EXT4 Filesystem Access Wrappers Analysis

This source file defines core helper functions and wrappers for reading and writing ext4 filesystem data structures.
The functions interact directly with low-level ext4 structures such as the superblock, block group descriptor, and inode metadata.

The overall workflow involves reading raw blocks from disk into temporary buffers, interpreting them as ext4 metadata structures, and then delegating data traversal or mutation to further specialized functions (read_inode_tree).

Superblock Reading

```c
static void read_superblock(superblock_t *sb)
{
    uint16_t buffer[512];

    read_byte(&buffer[0], 2, 1);
    read_byte(&buffer[256], 3, 1);
    cos_memcpy(sb, buffer, sizeof(superblock_t));
}
```

The superblock is read across two disk logical block addresses (LBAs): LBA 2 and LBA 3.
Each read_byte call reads 512 bytes (1 block of 512 16-bit units). Data is collected into a 1024-byte buffer (buffer is an array of 512 uint16_t units = 1024 bytes).

The superblock structure superblock_t (usually 1024 bytes in ext4) is copied from this buffer.
As 48bit LBA mode is used via the read_byte utilities, the use of 16-bit units is required for the static buffer.

Block Group Descriptor Table Reading

```
static void read_block_descriptor_table(block_descriptor_t
        *block_table, uint64_t inode_block_nb)
{
    uint16_t buffer[256];
    uint8_t lba = 8 + (uint8_t)(inode_block_nb >> 3); // Divided
        by 8

    read_byte(buffer, lba, 1);
    cos_memcpy(block_table, &buffer[(inode_block_nb & 0b111) <<
        5], sizeof(block_descriptor_t)); // Modulus 8 * 32
}
```

Block group descriptors are stored starting at LBA 8.
inode_block_nb >> 3 (divided by 8) selects which LBA block holds the target block group descriptor.

Each block contains 8 block descriptors (8 * 32 bytes = 256 bytes), hence indexing with (inode_block_nb & 0b111) << 5 (multiply by 32 bytes).
read_byte reads one block (512 bytes).
The correct block descriptor is copied to block_table.

Inode Metadata Reading

```
static void read_inode_metadata(uint64_t inode_offset,
        block_descriptor_t *block_group, inode_t *inode_metadata)
{
    uint64_t inode_table_lba = ((block_group->bg_inode_table_hi +
        block_group->bg_inode_table_lo) << 3) + (inode_offset >>
        1);
    uint16_t buffer[256] = {};

    read_byte(buffer, inode_table_lba, 1);
    cos_memcpy(inode_metadata, &buffer[(inode_offset & 0b1) << 7],
        sizeof(inode_t)); // Modulus 2 * 128
}
```

The inode table’s physical address is split across high and low 32-bit fields; they are combined and multiplied by 8 (<< 3) to get the base LBA.
Inode offset is divided by 2 (shifted right by 1) to find the LBA block within the inode table.

Each block holds 2 inodes (since each inode is 256 bytes, and a block is 512 bytes).
The correct inode metadata is copied using (inode_offset & 0b1) << 7 (0 or 128 bytes offset inside block).

Inode Block Group and Table Offset Calculation

```
static uint64_t get_inode_block_group_number(superblock_t *sb,
        uint64_t inode_nb)
{
    return (uint64_t)((inode_nb - 1) / sb->s_inodes_per_group);
}

static uint64_t get_inode_table_offset(superblock_t *sb, uint64_t
        inode_nb)
{
    return (uint64_t)((inode_nb - 1) % sb->s_inodes_per_group);
}
```

These functions calculate the block group number and the inode offset inside the group given a global inode number.
Subtract 1 because ext4 inode numbering starts at 1.
Integer division and modulo are used for indexing into groups.

