

Αναφορά Παράδοσης της Δεύτερης Εργαστηριακής Άσκησης

Στο Μάθημα των Λειτουργικών Συστημάτων

Ονοματεπώνυμο: Στεργίου Βασίλειος
ΑΜ: 4300

Περιγραφή του στόχου της εργαστηριακής άσκησης

Στη παρούσα εργαστηριακή άσκηση, ζητείται η υλοποίηση ενός **αρχείου καταγραφής** αλλαγών που πραγματοποιούνται στις εσωτερικές δομές του συστήματος αρχείων FAT, διθέντος του κώδικα του Linux με τη **μορφή βιβλιοθήκης (LKL)**. Ο σκοπός της παρούσας εργαστηριακής άσκησης επικεντρώνεται στην ανάκτηση των εγγραφών και των αλλαγών που πραγματοποιήθηκαν σε ορισμένα αρχεία ανά πάσα χρονική στιγμή, μέσω του αρχείου καταγραφής (Journal), σε περίπτωση κάποιας αστοχίας ή κατάρρευσης κατά τη διάρκεια της λειτουργίας του συστήματος.

Οι βασικές δομές των οποίων οι πληροφορίες έχουν ζωτική σημασία για το σύστημα αρχείων είναι οι εξής:

- Το **directory entry** ή **dentry**, το οποίο πρόκειται για το μονοπάτι (path) όπου είναι αποθηκευμένο το αρχείο.
- Το **inode**, το οποίο πρόκειται για δεδομένα που σχετίζονται άμεσα με το αρχείο. Ενδεικτικά, μερικά από τα πεδία είναι τα εξής:
 1. άδειες πρόσβασης,
 2. μέγεθος αρχείου
 3. αναγνωριστικό ιδιοκτήτη
 4. χρόνος δημιουργίας

Στο κώδικα της βιβλιοθήκης Linux Kernel Library (LKL) έχουν εντοπιστεί και περαιτέρω πεδία, τα οποία έχουν καταγραφεί και στα οποία θα γίνει αναφορά κατά την εξήγηση του κώδικα που έχει προσθεθεί.

- Το **superblock**, το οποίο αποτελεί τη δομή όπου αποθηκεύονται το inode του εκάστοτε αρχείου.
- Το **File Allocation Table (FAT)**, το οποίο είναι οργανωμένο σε clusters των **12, 16 ή 32 bits**, ανάλογα με τον αντίστοιχο τύπο του συστήματος αρχείων FAT που χρησιμοποιείται κάθε φορά.

Στο κώδικα των δομών αυτών, έχουν χρησιμοποιηθεί κατάλληλα οι κλήσεις συστήματος **sys_open()**, **sys_write()**, **sys_fsync()**, **sys_fdatasync()**, **sys_close()**, καθώς και η κλήση συστήματος **printk()** για την παρουσίαση της σειράς των κλήσεων που πραγματοποιούνται από τη βιβλιοθήκη, έως ότου να ολοκληρωθεί η μεταφορά των αρχείων στο σύστημα αρχείων FAT, αλλά και να γίνει αντιληπτή η αλληλεπίδραση των δομών μεταξύ τους.

Για τιν ορθή χρήση των κλήσεων αυτών στα αρχεία της βιβλιοθήκης LKL, έχουν οριστεί οι εξής μεταβλητές ως **extern** στο header file **fat.h**, όπως επίσης και το macro **FILE_BUFFER_SIZE**.

```
59 // The minimum buffer size
60 // of data that will be written into the journal
61 #define FILE_BUFFER_SIZE 128
62
63 extern int journal;
64 extern int file_opened;
65 extern char file_buf[FILE_BUFFER_SIZE];
```

- Το macro **FILE_BUFFER_SIZE** προσδιορίζει το **μέγιστο μέγεθος** της εγγραφής που γίνεται κάθε φορά στο journal. Η εγγραφή αυτή αποθηκεύεται στη μεταβλητή **file_buf** μέσω της κλήσης **sprintf()** και στη συνέχεια στη συνέχεια αξιοποιείται από τη κλήση **sys_write()** για τη μεταφορά της τιμής του buffer στο journal
- Η μεταβλητή **journal**, η οποία προσδιορίζει τον **file descriptor** του αρχείου καταγραφής
- Η μεταβλητή **file_opened**, στην οποία αποθηκεύεται η τιμή που επιστρέφεται από τη κλήση **sys_open()**, ώστε να ελέγχεται το αρχείο ως προς το ορθό άνοιγμα του

Σε εικόνες που έχουν προσθεθεί παρακάτω με τα μηνύματα προς το χρήστη με τις τιμές των πεδίων που εγγράφονται στο journal, έχει ακολουθηθεί η εξής τακτική για το έλεγχο των τιμών των πεδίων:

- Χρησιμοποιώντας το τελεστή “**>**” στο τερματικό για την εκτέλεση της εφαρμογής **cptofs**, το standard output μεταφέρεται σε ένα αρχείο με όνομα που επιλέγει ο χρήστης (ενδεικτικά **results.txt**) και έχει τη δυνατότητα να δει όλα τα μηνύματα των κλήσεων **printk()** στο αρχείο αυτό

Συλλογιστική πορεία για την υλοποίηση της εργαστηριακής άσκησης και τροποποιημένα αρχεία

Στην εκφώνηση της εργαστηριακής άσκησης επισημαίνονται τα αρχεία που αφορούν τις δομές στις οποίες έγινε αναφορά παραπάνω και συγκεκριμένα στα structs που περιέχουν λειτουργίες σχετιζόμενες με τις δομές αυτές. Στον παρακάτω πίνακα αναφέρονται τα structs στα οποία επικεντρώθηκε τον ενδιαφέρον ως προς τη μελέτη του κώδικα τους, καθώς και εικόνες από τον κώδικα των πεδίων για καθένα από αυτά.

Δομή (Struct)	Αρχείο
struct super_operations fat_sops	fs/fat/inode.c
struct address_space_operations fat_aops	fs/fat/inode.c
struct fatent_operations fat12/16/32_ops	fs/fat/fatent.c
struct file_operations fat_file_operations	fs/fat/file.c
struct inode_operations msdos_dir_inode_operations	fs/fat/namei_msdos.c
struct inode_operations vfat_dir_inode_operations	fs/fat/namei_vfat.c

```

static const struct super_operations fat_sops = {
    .alloc_inode    = fat_alloc_inode,
    .destroy_inode = fat_destroy_inode,
    .write_inode   = fat_write_inode,
    .evict_inode   = fat_evict_inode,
    .put_super     = fat_put_super,
    .statfs        = fat_statfs,
    .remount_fs   = fat_remount,
    .show_options  = fat_show_options,
};

static const struct address_space_operations fat_aops = {
    .readpage      = fat_readpage,
    .readpages     = fat_readpages,
    .writepage     = fat_writepage,
    .writepages    = fat_writepages,
    .write_begin   = fat_write_begin,
    .write_end     = fat_write_end,
    .direct_IO     = fat_direct_IO,
    .bmap          = _fat_bmap
};

static const struct fatent_operations fat12_ops = {
    .ent_blocknr   = fat12_ent_blocknr,
    .ent_set_ptr   = fat12_ent_set_ptr,
    .ent_bread     = fat12_ent_bread,
    .ent_get       = fat12_ent_get,
    .ent_put       = fat12_ent_put,
    .ent_next      = fat12_ent_next,
};

static const struct fatent_operations fat16_ops = {
    .ent_blocknr   = fat_ent_blocknr,
    .ent_set_ptr   = fat16_ent_set_ptr,
    .ent_bread     = fat_ent_bread,
    .ent_get       = fat16_ent_get,
    .ent_put       = fat16_ent_put,
    .ent_next      = fat16_ent_next,
};

static const struct fatent_operations fat32_ops = {
    .ent_blocknr   = fat_ent_blocknr,
    .ent_set_ptr   = fat32_ent_set_ptr,
    .ent_bread     = fat_ent_bread,
    .ent_get       = fat32_ent_get,
    .ent_put       = fat32_ent_put,
    .ent_next      = fat32_ent_next,
};

static const struct inode_operations msdos_dir_inode_operations = {
    .create        = msdos_create,
    .lookup        = msdos_lookup,
    .unlink        = msdos_unlink,
    .mkdir         = msdos_mkdir,
    .rmdir         = msdos_rmdir,
    .rename        = msdos_rename,
    .setattr       = fat_setattr,
    .getattr       = fat_getattr,
};

const struct file_operations fat_file_operations = {
    .llseek        = generic_file_llseek,
    .read_iter     = generic_file_read_iter,
    .write_iter    = generic_file_write_iter,
    .mmap          = generic_file_mmap,
    .release       = fat_file_release,
    .unlocked_ioctl = fat_generic_ioctl,
#ifndef CONFIG_COMPAT
    .compat_ioctl   = fat_generic_compat_ioctl,
#endif
    .fsync          = fat_file_fsync,
    .splice_read   = generic_file_splice_read,
    .fallocate     = fat_fallocate,
};

static const struct inode_operations vfat_dir_inode_operations = {
    .create        = vfat_create,
    .lookup        = vfat_lookup,
    .unlink        = vfat_unlink,
    .mkdir         = vfat_mkdir,
    .rmdir         = vfat_rmdir,
    .rename        = vfat_rename,
    .setattr       = fat setattr,
    .getattr       = fat getattr,
};

```

Ο **Editor** που χρησιμοποιήθηκε (**Sublime Text 3**) για την τροποποίηση του κώδικα της βιβλιοθήκης LKL, προσφέρει τη δυνατότητα στο χρήστη να κάνει **hover** στις συναρτήσεις που έχουν υλοποιηθεί και να **μεταβεί σε σημεία** στα αρχεία που αυτή χρησιμοποιείται ή έχει οριστεί.

Με αξιοποίηση της δυνατότητας αυτής, πραγματοποιήθηκε μετάβασης σε ορισμένες από τις συναρτήσεις αυτές, στις οποίες κρίθηκε απαραίτητη η μελέτη του κώδικα τους, και έγινε κατάλληλη χρήση των κλήσεων συστήματος **sys_***() και της κλήσης **printf()**.

Στο σημείο αυτό θα εξηγηθεί αναλυτικά ο κώδικας που προστέθηκε σε καθένα από τα παραπάνω αρχεία, καθώς και ο λόγος που πάρθηκε η απόφαση για τροποποίηση του κώδικα στα σημεία αυτά.

Αρχείο inode.c

Ο κώδικας του αρχείου **inode.c** αφορά της λειτουργίες του **superblock**, καθώς και του χώρου διεύθυνσεων (**address space**) της μνήμης του Linux. Στις συναρτήσεις των αντίστοιχων structs έχουν χρησιμοποιηθεί οι κλήσεις συστήματος sys_* καθώς και η κλήση printf() του πυρήνα (**Kernel**) του Linux.

