 0xDE;
```

Initialize tail structure at end of directory blocks for checksumming and integrity.

Create directory entries (dirent2_t)

```
dir_entry.inode = 2;
dir_entry.rec_len = 12;
dir_entry.name_len = 1;
dir_entry.file_type = 2;
```

```
cos_memcpy(dir_entry.name, ".", 1);
cos_memcpy(&disk_address[used_blocks * 4096], &dir_entry, 12);
```

Entry for inode 2 with name “.”, pointing to self.

```
dir_entry.inode = 11;
cos_memcpy(&disk_address[(used_blocks + 1) * 4096], &dir_entry,
        12);
```

Entry for inode 11 with name “.”, pointing to self.

```
dir_entry.inode = 2;
dir_entry.rec_len = 12;
dir_entry.name_len = 2;
dir_entry.file_type = 2;
cos_memcpy(dir_entry.name, "..", 2);
cos_memcpy(&disk_address[used_blocks * 4096 + 12], &dir_entry,
        12);
```

Entry for inode 11 with name “..”, pointing to parent.

```
dir_entry.inode = 11;
dir_entry.rec_len = DEFAULT_BLOCK_SIZE - 36;
dir_entry.name_len = 10;
dir_entry.file_type = 2;
cos_memcpy(dir_entry.name, "lost+found", 10);
cos_memcpy(&disk_address[used_blocks * 4096 + 24], &dir_entry,
        20);
```

Entry for “lost+found” directory.

Write these entries into directory blocks located at block indexes used_blocks and used_blocks + 1.

```
cos_memcpy(&dir_bitstream[16], &disk_address[used_blocks * 4096],
        DEFAULT_BLOCK_SIZE);
tail.det_checksum = cos_generate_crc32c_checksum(dir_bitstream, 0,
        DEFAULT_BLOCK_SIZE + 16);
cos_memcpy(&disk_address[used_blocks * 4096 + 4084], &tail, 12);

cos_memcpy(&dir_bitstream[16], &disk_address[(used_blocks + 1) *
        4096], DEFAULT_BLOCK_SIZE);
tail.det_checksum = cos_generate_crc32c_checksum(dir_bitstream, 0,
        DEFAULT_BLOCK_SIZE + 16);
cos_memcpy(&disk_address[(used_blocks + 1) * 4096 + 4084], &tail,
        12);
```

Copy directory block with UUID prefix to dir_bitstream. Generate CRC32C checksum and store in tail.det_checksum. Write tail structure back at the end of directory blocks.

```
dir_entry.inode = 0;
dir_entry.rec_len = DEFAULT_BLOCK_SIZE;
dir_entry.name_len = 0;
dir_entry.file_type = 0;
memset(dir_entry.name, 0, 255);
```

Create a special “null” directory entry with zero inode and full block length.

```
cos_memcpy(&disk_address[(used_blocks + 2) * 4096], &dir_entry,
        sizeof(dirent2_t));
cos_memcpy(&disk_address[(used_blocks + 3) * 4096], &dir_entry,
        sizeof(dirent2_t));
cos_memcpy(&disk_address[(used_blocks + 4) * 4096], &dir_entry,
        sizeof(dirent2_t));
```

Write this entry to directory blocks starting at used_blocks + 2, used_blocks + 3, and used_blocks + 4.
Ext4 filesystem mount requires ALL yet to be used directory blocks to start with a dirent structure that is NULL except for the rec_len field which MUST be equal to block size.
This is required by EXT4 to mark unused directory blocks properly.

Compiler Issues

```
#define STACK_CHK_GUARD 0xa5f3cc8d

uintptr_t __stack_chk_guard = STACK_CHK_GUARD;

__attribute((noreturn)) void __stack_chk_fail(void)
{
    while(1);
    //_exit();
}
```

This macro defines a stack guard value as a fixed hexadecimal constant: **0xa5f3cc8d**.
The stack guard is a canary value used for stack smashing protection. Its purpose is to detect if a buffer overflow or stack corruption has occurred.

This declares a global variable named **__stack_chk_guard** initialized with the defined guard value.
The uintptr_t type is an unsigned integer type guaranteed to be able to hold a pointer (usually 32 or 64 bits depending on platform).
This guard value is placed on the stack by the compiler-generated stack

protection code.
When a function returns, the guard value is checked to ensure it hasn't been altered by a buffer overflow attack.

This declares **the function __stack_chk_fail** with the GCC **noreturn** attribute, telling the compiler this function will never return.
This function is called when a stack smashing attack is detected, i.e., when the guard value has been corrupted.

An infinite loop to halt program execution when a stack corruption is detected. This effectively stops the program to avoid further damage or exploitation.

Proper implementation of the **stack_chk_fail** function shows that it should use the **_exit()** call.
This isn't yet done.

External Documentation:

- Linux Ext4 Definition
- Ext4 Disk Layout Definition 1
- Ext4 Disk Layout Definition 2