Read Wrapper

```
void read_wrapper(uint64_t inode_nb, uint8_t *content, size_t len)
{
    superblock_t sb;
    block_descriptor_t block_group;
    inode_t metadata;
    tree_t tree;
    uint64_t inode_block;
    uint64_t inode_offset;

    read_superblock(&sb);
    inode_block = get_inode_block_group_number(&sb, inode_nb);
    inode_offset = get_inode_table_offset(&sb, inode_nb);
    read_block_descriptor_table(&block_group, inode_block);
    read_inode_metadata(inode_offset, &block_group, &metadata);
    tree = *(struct ext_tree *)metadata.i_block;
    read_inode_tree(&tree, &content, &len, READ_MODE);
}
```

High-level function to read content linked to an inode.

Steps:

- Read superblock.
- Calculate inode block group and offset.
- Read block group descriptor.
- Read inode metadata.
- Interpret metadata.i_block as a tree structure (ext4 extent tree).
- Call read_inode_tree to read the actual content from the extent tree.

Write Wrapper

```
void write_wrapper(uint64_t inode_nb, uint8_t *content, size_t
        len)
{
    uint8_t *content_ptr = content;
    superblock_t sb;
    block_descriptor_t block_group;
    inode_t metadata;
    tree_t tree;
    uint64_t inode_block;
    uint64_t inode_offset;

    read_superblock(&sb);
    inode_block = get_inode_block_group_number(&sb, inode_nb);
    inode_offset = get_inode_table_offset(&sb, inode_nb);
    read_block_descriptor_table(&block_group, inode_block);
    read_inode_metadata(inode_offset, &block_group, &metadata);
    tree = *(struct ext_tree *)metadata.i_block;
    read_inode_tree(&tree, &content_ptr, &len, WRITE_MODE);
}
```

Mirrors the read_wrapper but performs writing.
Calls read_inode_tree with WRITE_MODE flag and a pointer to content pointer for mutability.

Casting metadata.i_block to ext_tree assumes the inode uses extent trees rather than block pointers.
The block size is implicitly 512 bytes given read_byte usage.

Bitwise operations used for indexing are efficient for power-of-two multiples.
Error handling is missing here; assumes valid inputs and successful reads.
read_inode_tree is a key function for traversing and reading/writing actual file data based on extents.

Extent Tree Reading

```
static void read_block_content(uint64_t block_addr, uint64_t
        block_len, size_t *len, uint8_t **data)
{
    uint16_t block_content[2048] = {};
    uint16_t i;

    for (i = 0; i < block_len && 4096 <= *len; ++i) {
        read_byte(block_content, block_addr << 3, 8);
        cos_memcpy(*data, (uint8_t *)block_content, 4096);
        *data += 4096;
        ++block_addr;
    }
    if (*len < 4096 && i < block_len) {
        read_byte(block_content, block_addr << 3, 8);
        cos_memcpy(*data, (uint8_t *)block_content, *len);
        *len = 0;
    }
}
```

Reads raw block data from disk into user memory buffer.
block_addr << 3 multiplies block address by 8 to convert to logical block address (LBA).

read_byte(block_content, block_addr << 3, 8) reads 8 LBAs (8 × 512 bytes = 4096 bytes, i.e., 4 KiB block).
Copies 4096 bytes to the user buffer pointed to by *data.
Advances the user buffer pointer and block address.

Stops when either the requested length *len is exhausted or all requested blocks are read.
Handles partial reads if less than 4096 bytes remain.

```
static void write_block_content(uint64_t block_addr, uint64_t
        block_len, size_t *len, uint8_t **data)
{
    uint16_t block_content[2048] = {};
```

```c
    uint16_t i;

    for (i = 0; i < block_len && 4096 <= *len; ++i) {
        cos_memcpy((uint8_t *)block_content, *data, 4096);
        write_byte(block_content, block_addr << 3, 8);
        *data += 4096;
        ++block_addr;
    }
    if (*len < 4096 && i < block_len) {
        cos_memcpy((uint8_t *)block_content, *data, *len);
        write_byte(block_content, block_addr << 3, 8);
        *len = 0;
    }
}
```

Symmetric counterpart of read_block_content.
Copies data from the user buffer to a temporary buffer (block_content).