Αναφορικά με τις κλήσεις sys_*, έχει γίνει include η βιβλιοθήκη **syscalls.h** όπως φαίνεται στην παρακάτω εικόνα στη γραμμή 24, ώστε να είναι δυνατή η αξιοποίηση των κλήσεων αυτών.

```
13  #include <linux/module.h>
14  #include <linux/pagemap.h>
15  #include <linux/mpage.h>
16  #include <linux/vfs.h>
17  #include <linux/seq_file.h>
18  #include <linux/parser.h>
19  #include <linux/uio.h>
20  #include <linux/blkdev.h>
21  #include <linux/backing-dev.h>
22  #include <asm/unaligned.h>
23  //Added to perform sys_call/sys_close
24  #include <linux/syscalls.h>
25  #include "fat.h"
26
27  #ifndef CONFIG_FAT_DEFAULT_I_CHARSET
28  /* if user don't select VFAT, this is undefined */
29  #define CONFIG_FAT_DEFAULT_I_CHARSET    ""
30  #endif
```

Έχοντας ως γνώμονα το struct fat_sops, εξετάστηκε ο κώδικας των εξής συναρτήσεων από τις ακόλουθες:

- **fat_alloc_inode()**
- **fat_destroy_inode()**
- **fat_write_inode()**
- **fat_evict_inode()**
- **fat_put_super()**

```
static int fat_show_options(struct seq_file *m, struct dentry *root);
static const struct super_operations fat_sops = {
    .alloc_inode    = fat_alloc_inode,
    .destroy_inode = fat_destroy_inode,
    .write_inode   = fat_write_inode,
    .evict_inode   = fat_evict_inode,
    .put_super     = fat_put_super,
    .statfs        = fat_statfs,
    .remount_fs   = fat_remount,
    .show_options  = fat_show_options,
};
```

Συνάρτηση fat_alloc_inode()

```
static struct inode *fat_alloc_inode(struct super_block *sb)
{
    printk(KERN_INFO "file: inode.c fat_alloc_inode() START\n");
    struct msdos_inode_info *ei;
    ei = kmalloc(sizeof(fat_inode_cachep), GFP_NOFS);
    if (!ei)
        return NULL;

    init_rwsem(&ei->truncate_lock);
    printk(KERN_INFO "file: inode.c fat_alloc_inode() END\n");
    return &ei->vfs_inode;
}
```

Στον κώδικα της συνάρτησης αυτής έχουν προστεθεί οι κλήσεις **printk()** με κατάλληλα διαγνωστικά μηνύματα, ώστε ο χρήστης να έχει τη δυνατότητα να διακρίνει **πότε γίνεται χρονικά η κλήση αυτή** από το σύστημα FAT, αλλά και να γνωρίζει πότε **αρχίζει και τελειώνει** η χρήση της, καθώς μεταξύ της κλήσης αυτής ενδέχεται να πραγματοποιούνται κλήσεις και από τις υπόλοιπες δομές.

Συγκεκριμένα, η κλήση αυτή πραγματοποιείται για κάθε αρχείο που πρέπει να αντιγραφεί στο σύστημα αρχείων FAT και στη συνέχεια ακολουθούν οι κλήσεις από τις υπόλοιπες δομές, στις οποίες θα γίνει αντίστοιχη αναφορά παρακάτω.

```

0.018142] This architecture does not have kernel memory protection.
0.018416] file: namei_vfat.c vfat_mount START
0.018421] file: namei_vfat.c vfat_fill_super START
0.018445] file: inode.c fat_fill_super START
0.018451] Journal opens to write superblock data
0.018478] FILE DESCRIPTOR 0
0.018483] file: inode.c Journal opened successfully
0.018490] sb->s_flags 268437504
0.018495] sb->s_magic 19780
0.018497] file: namei_fat.c setup
0.018512] sbi->vol_id -15078511
0.018528] sbi->fsinfo_sector 0l
0.018532] sbi->dir_per_block 16
0.018538] sbi->dir_per_block_bits 4
0.018540] sbi->dir_start 404l
0.018543] sbi->dir_entries 512i
0.018546] sbi->fat_bits 0
0.018549] sbi->fat_length 200
0.018551] sbi->root_cluster 0
0.018554] sbi->free_clusters 4294967295
0.018556] sbi->prev_free 2i
0.018559] sbi->fat_bits 16
0.018561] sbi->dirty 0

```

```

0.018565] file: inode.c fat_alloc_inode() START
0.018571] file: inode.c fat_alloc_inode() END
0.018574] file: inode.c fat_alloc_inode() START
0.018576] file: inode.c fat_alloc_inode() END
0.018578] file: inode.c fat_alloc_inode() START
0.018580] file: inode.c fat_alloc_inode() END
0.018868] file: inode.c fat_fill_super END
0.018946] file: namei_fat.c vfat_lookup START
0.018978] file: namei_fat.c vfat_find START
0.018980] file: namei_fat.c vfat_find END
0.018985] file: inode.c fat_alloc_inode() START
0.018987] file: inode.c fat_alloc_inode() END
0.018999] file: namei_fat.c vfat_lookup END

```

Σχετικά με τα διαγνωστικά μηνύματα που έχουν συμπεριληφθεί στην **αριστερή εικόνα**, τα μηνύματα αυτά αφορούν την **εκκίνηση του συστήματος FAT**, τη στιγμή που αυτό ξεκινά να χρησιμοποιείται ως εικονικός δίσκος (**mounting**).

Ο σχολιασμός των μηνυμάτων αυτών θα πραγματοποιηθεί τόσο κατά την εξήγηση του κώδικα που έχει προστεθεί στο αρχείο **namei_vfat.c**, όσο και κατά την εξήγηση του κώδικα που προστέθηκε στη συνάρτηση **fat_fill_super()** του αρχείου **inode.c**

Συνάρτηση **fat_destroy_inode()**

Ομοίως με τη συνάρτηση **fat_alloc_inode()**, στη συνάρτηση **fat_destroy_inode()** έχει γίνει χρήση των κλήσεων `printk()` με προσθήκη κατάλληλων διαγνωστικών μηνυμάτων, ώστε ο χρήστης να έχει τη δυνατότητα να διακρίνει πότε γίνεται χρονικά η κλήση αυτή από το σύστημα FAT, αλλά και να γνωρίζει πότε αρχίζει και τελειώνει η χρήση της, καθώς μεταξύ της κλήσης αυτής ενδέχεται να πραγματοποιούνται κλήσεις και από τις υπόλοιπες δομές.

Η **παρούσα συνάρτηση** της βιβλιοθήκης LKL καλείται **αφότου γραφούν** οι πληροφορίες της δομής `inode` του εκάστοτε αρχείου, δηλαδή μόλις ολοκληρωθεί η κλήση της συνάρτησης **fat_write_inode()**.

Ο ρόλος της συνάρτησης αυτής είναι η **αποδέσμευση της μνήμης** που έχει δεσμευτεί κατά την εξέταση των πληροφοριών που εμπεριέχονται στο `inode`.

```

static void fat_destroy_inode(struct inode *inode)
{
    printk(KERN_INFO "file: inode.c fat_destroy_inode() START\n");
    call_rcu(&inode->i_rcu, fat_i_callback);
    printk(KERN_INFO "file: inode.c fat_destroy_inode() END\n");
}

```

Συνάρτηση fat_write_inode()

```
static int fat_write_inode(struct inode *inode, struct writeback_control *wbc)
{
    int err;

    printk(KERN_INFO "file: inode.c fat_write_inode() START\n");
    if (inode->i_ino == MSDOS_FSINFO_INO) {
        struct super_block *sb = inode->i_sb;

        mutex_lock(&MSDOS_SB(sb)->s_lock);
        err = fat_clusters_flush(sb);
        mutex_unlock(&MSDOS_SB(sb)->s_lock);
    } else
        err = __fat_write_inode(inode, wbc->sync_mode == WB_SYNC_ALL);
    printk(KERN_INFO "file: inode.c fat_write_inode() END\n");

    return err;
}
```

Η συνάρτηση αυτή είναι **υπεύθυνη για την εγγραφή**, στη δομή inode του συστήματος αρχείων FAT, των **μεταδεδομένων** που αφορούν τα αρχεία που επρόκειτο να αποθηκευτούν σε αυτό.

Στον κώδικα της συνάρτησης αυτής έχουν προστεθεί οι **κλήσεις printk()** με **κατάλληλα διαγνωστικά μηνύματα**, ώστε ο χρήστης να έχει τη δυνατότητα να διακρίνει πότε γίνεται χρονικά η κλήση αυτή από το σύστημα FAT, αλλά και να γνωρίζει πότε αρχίζει και τελειώνει η χρήση της, καθώς μεταξύ της κλήσης αυτής ενδέχεται να πραγματοποιούνται κλήσεις και από τις υπόλοιπες δομές.

Ο τρόπος με τον οποίο καταγράφονται τα πεδία στο αρχείο καταγραφής είναι κοινός για όλες τις συναρτήσεις των δομών του συστήματος αρχείων όπου κρίθηκε απαραίτητη η καταγραφή των ενδεχόμενων αλλαγών στα πεδία αυτά.

Πιο συγκεκριμένα:

1. Έλεγχος για τη τιμή της μεταβλητής journal

- Εάν η τιμή είναι **μη-αρνητική**, τότε η τιμή της μεταβλητής **file_opened** τίθεται στη **τιμή 1**
 - Εάν η τιμή είναι **αρνητική**, τότε η τιμή της μεταβλητής **file_opened** τίθεται στη **τιμή 0**
2. Εάν η τιμή μεταβλητής **file_opened** είναι η **τιμή 1**, τότε καταγράφονται τα πεδία με τη χρήση της κλήσης συστήματος **sys_write()** στο αρχείο καταγραφής, καταγράφει τις αλλαγές που έγιναν στο journal μέσω των κλήσεων **sys_fsync()** και **sys_fdatasync()** και ενημερώνει το σύστημα αρχείων FAT για της αλλαγές αυτές.
3. Μόλις καταγραφεί και το τελευταίο πεδίο, πραγματοποιείται η **κλήση sys_close()**.

Εάν η τιμή μεταβλητής **file_opened** είναι η **τιμή 0**, το άνοιγμα του αρχείου δεν πραγματοποιείται.

Στη συνάρτηση αυτή καλεύται επίσης η συνάρτηση **_fat_write_inode()**, της οποίας ο κώδικας εξετάστηκε και κρίθηκε απαραίτητη η εγγραφή των δεδομένων που αφορούν τη δομή inode στο αρχείο καταγραφής (**journal**) στο σημείο αυτό.

```
1043 static int __fat_write_inode(struct inode *inode, int wait)
1044 {
1045     struct super_block *sb = inode->i_sb;
1046     struct msdos_sb_info *sbi = MSDOS_SB(sb);
1047     struct buffer_head *bh;
1048     struct msdos_dir_entry *raw_entry;
1049     loff_t i_pos;
1050     sector_t blocknr;
1051     int err, offset;
1052     long len;
1053
1054     printk(KERN_INFO "file: inode.c __fat_write_inode START\n");
1055     printk(KERN_INFO "\n");
1056     printk(KERN_INFO "Journal opens to write superblock data\n");
1057
1058     journal = sys_open("journal", O_CREAT|O_WRONLY|O_APPEND,S_IRWXU);
1059     printk(KERN_INFO "FILE DESCRIPTOR %d \n",journal);
1060
1061     if (journal>=0){
1062         printk(KERN_INFO "file: inode.c Journal opened successfully\n");
1063         //can define a macro for that
1064         file_opened =1;
1065     }
1066     else{
1067         file_opened =0;
1068         printk(KERN_INFO "file: inode.c Could not open journal\n");
1069     }
1070     printk(KERN_INFO "\n");
1071
1072     if (inode->i_ino == MSDOS_ROOT_INO)
1073         return 0;
1074
1075     retry:
1076         i_pos = fat_i_pos_read(sbi, inode);
1077         if (!i_pos)
1078             return 0;
1079
1080         fat_get_blknr_offset(sbi, i_pos, &blocknr, &offset);
1081
1082         printk(KERN_INFO "file: inode.c sb_bread() START\n");
1083
1084         bh = sb_bread(sb, blocknr);
1085
1086         if (file_opened){
1087             printk(" ----- WRITING CLUSTER CHANGES INTO THE JOURNAL START -----");
1088
1089             sprintf(file_buf,"bh->b_state %u",bh->b_state);
1090             sys_write(journal,file_buf,FILE_BUFFER_SIZE);
1091             sys_fsync(journal);
1092             sys_fdatasync(journal);
1093             printk(KERN_INFO "%s \n",file_buf);
1094
1095             sprintf(file_buf,"bh->b_blocknr %d",bh->b_blocknr);
1096             sys_write(journal,file_buf,FILE_BUFFER_SIZE);
1097             sys_fsync(journal);
1098             sys_fdatasync(journal);
1099             printk(KERN_INFO "%s \n",file_buf);
1100
1101             sprintf(file_buf,"bh->b_size %d",bh->b_size);
1102             sys_write(journal,file_buf,FILE_BUFFER_SIZE);
1103             sys_fsync(journal);
1104             sys_fdatasync(journal);
1105             printk(KERN_INFO "%s \n",file_buf);
1106
1107             sprintf(file_buf,"bh->b_data %p",bh->b_data);
1108             sys_write(journal,file_buf,FILE_BUFFER_SIZE);
1109             sys_fsync(journal);
1110             sys_fdatasync(journal);
1111             printk(KERN_INFO "%s \n",file_buf);
1112
1113
1114         }
1115
1116         printk(" ----- WRITING CLUSTER CHANGES INTO THE JOURNAL END -----");
1117 }
```

Στην αριστερή εικόνα, έχουν οριστεί οι μεταβλητές **journal** και **file_opened**, τύπου **int** και οι δύο.

Η μεταβλητή **journal** πρόκειται για τον **file descriptor** του αρχείου καταγραφής που ανοίγει για τη καταγραφή των αλλαγών στο σύστημα αρχείων FAT.

Η μεταβλητή **file_opened** χρησιμοποιείται για τον έλεγχο ορθού ανοίγματος του αρχείου καταγραφής, δηλαδή ο file descriptor να έχει **μη-αρνητική τιμή**.
Εάν η τιμή του **file_descriptor** είναι **μη-αρνητική τιμή**, η τιμή της μεταβλητής **file_opened** τίθεται στη τιμή 1, διαφορετικά τυπώνεται **κατάλληλο μήνυμα σφάλματος προς το χρήστη**.

Για τη καταγραφή των πεδίων, χρησιμοποιείται η μεταβλητή **file_buf** η οποία πρόκειται για ένα πίνακα τύπου **char** μεγέθους **FILE_BUFFER_SIZE**.

Ο **πίνακας από char** είναι το αντίστοιχο του **τύπου string** που χρησιμοποιείται σε άλλες γλώσσες προγραμματισμού αλλά **όχι στη γλώσσα C**.

Το **macro FILE_BUFFER_SIZE** έχει οριστεί στο header file “**fat.h**”, δίνοντας στο χρήστη τη δυνατότητα να μεταβάλλει **δυναμικά** τη τιμή του, η οποία κατά συνέπεια μεταβάλλει και το μέγιστο μέγεθος του string που θα αποθηκευτεί στο αρχείο καταγραφής.

Η τιμή που έχει ανατεθεί στο header file “**fat.h**” είναι η **τιμή 128**, ωστόσο εάν ο χρήστης επιθυμεί να τη μεταβάλλει, ενδείκνυται μία **τιμή μεγαλύτερη του 64** για να αποθηκευτεί με **βεβαιότητα** η πληροφορία των πεδίων στο αρχείο καταγραφής.

Στη **δεξιά εικόνα**, αφού πραγματοποιηθεί η κλήση **sb_read()**, μέσω της οποίας δίνεται η πρόσβαση στις πληροφορίες του cluster με συνολικό μέγεθος εγγραφής **blocknr** του συστήματος αρχείων FAT, οι πληροφορίες αυτές ανατίθενται στη μεταβλητή **bh**, η οποία είναι τύπου **struct buffer_head ***.

Δεδομένου ότι το documentation της βιβλιοθήκης LKL στην εκφώνηση δεν αναφέρει δεν περιλαμβάνει τα πεδία του struct buffer_head, έγινε αναζήτηση στο φιλομετρητή Google Chrome για την αναζήτηση των πεδίων του.

Το πόρισμα της αναζήτησης σχετικά με τα πεδία του struct αυτού είναι το εξής:

```
The buffer_head structure, as found in <linux/buffer_head.h> in the 2.6.29 kernel.

/*
 * Historically, a buffer_head was used to map a single block
 * within a page, and of course as the unit of I/O through the
 * filesystem and block layers. Nowadays the basic I/O unit
 * is the bio, and buffer_heads are used for extracting block
 * mappings (via a get_block_t call), for tracking state within
 * a page (via a page_mapping) and for wrapping bio submission
 * for backward compatibility reasons (e.g. submit_bh).
 */
struct buffer_head {
    unsigned long b_state;           /* buffer state bitmap (see above) */
    struct buffer_head *b_this_page; /* circular list of page's buffers */
    struct page *b_page;            /* the page this bh is mapped to */

    sector_t b_blocknr;             /* start block number */
    size_t b_size;                 /* size of mapping */
    char *b_data;                  /* pointer to data within the page */

    struct block_device *b_bdev;    /* I/O completion */
    bh_end_io_t *b_end_io;          /* reserved for b_end_io */
    void *b_private;               /* associated with another mapping */
    struct list_head b_assoc_buffers; /* mapping this buffer is
                                         associated with */
    struct address_space *b_assoc_map; /* users using this buffer_head */
    atomic_t b_count;               /* */
};

};

The buffer_head structure, as found in <linux/buffer_head.h> in the 2.6.29 kernel.
```

Από τα πεδία που περιέχονται στο struct buffer_head, κρίθηκε απαραίτητο να καταγράφονται στο journal τα εξής πεδία, με τη δομή του κώδικα που περιγράφηκε νωρίτερα σχετικά με τη καταγραφή των πεδίων:

- Το πεδίο **b_state**, το οποίο αφορά τη κατάσταση που βρίσκεται το cluster.
- Το πεδίο **b_blocknr**, το οποίο πρόκειται το συνολικό μέγεθος της εγγραφής σε bits στο δίσκο
- Το πεδίο **b_data**, το οποίο πρόκειται για pointer στα δεδομένα του cluster

Συνεχίζοντας στο κώδικα της ίδιας συνάρτησης, καταγράφονται οι αλλαγές που πραγματοποιούνται στη δομή directory entry ή dentry.

Με βάση το documentation της βιβλιοθήκης LKL, το struct που αντιπροσωπεύει τη δομή msdos_dir_entry έχει τα πεδία που παρουσιάζονται στην παρακάτω εικόνα.

```
#include <msdos_fs.h>
```

Data Fields

```
__u8    name [MSDOS_NAME]
__u8    attr
__u8    lcase
__u8    ctime_cs
__le16   ctime
__le16   cdate
__le16   adate
__le16   starthi
__le16   time
__le16   date
__le16   start
__le32   size
```

Για τα πεδία αυτά, κρίθηκε απαραίτητη η καταγραφή τους στο αρχείο καταγραφής (journal), καθώς πριν τη δομή if που προστέθηκε, προηγείται κώδικας που ενδέχεται να επιφέρει μεταβολές στα πεδία της μεταβλητής raw_entry.

Ένα επιπλόν πεδίο που κρίθηκε απαραίτητη η καταγραφή του είναι το πεδίο name το οποίο προσδιορίζει με έναν ακέραιο το inode του αρχείου που αντιγράφεται στο αρχείο καταγραφής FAT.

```

1114     printk(" ----- WRITING CLUSTER CHANGES INTO THE JOURNAL END -----");
1115 }
1116
1117
1118 printk(KERN_INFO "file: inode.c sb_bread() END\n");
1119
1120 if (!bh) {
1121     fat_msg(sb, KERN_ERR, "unable to read inode block "
1122             "for updating (i_pos %lld)", i_pos);
1123     return -EIO;
1124 }
1125 spin_lock(&sbi->inode_hash_lock);
1126 if (i_pos != MSDOS_I(inode)->i_pos) {
1127     spin_unlock(&sbi->inode_hash_lock);
1128     brelse(bh);
1129     goto retry;
1130 }
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2489
2490
2491
2492
2493
2494
2495
2496
2497
2497
2498
2499
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2589
2590
2591
2592
2593
2594
2595
2596
2597
2597
2598
2599
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2689
2690
2691
2692
2693
2694
2695
2696
2697
2697
2698
2699
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2789
2790
2791
2792
2793
2794
2795
2796
2797
2797
2798
2799
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2889
2890
2891
2892
2893
2894
2895
2896
2897
2897
2898
2899
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2989
2990
2991
2992
2993
2994
2995
2996
2997
2998
2999
2999
3000
3001
3002
3003
3004
3005
3006
3007
3008
3009
3009
3010
3011
3012
3013
3014
3015
3016
3017
3018
3019
3019
3020
3021
3022
3023
3024
3025
3026
3027
3028
3029
3029
3030
3031
3032
3033
3034
3035
3036
3037
3038
3039
3039
3040
3041
3042
3043
3044
3045
3046
3047
3048
3049
3049
3050
3051
3052
3053
3054
3055
3056
3057
3058
3059
3059
3060
3061
3062
3063
3064
3065
3066
3067
3068
3069
3069
3070
3071
3072
3073
3074
3075
3076
3077
3078
3079
3079
3080
3081
3082
3083
3084
3085
3086
3087
3088
3089
3089
3090
3091
3092
3093
3094
3095
3096
3097
3098
3098
3099
3099
3100
3101
3102
3103
3104
3105
3106
3107
3108
3109
3109
3110
3111
3112
3113
3114
3115
3116
3117
3118
3119
3119
3120
3121
3122
3123
3124
3125
3126
3127
3128
3129
3129
3130
3131
3132
3133
3134
3135
3136
3137
3138
3139
3139
3140
3141
3142
3143
3144
3145
3146
3147
3148
3149
3149
3150
3151
3152
3153
3154
3155
3156
3157
3158
3159
3159
3160
3161
3162
3163
3164
3165
3166
3167
3168
3169
3169
3170
3171
3172
3173
3174
3175
3176
3177
3178
3179
3179
3180
3181
3182
3183
3184
3185
3186
3187
3188
3189
3189
3190
3191
3192
3193
3194
3195
3196
3197
3198
3198
3199
3199
3200
3201
3202
3203
3204
3205
3206
3207
3208
3209
3209
3210
3211
3212
3213
3214
3215
3216
3217
3218
3219
3219
3220
3221
3222
3223
3224
3225
3226
3227
3228
3229
3229
3230
3231
3232
3233
3234
3235
3236
3237
3238
3239
3239
3240
3241
3242
3243
3244
3245
3246
3247
3248
3249
3249
3250
3251
3252
3253
3254
3255
3256
3257
3258
3259
3259
3260
3261
3262
3263
3264
3265
3266
3267
3268
3269
3269
3270
3271
3272
3273
3274
3275
3276
3277
3278
3279
3279
3280
3281
3282
3283
3284
3285
3286
3287
3288
3289
3289
3290
3291
3292
3293
3294
3295
3296
3297
3298
3298
3299
3299
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3309
3310
3311
3312
3313
3314
3315
3316
3317
3318
3319
3319
3320
3321
3322
3323
3324
3325
3326
3327
3328
3329
3329
3330
3331
3332
3333
3334
3335
3336
3337
3338
3339
3339
3340
3341
3342
3343
3344
3345
3346
3347
3348
3349
3349
3350
3351
3352
3353
3354
3355
3356
3357
3358
3359
3359
3360
3361
3362
3363
3364
3365
3366
3367
3368
3369
3369
3370
3371
3372
3373
3374
3375
3376
3377
3378
3379
3379
3380
3381
3382
3383
3384
3385
3386
3387