Writes 4 KiB chunks (8 LBAs × 512 bytes).
Handles partial writes if less than 4096 bytes remain.
Advances user buffer and block address similarly.

```c
void read_inode_tree(tree_t *tree, uint8_t **content, size_t *len,
        uint8_t mode)
{
    uint64_t block_addr = 0;

    if (*len <= 0)
        return;
    if (tree->ext_head.eh_magic != EXTENT_MAGIC)
        return;
    if (tree->ext_head.eh_entries > tree->ext_head.eh_max)
        return;

    if (!tree->ext_head.eh_depth) {
        // Leaf node: direct extents
        switch (tree->ext_head.eh_entries) {
            case 4:
                block_addr =
        ((uint64_t)tree->node4.ext_leaf_fourth.ee_start_hi << 32)
        + tree->node4.ext_leaf_fourth.ee_start_lo;
                mode ? read_block_content(block_addr,
        tree->node4.ext_leaf_fourth.ee_len, len, content) :
        write_block_content(block_addr,
        tree->node4.ext_leaf_fourth.ee_len, len, content);
                if (*len <= 0) return;
                /* fall through */
            case 3:
                block_addr =
        ((uint64_t)tree->node3.ext_leaf_third.ee_start_hi << 32) +
        tree->node3.ext_leaf_third.ee_start_lo;
                mode ? read_block_content(block_addr,
        tree->node3.ext_leaf_third.ee_len, len, content) :
        write_block_content(block_addr,
        tree->node3.ext_leaf_third.ee_len, len, content);
                if (*len <= 0) return;
                /* fall through */
            case 2:
                block_addr =
        ((uint64_t)tree->node2.ext_leaf_second.ee_start_hi << 32)
        + tree->node2.ext_leaf_second.ee_start_lo;
                mode ? read_block_content(block_addr,
        tree->node2.ext_leaf_second.ee_len, len, content) :
        write_block_content(block_addr,
        tree->node2.ext_leaf_second.ee_len, len, content);
                if (*len <= 0) return;
                /* fall through */
            case 1:
                block_addr =
        ((uint64_t)tree->node1.ext_leaf_first.ee_start_hi << 32) +
        tree->node1.ext_leaf_first.ee_start_lo;
                mode ? read_block_content(block_addr,
        tree->node1.ext_leaf_first.ee_len, len, content) :
        write_block_content(block_addr,
        tree->node1.ext_leaf_first.ee_len, len, content);
                break;
            case 0:
                return;
            default:
                return;
        }
    } else {
        // Internal node: recurse through extent index nodes
        switch (tree->ext_head.eh_entries) {
            case 4:
                block_addr =
        ((uint64_t)tree->node4.ext_node_fourth.ei_leaf_hi << 32) +
        tree->node4.ext_node_fourth.ei_leaf_lo;
                read_inode_tree(tree, content, len, mode);
                if (*len <= 0) return;
                /* fall through */
            case 3:
                block_addr =
        ((uint64_t)tree->node3.ext_node_third.ei_leaf_hi << 32) +
        tree->node3.ext_node_third.ei_leaf_lo;
                read_inode_tree(tree, content, len, mode);
                if (*len <= 0) return;
                /* fall through */
            case 2:
                block_addr =
        ((uint64_t)tree->node2.ext_node_second.ei_leaf_hi << 32) +
        tree->node2.ext_node_second.ei_leaf_lo;
                read_inode_tree(tree, content, len, mode);
                if (*len <= 0) return;
                /* fall through */
            case 1:
                block_addr =
        ((uint64_t)tree->node1.ext_node_first.ei_leaf_hi << 32) +
        tree->node1.ext_node_first.ei_leaf_lo;
                read_inode_tree(tree, content, len, mode);
                break;
            case 0:
                return;
            default:
                return;
        }
    }
    return;
}
```

This function traverses the ext4 extent tree inside an inode to read or write file data.

The extent tree can have multiple levels:

- eh_depth == 0 means the current node is a leaf node containing direct extents.
- eh_depth > 0 means internal nodes with pointers to other extent nodes.

It checks:

- That the extent header magic number matches (EXTENT_MAGIC).
- That the number of entries (eh_entries) is valid (not more than max).

Depending on eh_entries, it processes 1 to 4 extent entries.

For leaf nodes:

- Extracts physical block address and length from each extent entry.
- Calls read_block_content or write_block_content depending on mode.