```

Οι τιμές των πεδίων που ενδέχεται να μεταβάλλονται παρουσιάζονται στη παρακάτω εικόνα, όπου παρουσιάζεται επίσης το γεγονός ότι η συνάρτηση fat_write_inode() καλεί τη __fat_write_inode() για τη καταγραφή των δεδομένων του inode του παρόντος αρχείου.

```
[ 0.025836] file: inode.c fat_write_inode() START
[ 0.025840] file: inode.c __fat_write_inode START
[ 0.025843]
[ 0.025844] Journal opens to write superblock data
[ 0.025856] FILE DESCRIPTOR 0
[ 0.025860] file: inode.c Journal opened successfully
[ 0.025862]
[ 0.025864] file: inode.c sb_bread() START
[ 0.025866] ----- WRITING CLUSTER CHANGES INTO THE JOURNAL START -----
[ 0.025869] bh->b_state 41
[ 0.025874] bh->b_blocknr 404
[ 0.025877] bh->b_size 512
[ 0.025880] bh->b_data 00007fef5ffffe800
[ 0.025882] ----- WRITING CLUSTER CHANGES INTO THE JOURNAL END -----
[ 0.025882] file: inode.c sb_bread() END
[ 0.025887] ----- WRITING DIRECTORY ENTRY CHANGES INTO THE JOURNAL START -----
[ 0.025888] raw_entry->name 16106066624
[ 0.025893] raw_entry->attr 32
[ 0.025895] raw_entry->size 13958
[ 0.025898] raw_entry->ctime 0
[ 0.025901] raw_entry->ctime_cs 0
[ 0.025903] raw_entry->date 33
[ 0.025905] raw_entry->date 33
[ 0.025906] raw_entry->date 33
[ 0.025910] ----- WRITING DIRECTORY ENTRY CHANGES INTO THE JOURNAL END -----
[ 0.025911] file: inode.c __fat_write_inode END
[ 0.025914] file: inode.c fat_write_inode() END
```

Συνάρτηση fat_evict_inode()

Η συνάρτηση **fat_evict_inode()** συνιστά μία από τις συναρτήσεις που αποτελεί πεδίο του **struct fat_sops** και ,επομένως, έγινε **έλεγχος του κώδικα της**. Στο σημείο αυτό, διαπιστώθηκε , μέσα από αναζήτηση στο documentation της βιβλιοθήκης LKL, ότι η συνάρτηση αυτή καλεί τη συνάρτηση **clear_inode()**, η οποία μεταβάλλει το πεδίο **i_state του inode**, ώστε με βάση τη τιμή αυτή να γνωρίζει το σύστημα αρχείων εάν πρέπει τη χρονική αυτή στιγμή να αποδεσμευτεί το inode από τη μνήμη, εφόσον έχουν γραφεί οι πληροφορίες του.

◆ clear_inode()

```
void clear_inode ( struct inode * inode )
```

Definition at line 498 of file inode.c.

```
499 {   might_sleep();
500 /*
501  * We have to cycle tree_lock here because reclaim can be still in the
502  * process of removing the last page (in __delete_from_page_cache())
503  * and we must not free mapping under it.
504  */
505 spin_lock_irq(&inode->i_data.tree_lock);
506 BUG_ON(inode->i_data.nrpages);
507 BUG_ON(inode->i_data.nrexceptional);
508 spin_unlock_irq(&inode->i_data.tree_lock);
509 BUG_ON(!list_empty(&inode->i_data.private_list));
510 BUG_ON(!(inode->i_state & I_FREEING));
511 BUG_ON(inode->i_state & I_CLEAR);
512 BUG_ON(!list_empty(&inode->i_wb_list));
513 /* don't need i_lock here, no concurrent mods to i_state */
514 inode->i_state = I_FREEING | I_CLEAR;
515 }
```

Όπως **επισημάνθηκε νωρίτερα**, ο κώδικας που χρησιμοποιείται στη παρακάτω εικόνα έχει την **ίδια δομή** για όλες τις συναρτήσεις όπου καταγράφονται αλλαγές σε πεδία των δομών του συστήματος αρχείων FAT και η δομή αυτή είναι η εξής:

1. Έλεγχος για τη τιμή της μεταβλητής **journal**

- Εάν η τιμή είναι **μη-αρνητική**, τότε η τιμή της μεταβλητής **file_opened** τίθεται στη τιμή **1**
- Εάν η τιμή είναι **αρνητική**, τότε η τιμή της μεταβλητής **file_opened** τίθεται στη **τιμή 0**

2.

Εάν η τιμή μεταβλητής **file_opened** είναι η **τιμή 1**, τότε καταγράφονται τα πεδία με τη χρήση της κλήσης συστήματος **sys_write()** στο αρχείο καταγραφής, καταγράφει τις αλλαγές που έγιναν στο journal μέσω των κλήσεων **sys_fsync()** και **sys_fdatasync()** και ενημερώνει το σύστημα αρχείων FAT για τις αλλαγές αυτές.

3. Μόλις καταγραφεί και το τελευταίο πεδίο, πραγματοποιείται η **κλήση sys_close()**.

Εάν η τιμή μεταβλητής **file_opened** είναι η **τιμή 0**, το άνοιγμα του αρχείου δεν πραγματοποιείται.

Η εκτέλεση της συνάρτησης **fat_evict_inode()** πραγματοποιείται **μετά την ολοκλήρωση** της κλήσης της συνάρτησης **fat_write_inode()**, ώστε να απελευθερωθεί το inode από τη μνήμη του συστήματος. Η καταγραφή της κατάστασης που βρίσκεται δίνει τη δυνατότητα στο σύστημα, σε περίπτωση κάποιας αστοχίας, να έχει αποθηκευμένη τη κατάσταση αυτή και χρησιμοποιώντας της να προσαρμοστεί με ανάλογο τρόπο, προκειμένου να εκτελεστεί επιβλαβείς επιπτώσεις.

```
static void fat_evict_inode(struct inode *inode)
{
    printk(KERN_INFO "file: inode.c fat_evict_inode() START\n");
    trunc_inode(inode);
    if (!inode->i_link) {
        if (inode->i_size == 0)
            fat_truncate_blocks(inode, 0);
        else
            fat_free_eofblocks(inode);
        invalidate_inode_buffers(inode);
        clear_inode(inode);
    }
    int journal;
    int file_opened;
    journal = sys_open("journal", O_CREAT | O_WRONLY | O_APPEND, S_IRWXU);
    printk(KERN_INFO "FILE DESCRIPTOR %d \n", journal);
    if (journal>=0){
        printk(KERN_INFO "file: inode.c Journal opened successfully\n");
        //can define a macro for that
        file_opened =1;
    }
    else{
        file_opened =0;
        printk(KERN_INFO "file: inode.c Could not open journal\n");
    }
    if (file_opened){
        char file_buf[FILE_BUFFER_SIZE];
        printk(KERN_INFO "----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED START-----\n");
        sprintf(file_buf, "sb1->fat_inode->i_state %u", inode->i_state);
        sys_write(journal, file_buf, FILE_BUFFER_SIZE);
        sys_fsync(journal);
        sys_fdatasync(journal);
        printk(KERN_INFO "%s \n", file_buf);
        printk(KERN_INFO "----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED END-----\n");
    }
    sys_close(journal);
    printk(KERN_INFO "\n");
    fat_cache_inval_inode(inode);
    fat_detach(inode);
    printk(KERN_INFO "file: inode.c fat_evict_inode() END\n");
}
```

```

0.021781] file: inode.c __fat_write_inode END
0.021785] file: inode.c fat_write_inode() END
0.021836] file: inode.c fat_evict_inode() START
0.021862] FILE DESCRIPTOR 0
0.021866] file: inode.c Journal opened successfully
0.021868] ----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED START-----
0.021872] inode->i_state 96
0.021874] ----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED END-----
0.021876]
0.021878] file: inode.c fat_evict_inode() END
0.021880] file: inode.c fat_destroy_inode() START
0.021889] file: inode.c fat_destroy_inode() END
0.021891] file: inode.c fat_evict_inode() START
0.021906] FILE DESCRIPTOR 0
0.021908] file: inode.c Journal opened successfully
0.021910] ----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED START-----
0.021913] inode->i_state 96
0.021915] ----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED END-----
0.021917]
0.021919] file: inode.c fat_evict_inode() END
0.021921] file: inode.c fat_destroy_inode() START
0.021923] file: inode.c fat_destroy_inode() END
0.021925] file: inode.c fat_put_super() START
0.021938] file: inode.c fat_evict_inode() START
0.021961] FILE DESCRIPTOR 0
0.021965] file: inode.c Journal opened successfully
0.021967] ----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED START-----
0.021970] inode->i_state 96
0.021972] ----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED END-----
0.021974]
0.021976] file: inode.c fat_evict_inode() END
0.021978] file: inode.c fat_destroy_inode() START
0.021980] file: inode.c fat_destroy_inode() END
0.021982] file: inode.c fat_evict_inode() START
0.021992] FILE DESCRIPTOR 0
0.021995] file: inode.c Journal opened successfully
0.021997] ----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED START-----
0.022000] inode->i_state 96
0.022002] ----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED END-----
0.022004]

```

Η συνάρτηση **fat_put_super()** συνιστά μία από τις συναρτήσεις που αποτελεί πεδίο του **struct fat_sops** και ,επομένως, έγινε έλεγχος του κώδικα της. Στο σημείο αυτό, διαπιστώθηκε , μέσα από **αναζήτηση στο documentation της βιβλιοθήκης LKL**, ότι η συνάρτηση αυτή καλεί τη συνάρτηση **iput()**, η οποία μειώνει συνεχώς τον counter του struct inode, μέχρις ότου αυτό να μην είναι πλέον χρήσιμο για το σύστημα και συνεπώς να **πρέπει να αποδεσμευτεί** από τη μνήμη.

```

static void fat_put_super(struct super_block *sb)
{
    printk(KERN_INFO "file: inode.c fat_put_super() START\n");
    struct msdos_sb_info *sbi = MSDOS_SB(sb);

    fat_set_state(sb, 0, 0);

    iput(sbi->fsinfo_inode);
    iput(sbi->fat_inode);

    int journal;
    int file_opened;

    journal = sys_open("journal",O_CREAT|O_WRONLY|O_APPEND,S_IRWXU);
    printk(KERN_INFO "FILE DESCRIPTOR %d \n",journal);

    if (journal>=0){
        printk(KERN_INFO "file: inode.c Journal opened successfully\n");
        //can define a macro for that
        file_opened =1;
    }
    else{
        file_opened =0;
        printk(KERN_INFO "file: inode.c Could not open journal\n");
    }

    if (file_opened){
        char file_buf[FILE_BUFFER_SIZE];
        printk(KERN_INFO "file: inode.c fat_put_super()\n");
        printk(KERN_INFO "----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED START-----\n");
        sprintf(file_buf,"%s",sbi->fat_inode->i_state);
        sys_write(journal,file_buf,FILE_BUFFER_SIZE);
        sys_fsync(journal);
        sys_fdatasync(journal);
        printk(KERN_INFO "%s \n",file_buf);
        printk(KERN_INFO "file: inode.c fat_put_super()\n");
        printk(KERN_INFO "----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED END-----\n");
    }
    sys_close(journal);

    printk(KERN_INFO "\n");

    call_rcu(&sbi->rcu, delayed_free);
    printk(KERN_INFO "file: inode.c fat_put_super() END\n");
}

```

• iput()

```
void iput ( struct inode * inode )
```

iput - put an inode @inode: inode to put

PUTS an inode, dropping its usage count. If the inode use count hits zero, the inode is then freed and may also be destroyed.

Consequently, iput() can sleep.

Definition at line 1527 of file inode.c.

```
1528 {  
1529     if (!inode)  
1530         return;  
1531     BUG_ON(inode->i_state & I_CLEAR);  
1532     retry:  
1533         if (atomic_dec_and_lock(&inode->i_count, &inode->i_lock)) {  
1534             if (inode->i_nlink && (inode->i_state & I_DIRTY_TIME)) {  
1535                 atomic_inc(&inode->i_count);  
1536                 inode->i_state & ~I_DIRTY_TIME;  
1537                 spin_unlock(&inode->i_lock);  
1538                 trace_writeback_lazystime_iput(inode);  
1539                 mark_inode_dirty_sync(inode);  
1540                 goto retry;  
1541             }  
1542         }  
1543     }  
1544 }
```

```
0.022854] FILE DESCRIPTOR 0  
0.022856] file: inode.c Journal opened successfully  
0.022858] file: inode.c fat_put_super()  
0.022860] ----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED START-----  
0.022863] sbi->fsinfo_inode->i_state 96  
0.022865] file: inode.c fat_put_super()  
0.022867] ----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED START-----  
0.022870] sbi->fat_inode->i_state 96  
0.022872] file: inode.c fat_put_super()  
0.022874] ----- WRITING THE STATE OF INODE BEFORE IT IS RELEASED END-----  
0.022876]  
0.022878] file: inode.c fat_put_super() END  
0.023047] reboot: Restarting system
```

Η εκτέλεση του της συνάρτησης **fat_put_super()** σηματοδοτεί την εκτέλεση των συναρτήσεων **fat_evict_inode()** και **fat_destroy_inode()**, μέσα από τις οποίες αφαιρείται σταδιακά το inode από τη μνήμη του συστήματος αρχείων FAT. Τη χρονική στιγμή που έχουν ολοκληρωθεί αυτές οι κλήσεις, το inode με την ενημερωμένη **πλέον κατάσταση (state)** επιστρέφεται και καταγράφεται η τιμή του πεδίου **i_state**, προκειμένου να αξιοποιηθεί κατάλληλα από το σύστημα.

Όσον αφορά τις λειτουργίες για το χώρο διευθύνσεων της μνήμης (**address space**), οι λειτουργίες αυτές αποτελούν πεδία του **struct fat_aops**.

Εξετάζοντας τις συναρτήσεις αυτές, κρίθηκε απαραίτητη η προσθήκη των κλήσεων **printk()** στις εξής από αυτές, ώστε να γίνει αντιληπτός ο τρόπος που αλληλεπιδρούν με τις υπόλοιπες δομές του συστήματος αρχείων FAT:

- **fat_readpage()**
- **fat_readpages()**

```
static const struct address_space_operations fat_aops = {  
    .readpage    = fat_readpage,  
    .readpages   = fat_readpages,  
    .writepage   = fat_writepage,  
    .writepages  = fat_writepages,  
    .write_begin  = fat_write_begin,  
    .write_end   = fat_write_end,  
    .direct_IO   = fat_direct_IO,  
    .bmap        = _fat_bmap  
};
```

- **fat_writepage()**
 - **fat_writepages()**
 - **fat_writebegin()**

Συνάρτησεις fat_readpage() και fat_readpages()

```
199 static int fat_readpage(struct file *file, struct page *page)
200 {
201     printk(KERN_INFO "file: inode.c fat_readpage\n");
202     return mpage_readpage(page, fat_get_block);
203 }
```

```
205 static int fat_readpages(struct file *file, struct address_space *mapping,
206 | | | struct list_head *pages, unsigned nr_pages)
207 {
208 | | printk(KERN_INFO "file: inode.c fat_readpages\n");
209 | | return mpage_readpages(mapping, pages, nr_pages, fat_get_block);
210 }
```

Μελετώντας τον κώδικα των συναρτήσεων **fat_readpage()** και **fat_readpages()**, δεν παρατηρείται κάποια μεταβολή στις δομές από τις καλούμενες συναρτήσεις **mpage_readpage()** και **block_write_full_page()** αντίστοιχα.

Οπότε ο λόγος που έχει γίνει χρήση της κλήσης **printk()** σε αυτές είναι για καθοδήγηση μέσα στις δομές του συστήματος αρχείων FAT.

- ◆ `mpage_readpage()`

```
int mpage_readpage( struct page * page,  
                    get_block_t get_block  
)
```

[Definition at line 398 of file mpag.c.](#)

```

399 {
400     struct bio *bio = NULL;
401     sector_t last_block_in_bio = 0;
402     struct buffer_head map_bh;
403     unsigned long first_logical_block = 0;
404     gfp_t gfp = mapping_gfp_constraint(page->mapping, GFP_KERNEL);
405
406     map_bh.b_state = 0;
407     map_bh.b_size = 0;
408     bio = do_mpage_readpage(bio, page, 1, &last_block_in_bio,
409                            &map_bh, &first_logical_block, get_Block, gfp);
410     if (bio)
411         mpage_bio_submit(REQ_OP_READ, 0, bio);
412     return 0;
413 }

```

- ◆ `block_write_full_page()`

```
int block_write_full_page ( struct page * page,
                           get_block_t * get_block,
                           struct writeback_control * wbc
                           )
```

Definition at line 2986 of file buffer.c.

Συνάρτησεις fat_writepage() και fat_writepages()

```
186 static int fat_writepage(struct page *page, struct writeback_control *wbc)
187 {
188     printk(KERN_INFO "file: inode.c fat writepage\n");
189     return block_write_full_page(page, fat_get_block, wbc);
190 }
```

```
192 static int fat_writepages(struct address_space *mapping,
193                          struct writeback_control *wbc)
194 {
195     printk(KERN_INFO "file: inode.c fat_writepages\n");
196     return mpage_writepages(mapping, wbc, fat_get_block);
197 }
```

Σε αντιστοιχία με τη περίπτωση των συναρτήσεων **fat_readpage()** και **fat_readpages()**, οι συναρτήσεις **fat_writepage()** και **fat_writepages()** δεν παρατηρείται να τροποποιούν με οποιοδήποτε τρόπο κάποιο από τα πεδία των δομών του συστήματος αρχείων FAT.

Η προσθήκη των κλήσεων **printk()** σε κάθε μία από αυτές έχει ως απότερο στόχο, όπως και στη περίπτωση των συναρτήσεων **fat_readpage()** και **fat_readpages()**, την **κατανόηση της πορείας που ακολουθεί ο κώδικας** της βιβλιοθήκης LKL και του τρόπου αλληλεπίδρασης των δομών μεταξύ τους.

Συνάρτηση fat_write_begin()

Η συνάρτηση **fat_write_begin()** συμβάλλει στη καταγραφή των απαραίτητων πεδίων της δομής **file**, μεταβάλλοντας τις τιμές των πεδίων του **struct file**. Σύμφωνα με το documentation της βιβλιοθήκης LKL, το **struct file** αποτελείται από τα ακόλουθα πεδία

file Struct Reference	
Data Field	
#include <fs.h>	
Data Fields	
union {	
struct llist_node fu_llist	
struct rcu_head fu_rcuhead	
}	f_u
	struct path f_path
	struct inode * f_inode
	const struct file_operations * f_op
	spinlock_t f_lock
	atomic_long_t f_count
	unsigned int f_flags
	fmode_t f_mode
	struct mutex f_pos_lock
	loff_t f_pos
	struct fown_struct f_owner
	const struct cred * f_cred
	struct file_ra_state f_ra
	u64 f_version
	void * private_data
	struct address_space * f_mapping
	struct file * next
	struct file * parent
	const char * name
	int lineno

Από τα πεδία αυτά, κρίθηκε απαραίτητη η καταγραφή των ακόλουθων:

- Το πεδίο **f_flags**, καθώς αφορά μεταβλητή σημαίας που χρησιμοποιείται από το σύστημα αρχείων FAT. Η καταγραφή ενός τέτοιου πεδίου είναι κρίσιμη για το σύστημα, καθώς σε **περίπτωση κατάρρευσή του**, είναι εφικτή η επαναφορά του στη προγενέστερη κατάσταση του.
- Τα πεδία **d_name και inode**, τα οποία σχετίζονται με το **μονοπάτι που είναι αποθηκευμένο το αρχείο** και συγκεκριμένα έχουν αποθηκευμένο το όνομα όνομα του αρχείου.
Όπως παρατηρείται στις παρακάτω εικόνες, στον κώδικα που έχει προστεθεί, η πρόσβαση στα παραπάνω πεδία που αναφέρθηκαν είναι εφικτή μέσω του πεδίου **f_path**, το οποίο είναι τύπου **struct path**, το οποίο έχει τα ακόλουθα πεδία που παρουσιάζονται στη κάτω αριστερή εικόνα:

- Το πεδίο **mnt** που πρόκειται για το **mounting point** του συστήματος
- Το πεδίο **dentry**, το οποίο πρόκειται για αναφορά στο **directory entry** του αρχείου, όπου περιέχονται πληροφορίες σχετικές με το path του.
Αντίστοιχα με τη σειρά του, το **struct dentry** αποτελείται από τα ακόλουθα πεδία της κάτω δεξιά εικόνα.

- Το **d_parent** που είναι αναφορά στο **γονικό directory** που βρίσκεται το παρόν αρχείο. Η καταγραφή του αποτελεί εξίσου σημαντική πληροφορία για το σύστημα σε περίπτωση κατάρρευσή του.
- Το πεδίο **d_time**, το οποίο αφορά τη χρονική στιγμή που τροποποιήθηκε το αρχείο

path Struct Reference

```
#include <path.h>
```

Data Fields

```
struct vfstype * mnt
struct dentry * dentry
```

dentry Struct Reference

```
#include <dcache.h>

Data Fields
unsigned int d_flags
seqcount_t d_seq
struct hlist_node d_hash
struct dentry * d_parent
struct qstr d_name
struct inode * d_inode
unsigned char d_iname [DNAME_INLINE_LEN]
struct lockref d_lockref
const struct dentry_operations * d_op
struct super_block * d_sb
unsigned long d_time
void * d_fsdata

union {
    struct list_head d_lru
    wait_queue_head_t * d_wait
};

    struct list_head d_child
    struct list_head d_subdirs

union {
    struct hlist_node d_alias
    struct hlist_node d_in_lookup_hash
    struct rcu_head d_rcu
}
d_u
```

Συνεχίζοντας με τον κώδικα που προστέθηκε για τη καταγραφή των αλλαγών στα πεδία του συστήματος αρχείων, προηγείται το άνοιγμα του αρχείου journal και έλεγχος ορθού ανοίγματος, με τα βήματα που αναφέρθηκαν νωρίτερα στη παρούσα αναφορά.

Εφόσον το άνοιγμα του αρχείου journal είναι επιτυχές με τη κλήση sys_open(), ακολουθεί η καταγραφή των πεδίων που που παρουσιάστηκαν παραπάνω με τη κλήση sys_write() και το αρχείο κλείνει με τη κλήση sys_close().

```

221 static int fat_write_begin(struct file *file, struct address_space *mapping,
222         loff_t pos, unsigned len, unsigned flags,
223         struct page **pagep, void **fsdata)
224 {
225     int err;
226
227     int journal;
228     int file_opened;
229
230     *pagep = NULL;
231     printk(KERN_INFO "----- file: inode.c fat_write_begin START -----\\n");
232     err = cont_write_begin(file, mapping, pos, len, flags,
233         pagep, fsdata, fat_get_block,
234         &MSDOS_I(mapping->host)->mmu_private);
235
236     printk(KERN_INFO "\\n");
237     printk(KERN_INFO "Journal opens to write cluster data\\n");
238
239
240     journal = sys_open("journal", O_CREAT|O_WRONLY|O_APPEND, S_IRWXU);
241     printk(KERN_INFO "FILE DESCRIPTOR %d \\n", journal);
242
243     if (journal>=0){
244         printk(KERN_INFO "file: inode.c Journal opened successfully\\n");
245         //can define a macro for that
246         file_opened =1;
247     }
248     else{
249         file_opened =0;
250         printk(KERN_INFO "file: inode.c Could not open journal\\n");
251     }
252
253     printk(KERN_INFO "\\n");
254
255 }
```

```

256     if (file_opened){
257         char file_buf[FILE_BUFFER_SIZE];
258         printk(KERN_INFO "file:inode.c fat_write_begin()\\n");
259         printk(KERN_INFO " ----- WRITING FILE RELATED INFO START ----- \\n");
260         sprintf(file_buf,"file->f_flags %x",file->f_flags);
261         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
262         sys_fsync(journal);
263         sys_fdatasync(journal);
264         printk(KERN_INFO "%s \\n",file_buf);
265
266         sprintf(file_buf,"file->f_path.dentry->d_name.name %s",file->f_path.dentry->d_name.name);
267         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
268         sys_fsync(journal);
269         sys_fdatasync(journal);
270         printk(KERN_INFO "%s \\n",file_buf);
271
272         sprintf(file_buf,"file->f_path->dentry->d_iname %s",file->f_path.dentry->d_iname);
273         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
274         sys_fsync(journal);
275         sys_fdatasync(journal);
276         printk(KERN_INFO "%s \\n",file_buf);
277
278         sprintf(file_buf,"file->f_path->dentry->d_time %u",file->f_path.dentry->d_time);
279         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
280         sys_fsync(journal);
281         sys_fdatasync(journal);
282         printk(KERN_INFO "%s \\n",file_buf);
283
284         printk(KERN_INFO " ----- WRITING FILE RELATED INFO END ----- \\n");
285
286     }
287
288     if (err < 0)
289         fat_write_failed(mapping, pos + len);
290     printk(KERN_INFO "----- file: inode.c fat_write_begin END -----\\n");
291     return err;
292 }
```

Στα διαγνωστικά μηνύματα που ακολουθούν, η συνάρτηση **fat_write_begin()** καλείται **τόσες φορές, όσες και τα αρχεία** που πρόκειται να **καταγραφούν** στο σύστημα αρχείων επί δύο, καθώς τα αρχεία αυτά καταγράφονται και στο αντίγραφο του συστήματος αρχείων, σε περίπτωση που αυτό καταρρεύσει.