Stops if requested length is fulfilled.

For internal nodes, it recursively calls itself on child nodes pointed by extent index entries.

Note:
The recursive calls to read_inode_tree in internal nodes are currently passing the same tree pointer instead of the subtree at the given block address.
This is due to incomplete implementation of the extent tree.

mode is a flag such that: - 0 = read - non-zero = write.

CMOS RTC IRQ Handler and BCD to Binary Conversion

BCD to Binary Conversion Function

The function bcd_to_binary converts a Binary-Coded Decimal (BCD) value into its binary equivalent.
BCD encoding stores each decimal digit in a 4-bit nibble, meaning the higher nibble holds the tens digit and the lower nibble holds the units digit.

The formula used:

```
return ((value & 0xF0) >> 1) + ((value & 0xF0) >> 3) + (value &
        0xF);
```

breaks down as follows:

- (value & 0xF0) >> 1 extracts the high nibble and shifts it right by 1 (equivalent to multiplying the tens digit by 8).
- (value & 0xF0) >> 3 extracts the high nibble and shifts it right by 3 (equivalent to multiplying the tens digit by 2).

Adding these two yields the tens digit multiplied by 10 (8 + 2).

(value & 0xF) extracts the low nibble, representing the units digit.
Together, the sum reconstructs the decimal value from its BCD encoding.

CMOS IRQ Handler (cos_irq_cmos_handler)

This interrupt handler reads and processes time data from the Real-Time Clock (RTC) via CMOS registers.

Overview and Constants

- The Epoch for timestamp calculations is fixed at January 1, 1970.
- Some CMOS registers, such as the weekday (0x06) and century (0x32), may be unreliable or absent depending on hardware age.
- The month_table holds the cumulative number of seconds up to the start of each month (non-leap years assumed).
- Variables (seconds, minutes, hours, month_day, month, year, leap_days) track the respective time components.
- leap_days starts at 8, accounting for leap years up to the start of the 21st century.

RTC Settings Reading

Register 0x8B is read from CMOS to obtain the RTC mode settings.
Bits in rtc_settings indicate if:

- 24-hour or 12-hour mode is active (0x10).
- Binary or BCD mode is active (0x100).

Time Values Retrieval

The CMOS RTC time values are read from specific CMOS registers:

CMOS Register	Time Value
0x80	Seconds
0x82	Minutes
0x84	Hours
0x87	Day of Month
0x88	Month
0x89	Year (last two digits)

For each:

- The value is read from CMOS port 0x71 after writing the register address to port 0x70.

- If RTC is in BCD mode (default), the value is converted to binary using bcd_to_binary.
- If in binary mode, the value is taken directly.

Hours Format Handling

If 12-hour format is enabled (bit 0x10 clear), the code converts to 24-hour format by: - Checking if the high bit (0x80) is set, indicating PM. - Adding 12 hours and wrapping modulo 24.

Leap Year and Month Adjustment

Leap days are adjusted based on the month and year:

- If month ≥ March and the year is divisible by 4 (checked by (year & 0x11)), leap days are incremented accordingly.
- The month is converted into seconds using month_table.
- If month is January, it maps to 0 seconds.

Unix Timestamp Calculation

The unix_timestamp is computed as:

```
unix_timestamp = (year + 30) * 31,536,000 + month_seconds + (leap_days
        + month_day) * 86,400 + hours * 3,600 + minutes * 60 +
        seconds
```

(year + 30) * 31,536,000 converts years since 1970 into seconds (assuming non-leap years).
month_seconds adds elapsed seconds within the current year.
(leap_days + month_day) * 86,400 adds leap days and day of month in seconds.

Remaining components convert hours, minutes, and seconds into total seconds.

Interrupt Acknowledgment

Reading CMOS Register 0x8C (RTC Register C) acknowledges the RTC interrupt, enabling further IRQ8 triggers.
The End-of-Interrupt (EOI) commands are sent to the PICs (Programmable Interrupt Controllers) at ports 0x20 (master) and 0xA0 (slave) to signal that the interrupt has been handled.