Οι πληροφορίες που παρουσιάζονται στις παρακάτω εικόνες οργανώνονται με τον ακόλουθο τρόπο, αλληλεπιδρώντας παράλληλα με τις συναρτήσεις **fat_add_cluster()**, **fat_alloc_cluster()**, **fat16_set_ptr()** και **fat16_ent_put()** του αρχείου fatent.c :

- Η **εγγραφή του αρχείου** στο σύστημα αρχείων **εκκινεί** με την συνάρτηση **fat_write_begin()**
- Οι συναρτήσεις **fat_add_cluster()** και **fat_alloc_cluster()** είναι υπεύθυνες για τη δέσμευση της απαραίτητης μνήμης στο δίσκο και στη διευθυνσιοδότηση των clusters στα οποία θα αποθηκευτεί το αρχείο.
- Η συνάρτηση **fat16_set_ptr()** είναι υπεύθυνη για την **ενημέρωση του pointer που θα δείχνει το cluster** και στη οποία θα αποθηκευτεί το επόμενο, ενώ η συνάρτηση **fat16_ent_put()** **ενημερώνει τα πεδία του cluster** όποτε αυτό είναι αναγκαίο σε μία τιμή που υποδηλώνει ότι είναι **dirty**, καθώς και ποιο ήταν το cluster κατά αριθμό που δεσμεύτηκε πρόσφατα.

```
0.017028] file: inode.c fat_write_begin()
0.017030] ----- WRITING FILE RELATED INFO START -----
0.017032] file->f_flags 8002
0.017034] file->f_path.dentry->d_name.name lklfuse.c
0.017037] file->f_path.dentry->d_parent->d_name.name /
0.017040] file->f_path.dentry->d_parent->d_iname /
0.017043] file->f_path->dentry->d_iname lklfuse.c
0.017045] file->f_path->dentry->d_time 0
0.017048] file->f_path->dentry->d_time 0
0.017050] ----- WRITING FILE RELATED INFO END ---|-----
0.017052] ----- file: inode.c fat_write_begin END -----
0.017056] file: inode.c fat_write_end START
0.017058] ----- file: inode.c fat_write_end END -----
0.017059] ----- file: inode.c fat_write_begin START -----
0.017063] file: inode.c fat_add_cluster START
0.017067] file: fatent.c fat_alloc_clusters START
0.017071]
0.017072] Journal opens to write cluster data
0.017084] FILE DESCRIPTOR 0
0.017087] file: fatent.c Journal opened successfully
0.017088]
0.017090] ----- WRITING WHICH FAT ENTRY WAS PARSED START -----
0.017093] fatent.entry 5
0.017095] ----- WRITING WHICH FAT ENTRY WAS PARSED END -----
0.017097] file: fatent.c fat_ent_blocknr() START
0.017099] file: fatent.c fat_ent_blocknr() END
0.017101]
0.017102] Journal opens to write superblock data
0.017112] FILE DESCRIPTOR 3
0.017115] file: fatent.c Journal opened successfully
0.017117]
0.017118] file: fatent.c fat_ent_bread() START
0.017120] ----- WRITING CLUSTER INFO IN THE JOURNAL START-----
0.017123] fatent->bhs[0]->b_state 43
0.017126] fatent->bhs[0]->b_blocknr 4
0.017129] fatent->bhs[0]->b_size 512
0.017131] fatent->bhs[0]->b_data 00007faa1dc00800
0.017133] ----- WRITING CLUSTER INFO IN THE JOURNAL END-----
0.017135] file: fatent.c fat16_ent_set_ptr() START

0.017135] file: fatent.c fat16_ent_set_ptr() START
0.017145] FILE DESCRIPTOR 3
0.017148] file: inode.c Journal opened successfully
0.017150] ----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START -----
0.017153] fatent->u.enty[0] = 49912400
0.017155] ----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START -----
0.017157]
0.017158] file: fatent.c fat16_ent_set_ptr() END
0.017160] file: fatent.c fat_ent_bread() END
0.017163] file: fatent.c fat16_ent_get() START
0.017165] file: fatent.c fat16_ent_get() EOF
0.017180] NEXT 0
0.017182] file: fatent.c fat16_ent_get() END
0.017184] file: fatent.c fat16_ent_put() START
0.017186] ----- WRITING CLUSTER INFO IN THE JOURNAL -----
0.017188] Journal opens to write superblock data
0.017190] FILE DESCRIPTOR 3
0.017202] file: fatent.c Journal opened successfully
0.017204] ----- WRITING CLUSTER INFO IN THE JOURNAL START-----
0.017206] fatent->bhs[0]->b_state MARKED DIRTY
0.017213] fatent->bhs[0]->b_blocknr 43
0.017216] fatent->bhs[0]->b_size 512
0.017221] fatent->bhs[0]->b_data 00007faa1dc00800
0.017224] fatent->fat_inode->i_mode 0
0.017226] fatent->fat_inode->i_size 0
0.017231] fatent->fat_inode->i_flags 0
0.017233] fatent->fat_inode->dirited_when 0
0.017238] fatent->fat_inode->dirited_time when -
0.017240] file: fatent.c fat16_ent_put() END
0.017242] ----- WRITING LAST FAT ENTRY THAT WAS INSERTED START -----
0.017245] sbi->prev_free 5
0.017247] ----- WRITING LAST FAT ENTRY THAT WAS INSERTED END -----
0.017249] file: fatent.c fat_alloc_clusters END
```

Η συνάρτηση **fat_write_end()** αποτελεί τη συνάρτηση που εκκινεί τη στιγμή που τελειώνει η εκτέλεση της συνάρτησης **fat_write_begin()**, δηλαδή **αφότου γραφούν** οι μεταβολές των πεδίων στο σύστημα FAT στο journal. Το inode του εκάστοτε αρχείου αναγνωρίζεται πλέον ως dirty, δηλαδή ότι τα δεδομένα του **έχουν εγγραφεί** στο **σύστημα αρχείων FAT** και, επομένως, μπορεί να αποδεσμευτεί από τη μνήμη του συστήματος. Η καταγραφή αυτής της κατάστασης **κρίνεται σημαντική** για το λόγο ότι το σύστημα πρέπει να είναι σε θέση να **προσαρμοστεί ανάλογα** σε περίπτωση κατάρρευσης, αξιοποιώντας πεδία αυτού του τύπου για να αποφασίσει κατάλληλα για την αποθήκευση των δεδομένων.

Από τα πεδία του struct inode, επιλέχθηκε η καταγραφή των ακολούθων από αυτών:

- Τα πεδία **i_flags και i_state**, τα οποία σε **περίπτωση κατάρρευσης του συστήματος** μπορούν να χρησιμοποιηθούν για την επαναφορά του συστήματος σε **προγενέστερη κατάσταση**, αλλά και για τον χειρισμό των δεδομένων της δομής του inode που δεν έχουν γραφεί στο FAT
- Τα πεδία **i_atime, i_mtime και i_ctime**, τα οποία ανατίθενται κάθε φορά στη τρέχουσα **χρονική στιγμή**, όπως παρουσιάζεται στο κώδικα που προηγείται της **δομής if** της δεύτερης εικόνας. Η καταγραφή τους συνεισφέρει στο να είναι **γνωστή η χρονική στιγμή** που τροποποιήθηκε η δομή inode
- Τα πεδία **dirtied_when και dirtied_time_when**, που σηματοδοτούν τη χρονική στιγμή κατά την οποία τα δεδομένα του inode του αρχείου καταγράφηκαν στο σύστημα αρχεών FAT

Στις εικόνες που ακολουθούν, παρατίθεται ο κώδικας με τον οποίο πραγματοποιείται άνοιγμα του αρχείου καταγραφής (journal) με τη κλήση συστήματος `sys_open()`, ακολουθεί ο έλεγχος ορθού ανοίγματος του και, εφόσον είναι επιτυχής, οι αλλαγές στα δεδομένα των πεδίων του inode του αρχείου καταγράφονται στο journal με τη κλήση `sys_write()` και ακολουθεί κλείσιμο του αρχείου καταγραφής με τη κλήση `sys_close()`.

inode Struct Reference

```
#include <fs.h>
```

Data Fields

```
umode_t i_mode
unsigned short i_opflags
kuid_t i_uid
kgid_t i_gid
unsigned int i_flags
const struct inode_operations * i_op
struct super_block * i_sb
struct address_space * i_mapping
unsigned long i_ino
union {
    const unsigned int i_nlink
    unsigned int __i_nlink
};
    dev_t i_rdev
    loff_t i_size
    struct timespec i_atime
    struct timespec i_mtime
    struct timespec i_ctime
    spinlock_t i_lock
    unsigned short i_bytes
    unsigned int i_blkbits
    blkcnt_t i_blocks
    unsigned long i_state
    struct rw_semaphore i_rwsem
    unsigned long dirtied_when
    unsigned long dirtied_time_when
    struct hlist_node i_hash
    struct list_head i_io_list
    struct list_head i_lru
    struct list_head i_sb_list
    struct list_head i_wb_list
};
union {
    struct hlist_head i_dentry
    struct rcu_head i_rcu
};
    u64 i_version
    atomic_t i_count
    atomic_t i_dio_count
    atomic_t i_writecount
const struct file_operations * i_fop
struct file_lock_context * i_flotx
struct address_space i_data
    struct list_head i_devices
```

```

294 static int fat_write_end(struct file *file, struct address_space *mapping,
295                         loff_t pos, unsigned len, unsigned copied,
296                         struct page *pagep, void *fsdata)
297 {
298     printk(KERN_INFO "file: inode.c fat_write_end START\n");
299     struct inode *inode = mapping->host;
300     int err;
301     err = generic_write_end(file, mapping, pos, len, copied, pagep, fsdata);
302     if (err < len)
303         fat_write_failed(mapping, pos + len);
304     if (!(err < 0) && !(MSDOS_I(inode)->i_attrs & ATTR_ARCH)) {
305         inode->i_mtime = inode->i_ctime = current_time(inode);
306         MSDOS_I(inode)->i_attrs |= ATTR_ARCH;
307         mark_inode_dirty(inode);
308
309         int journal;
310         int file_opened;
311
312         printk(KERN_INFO "\n");
313         printk(KERN_INFO "Journal opens to write cluster data\n");
314     }

```

```

336     if (file_opened){
337         printk(KERN_INFO " ----- WRITING INODE INFO AFTER MARKED DIRTY START ----- \n");
338         sprintf(file_buf,"inode->i_flags %u",inode->i_flags);
339         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
340         sys_fsync(journal);
341         sys_fdatasync(journal);
342         printk(KERN_INFO "%s \n",file_buf);
343
344         sprintf(file_buf,"inode->i_state %u",inode->i_state);
345         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
346         sys_fsync(journal);
347         sys_fdatasync(journal);
348         printk(KERN_INFO "%s \n",file_buf);
349
350         sprintf(file_buf,"inode->i_atime.tv_nsec %f",inode->i_atime.tv_nsec);
351         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
352         sys_fsync(journal);
353         sys_fdatasync(journal);
354         printk(KERN_INFO "%s \n",file_buf);
355
356         sprintf(file_buf,"inode->i_mtime.tv_nsec %f",inode->i_mtime.tv_nsec);
357         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
358         sys_fsync(journal);
359         sys_fdatasync(journal);
360         printk(KERN_INFO "%s \n",file_buf);
361
362         sprintf(file_buf,"inode->i_ctime.tv_nsec %f",inode->i_ctime.tv_nsec);
363         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
364         sys_fsync(journal);
365         sys_fdatasync(journal);
366         printk(KERN_INFO "%s \n",file_buf);
367
368         sprintf(file_buf,"inode->dirtied_when %f",inode->dirtied_when);
369         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
370         sys_fsync(journal);
371         sys_fdatasync(journal);
372         printk(KERN_INFO "%s \n",file_buf);
373
374         sprintf(file_buf,"inode->dirtied_time_when %f",inode->dirtied_time_when);
375         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
376         sys_fsync(journal);
377         sys_fdatasync(journal);
378         printk(KERN_INFO "%s \n",file_buf);
379
380         printk(KERN_INFO " ----- WRITING INODE INFO AFTER MARKED DIRTY END ----- \n");
381     }
382
383     printk(KERN_INFO "\n");
384
385     printk(KERN_INFO "----- file: inode.c fat_write_end END ----- \n\n");
386     return err;
387 }
388
389     sys_fsync(journal);
390     sys_fdatasync(journal);
391     printk(KERN_INFO "%s \n",file_buf);
392
393     sprintf(file_buf,"inode->dirtied_when %f",inode->dirtied_when);
394     sys_write(journal,file_buf,FILE_BUFFER_SIZE);
395     sys_fsync(journal);
396     sys_fdatasync(journal);
397     printk(KERN_INFO "%s \n",file_buf);
398
399     sprintf(file_buf,"inode->dirtied_time_when %f",inode->dirtied_time_when);
400     sys_write(journal,file_buf,FILE_BUFFER_SIZE);
401     sys_fsync(journal);
402     sys_fdatasync(journal);
403     printk(KERN_INFO "%s \n",file_buf);
404
405     printk(KERN_INFO " ----- WRITING INODE INFO AFTER MARKED DIRTY END ----- \n");
406 }
407
408     printk(KERN_INFO "\n");
409
410     printk(KERN_INFO "----- file: inode.c fat_write_end END ----- \n\n");
411     return err;
412 }

```

Αρχείο fatent.c

Στο αρχείο **fatent.c** περιέχονται συναρτήσεις της βιβλιοθήκης LKL οι οποίες **χειρίζονται τις εγγραφές της δομής File Allocation Table (FAT)**. Η δομή αυτή είναι οργανωμένη σε **clusters**, στα οποία αποθηκεύονται τα δεδομένα των αρχείων του συστήματος αρχείων FAT.

Κάθε cluster αποτελεί μία εγγραφή των **12,16 ή 32 bits**, εξαρτώμενο άμεσα από το τύπο του συστήματος αρχείων FAT (FAT12, FAT16 ή FAT32), και καθένα από αυτά έχει ένα δείκτη (**pointer**) στο **επόμενο cluster**, το οποίο ενδέχεται να βρίσκεται σε **μη-διαδοχική** θέση μνήμης της διαμέρισης του δίσκου του συστήματος αρχείων FAT.

Στη διπλανή εικόνα παρουσιάζονται τα structs που αφορούν τις εγγραφές στο **File Allocation Table (FAT)**.

Σε κάθε μία από τις συναρτήσεις που αποτελούν πεδία τους, έχει χρησιμοποιηθεί η κλήση **printf()**, ενώ σε όσα σημεία κρίθηκε απαραίτητο, έχουν χρησιμοποιηθεί οι κλήσεις **sys_open()**, **sys_write()**, **sys_fsync()** και **sys_fdatasync()**

```
static const struct fatent_operations fat12_ops = {
    .ent_blocknr    = fat12_ent_blocknr,
    .ent_set_ptr   = fat12_ent_set_ptr,
    .ent_bread     = fat12_ent_bread,
    .ent_get       = fat12_ent_get,
    .ent_put       = fat12_ent_put,
    .ent_next      = fat12_ent_next,
};

static const struct fatent_operations fat16_ops = {
    .ent_blocknr    = fat_ent_blocknr,
    .ent_set_ptr   = fat16_ent_set_ptr,
    .ent_bread     = fat_ent_bread,
    .ent_get       = fat16_ent_get,
    .ent_put       = fat16_ent_put,
    .ent_next      = fat16_ent_next,
};

static const struct fatent_operations fat32_ops = {
    .ent_blocknr    = fat_ent_blocknr,
    .ent_set_ptr   = fat32_ent_set_ptr,
    .ent_bread     = fat_ent_bread,
    .ent_get       = fat32_ent_get,
    .ent_put       = fat32_ent_put,
    .ent_next      = fat32_ent_next,
};
```

Στη παρούσα εργαστηριακή άσκηση, **παρατηρήθηκε** μέσω των **διαγνωστικών μηνυμάτων** από τη κλήση **printf()** ότι το σύστημα αρχείων που χρησιμοποιείται είναι το **FAT16**, καθώς τα διαγνωστικά μηνύματα από τις συναρτήσεις των άλλων δύο τύπων FAT δεν εκτυπώνονται.

Ωστόσο, έχει προστεθεί κώδικας που καταγράφει τις αλλαγές και για τους τύπους συστήματος αρχείων **FAT12 και FAT32**, σε περίπτωση που χρησιμοποιηθεί κάποιο από αυτά αντί του συστήματος αρχείων FAT16.

Στο σημείο αυτό θα παρουσιαστεί ο κώδικας που προστέθηκε στις συναρτήσεις του **struct fat16_ops** και στη συνέχεια θα παρουσιαστεί και ο κώδικας των υπόλοιπων structs.

Συνάρτησεις fat_ent_blocknr() και fat12_ent_blocknr()

Ως προς τον κώδικα τους οι συναρτήσεις **fat_ent_blocknr()** και **fat12_ent_blocknr()** παρέχουν πληροφορίες σχετικές με τον **αριθμό** του τρέχοντος cluster που πραγματοποιείται εγγραφή. Παράλληλα, παρέχεται η πληροφορία ότι στο **superblock** του παρόντος αρχείου ενημερώνεται το **πεδίο free_clusters**, δηλαδή μετά την εγγραφή του αρχείου, **πόσα από τα clusters** της διαμέρισης του δίσκου του συστήματος αρχείων FAT είναι **ακόμα διαθέσιμα**, με συνέπεια να είναι κρίσιμη η καταγραφή του. Εξίσου σημαντική κρίθηκε η καταγραφή του πεδίου **free_clus_valid**, το οποίο τίθεται στη **τιμή 0** αφότου γίνει η εγγραφή σε αυτό το cluster, με βάση τα διαγνωστικά μηνύματα. Τα πεδία αυτά είναι εμφανή στην εικόνα της επόμενης σελίδα, όπου παρουσιάζονται τα πεδία του struct msdos_sb_info.

Ο κώδικα που έχει προστεθεί στις συνάρτηση **fat_ent_blocknr()** και **fat12_ent_blocknr()** ακολουθεί τα ίδια βήματα που αναφέρθηκαν και στις προηγούμενες συναρτήσεις, σχετικά με τη καταγραφή των αλλαγών στο αρχείο καταγραφής **Journal** που πραγματοποιούνται στις δομές του συστήματος αρχείων και συγκεκριμένα στο **superblock** στη παρούσα συνάρτηση.

Ο ίδιος κώδικας με τη παρακάτω εικόνα χρησιμοποιήθηκε αυτούσιος και στη συνάρτηση **fat12_ent_blocknr()**, καθώς τα ονόματα των απαραίτητων πεδίων έχουν ίδια ονόματα μεταβλητών και οι λειτουργίες που πραγματοποιούνται είναι κοινές.

```
23 static void fat12_ent_blocknr(struct super_block *sb, int entry,
24 |           int *offset, sector_t *blocknr)
25 {
26     struct msdos_sb_info *sbi = MSDOS_SB(sb);
27     int bytes = entry + (entry >> 1);
28     WARN_ON(entry < FAT_START_ENT || sbi->max_cluster <= entry);
29     printk(KERN_INFO "file: fatent.c fat12_ent_blocknr() START\n");
30     *offset = bytes & (sb->s_blocksize - 1);
31     *blocknr = sbi->fat_start + (bytes >> sb->s_blocksize_bits);
32
33
34     journal = sys_open("journal", O_CREAT|O_WRONLY|O_APPEND,S_IRWXU);
35     printk(KERN_INFO "FILE DESCRIPTOR %d \n",journal);
36
37     if (journal>=0){
38         printk(KERN_INFO "file: inode.c Journal opened successfully\n");
39         //can define a macro for that
40         file_opened =1;
41     }
42     else{
43         file_opened =0;
44         printk(KERN_INFO "file: inode.c Could not open journal\n");
45     }
46     if (file_opened){
47         printk(KERN_INFO "----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START ----- \n");
48         sprintf(file_buf,"fatent->u.ent12_p %d",blocknr);
49         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
50         sys_Tsync(journal);
51         sys_fdatasync(journal);
52         printk(KERN_INFO "%s \n",file_buf);
53         printk(KERN_INFO "----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START ----- \n\n");
54
55         printk(KERN_INFO "----- WRITING SUPERBLOCK INFO REGARDING CLUSTERS START ----- \n");
56
57         sprintf(file_buf,"%s->free_clusters %d",sbi->free_clusters);
58         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
59         sys_Tsync(journal);
60         sys_fdatasync(journal);
61         printk(KERN_INFO "%s \n",file_buf);
62
63         sprintf(file_buf,"%s->free_clus_valid %u",sbi->free_clus_valid);
64         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
65         sys_Tsync(journal);
66         sys_fdatasync(journal);
67         printk(KERN_INFO "%s \n",file_buf);
68
69         printk(KERN_INFO "----- WRITING SUPERBLOCK INFO REGARDING CLUSTERS END ----- \n\n");
70     }
71     sys_close(journal);
72
73     printk(KERN_INFO "\n");
74
75     printk(KERN_INFO "file: fatent.c fat12_ent_blocknr() END\n");
76 }
```

msdos_sb_info Struct Reference

```
#include <fat.h>

Data Fields
unsigned short sec_per_clus
unsigned short cluster_bits
unsigned int cluster_size
unsigned char fats
unsigned char fat_bits
unsigned short fat_start
unsigned long fat_length
unsigned long dir_start
unsigned short dir_entries
unsigned long data_start
unsigned long max_cluster
unsigned long root_cluster
unsigned long fsinfo_sector
struct mutex fat_lock
struct mutex nfs_build_inode_lock
struct mutex s_lock
unsigned int prev_free
unsigned int free_clusters
unsigned int free_clus_valid
struct fat_mount_options options
struct nls_table * nls_disk
struct nls_table * nls_io
const void * dir_ops
int dir_per_block
int dir_per_block_bits
unsigned int vol_id
int fatent_shift
const struct fatent_operations * fatent_ops
struct inode * fat_inode
struct inode * fsinfo_inode
struct ratelimit_state ratelimit
spinlock_t inode_hash_lock
struct hlist_head inode_hashtable [FAT_HASH_SIZE]
spinlock_t dir_hash_lock
struct hlist_head dir_hashtable [FAT_HASH_SIZE]
unsigned int dirty
struct rcu_head rcu
```

```

0.016233] file: fatent.c fat_ent_blocknr() START
0.016243] FILE DESCRIPTOR 1
0.016246] file: inode.c Journal opened successfully
0.016248] ----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START -----
0.016251] fatent->u.ent12_p 4
0.016253] ----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START -----
0.016253]
0.016255] ----- WRITING SUPERBLOCK INFO REGARDING CLUSTERS START -----
0.016258] fsbi->free_clusters 4294967295
0.016263] sbi->free_clus_valid 0
0.016265] ----- WRITING SUPERBLOCK INFO REGARDING CLUSTERS END -----
0.016265]
0.016267]
0.016269] file: fatent.c fat_ent_blocknr() END

```

```

79 static void fat_ent_blocknr(struct super_block *sb, int entry,
80 | | | | | int *offset, sector_t *blocknr)
81 {
82     printk(KERN_INFO "file: fatent.c fat_ent_blocknr() START\n");
83     struct msdos_sb_info *sbi = MSDOS_SB(sb);
84     int bytes = (entry <> sbi->fatent_shift);
85     WARN_ON(entry < FAT_START_ENT || sbi->max_cluster <= entry);
86     *offset = bytes & (sb->s_blocksize - 1);
87     *blocknr = sbi->fat_start + (bytes >> sb->s_blocksize_bits);
88
89     journal = sys_open("journal", O_CREAT|O_WRONLY|O_APPEND,S_IRWXU);
90     printk(KERN_INFO "FILE DESCRIPTOR %d \n",journal);
91
92     if (journal>=0){
93         printk(KERN_INFO "file: inode.c Journal opened successfully\n");
94         //can define a macro for that
95         file_opened =1;
96     }
97     else{
98         file_opened =0;
99         printk(KERN_INFO "file: inode.c Could not open journal\n");
100    }
101    if (file_opened){
102        printk(KERN_INFO "----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START ----- \n");
103        sprintf(file_buf,"fatent->u.ent12_p %d",*blocknr);
104        sys_write(journal,file_buf,FILE_BUFFER_SIZE);
105        sys_fsync(journal);
106        sys_fdatasync(journal);
107        printk(KERN_INFO "%s \n",file_buf);
108        printk(KERN_INFO "----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START ----- \n\n");
109        printk(KERN_INFO "----- WRITING SUPERBLOCK INFO REGARDING CLUSTERS START ----- \n");
110        sprintf(file_buf,"fsbi->free_clusters %d",sbi->free_clusters);
111        sys_write(journal,file_buf,FILE_BUFFER_SIZE);
112        sys_fsync(journal);
113        sys_fdatasync(journal);
114        printk(KERN_INFO "%s \n",file_buf);
115        sprintf(file_buf,"sbi->free_clus_valid %u",sbi->free_clus_valid);
116        sys_write(journal,file_buf,FILE_BUFFER_SIZE);
117        sys_fsync(journal);
118        sys_fdatasync(journal);
119        printk(KERN_INFO "%s \n",file_buf);
120        printk(KERN_INFO "----- WRITING SUPERBLOCK INFO REGARDING CLUSTERS END ----- \n\n");
121    }
122    sys_close(journal);
123
124    printk(KERN_INFO "\n");
125    printk(KERN_INFO "file: fatent.c fat_ent_blocknr() END\n");
126}

```

Συναρτήσεις fat12_ent_set_ptr() και fat16_ent_set_ptr()

Οι συναρτήσεις **fat12_ent_set_ptr()** και **fat16_ent_set_ptr()** προσδιορίζουν τον pointer του cluster στο οποίο επρόκειτο να γίνει μία εγγραφή στο **File Allocation Table**. Ο pointer αυτός κρίθηκε απαραίτητος προς καταγραφή, καθώς σε περίπτωση κατάρρευσης του συστήματος, το σύστημα αρχείων FAT πρέπει να βρίσκεται σε θέση να γνωρίζει **σε ποια clusters έχει πραγματοποιήσει εγγραφές** και να προσαρμόσει ανάλογα τη συμπεριφορά του, ώστε να **μην αλλοιωθούν τα δεδομένα** που έχουν **ήδη καταγραφεί**.

Ο κώδικας καταγραφής των αλλαγών του **struct fatent**, που αντιπροσωπεύει τις εγγραφές στο **File Allocation Table**, είναι κοινός και για τις δύο συναρτήσεις. Η μεταξύ τους διαφοροποίηση είναι ότι στη συνάρτηση **fat12_ent_set_ptr()** γίνεται εγγραφή του πεδίου **fatent → u.ent12_p**, ενώ στη συνάρτηση **fat16_ent_set_ptr()** γίνεται εγγραφή του πεδίου **fatent → u.ent16_p**.

```

static void fat12_ent_set_ptr(struct fat_entry *fatent, int offset)
{
    struct buffer_head **bhs = fatent->bhs;
    printk(KERN_INFO "file: fatent.c fat12_ent_set_ptr() START\n");
    if (fatent->r.bhs == 1) {
        WARN_ON(offset >= (bhs[0]->b_size - 1));
        fatent->u.ent12_p[0] = bhs[0]->b_data + offset;
        fatent->u.ent12_p[1] = bhs[0]->b_data + (offset + 1);
    } else {
        WARN_ON(offset != (bhs[0]->b_size - 1));
        fatent->u.ent12_p[0] = bhs[0]->b_data + offset;
        fatent->u.ent12_p[1] = bhs[1]->b_data;
    }

    journal = sys_open("journal", O_CREAT|O_WRONLY|O_APPEND,S_IRWXU);
    printk(KERN_INFO "FILE DESCRIPTOR %d\n", journal);

    if (journal>=0){
        printk(KERN_INFO "file: inode.c Journal opened successfully\n");
        //can define a macro for that
        file_opened =1;
    }
    else{
        file_opened =0;
        printk(KERN_INFO "file: inode.c Could not open journal\n");
    }

    if (file_opened){
        printk(KERN_INFO "..... WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START ..... \n");
        sprintf(file_buf, "fatent->u.ent12_p %u", fatent->u.ent12_p);
        sys_write(journal,file_buf,FILE_BUFFER_SIZE);
        sys_fsync(journal);
        sys_fdatasync(journal);
        printk(KERN_INFO "%s\n",file_buf);
        printk(KERN_INFO "..... WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START ..... \n\n");
    }
    sys_close(journal);

    printk(KERN_INFO "\n");

    printk(KERN_INFO "file: fatent.c fat12_ent_set_ptr() END\n");
}

static void fat16_ent_set_ptr(struct fat_entry *fatent, int offset)
{
    WARN_ON(offset & (2 - 1));
    printk(KERN_INFO "file: fatent.c fat16_ent_set_ptr() START\n");
    fatent->u.ent16_p = (_le16 *) (fatent->bhs[0]->b_data + offset);

    journal = sys_open("journal", O_CREAT|O_WRONLY|O_APPEND,S_IRWXU);
    printk(KERN_INFO "FILE DESCRIPTOR %d\n", journal);

    if (journal>=0){
        printk(KERN_INFO "file: inode.c Journal opened successfully\n");
        //can define a macro for that
        file_opened =1;
    }
    else{
        file_opened =0;
        printk(KERN_INFO "file: inode.c Could not open journal\n");
    }

    if (file_opened){
        printk(KERN_INFO "..... WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START ..... \n");
        sprintf(file_buf, "fatent->u.ent16_p %u", fatent->u.ent16_p);
        sys_write(journal,file_buf,FILE_BUFFER_SIZE);
        sys_fsync(journal);
        sys_fdatasync(journal);
        printk(KERN_INFO "%s\n",file_buf);
        printk(KERN_INFO "..... WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START ..... \n\n");
    }
    sys_close(journal);

    printk(KERN_INFO "\n");

    printk(KERN_INFO "file: fatent.c fat16_ent_set_ptr() END\n");
}

```

The **buffer_head** structure, as found in `<linux/buffer_head.h>` in the 2.6.29 kernel.

```

/*
 * Historically, a buffer_head was used to map a single block
 * within a page, and of course as the unit of I/O through the
 * filesystem and block layers. Nowadays the basic I/O unit
 * is the bio, and buffer_heads are used for extracting block
 * mappings (via a get_block_t call), for tracking state within
 * a page (via a page_mapping) and for wrapping bio submission
 * for backward compatibility reasons (e.g. submit_bh).
 */
struct buffer_head {
    unsigned long b_state;           /* buffer state bitmap (see above) */
    struct buffer_head *b_this_page; /* circular list of page's buffers */
    struct page *b_page;            /* the page this bh is mapped to */

    sector_t b_blocknr;             /* start block number */
    size_t b_size;                 /* size of mapping */
    char *b_data;                  /* pointer to data within the page */

    struct block_device *b_bdev;    /* I/O completion */
    bh_end_io_t *b_end_io;          /* reserved for b_end_io */
    void *b_private;               /* associated with another mapping */
    struct list_head b_assoc_buffers; /* mapping this buffer is
                                         associated with */
    struct address_space *b_assoc_map; /* users using this buffer_head */
    atomic_t b_count;               /* count of references */
}

```

Από τα πεδία αυτά που παρατίθενται στη παρακάτω εικόνα, κρίθηκε απαραίτητη η καταγραφή των εξής πεδίων:

- το πεδίο **b_state**, το οποίο πρόκειται για μία **flag variable** και αφορά τη κατάσταση που βρίσκεται το cluster τη δεδομένη χρονική στιγμή
 - το πεδίο **blocknr**, δηλαδή συνολικό **μέγεθος της εγγραφής σε bits**. Κατά συνέπεια προκύπτει και ο **αριθμός των απαιτούμενων clusters** για την αποθήκευση της εγγραφής, διαιρώντας το **μέγεθος της εγγραφής** με τον **αριθμό bits** του ενός cluster.
 - Το πεδίο **b_data**, το οποίο πρόκειται για τον pointer στη θέση μνήμης που είναι αποθηκευμένα τα δεδομένα του cluster.

Ο κώδικας που είναι υπεύθυνος για τη καταγραφή των αλλαγών στις δομές του συστήματος αρχείων FAT ακολουθεί και στη περίπτωση αυτή το ήδη γνωστό pattern ως προς τις κλήσεις sys_* που περιγράφηκε νωρίτερα:

1. Άνοιγμα του αρχείου με τη κλήση sys_open()
 2. Έλεγχος ορθού ανοίγματος του αρχείου με τη χρήση της μεταβλητής file_opened
 3. Εγγραφή των δεδομένων στο αρχείο καταγραφής με τη κλήση sys_write()
 4. Μεταφορά των δεδομένων του αρχείου στο σύστημα αρχείων FAT μέσω των κλήσεων sys_fsync() και sys_fdatasync()
 5. κλείσιμο του αρχείου καταγραφής με την εντολή sys_close()

```

295     if (file_opened){
296         printk(KERN_INFO "----- WRITING CLUSTER INFO IN THE JOURNAL START-----\n");
297         sprintf(file_buf, "%tent->bhs[1]->b_state %u", fatent->bhs[1]->b_state);
298         sys_write(journal, file_buf,FILE_BUFFER_SIZE);
299         sys_fsync(journal);
300         sys_fdatasync(journal);
301         printk(KERN_INFO "%s \n",file_buf);
302
303         sprintf(file_buf, "%tent->bhs[1]->b_blocknr %d", fatent->bhs[1]->b_blocknr);
304         sys_write(journal, file_buf,FILE_BUFFER_SIZE);
305         sys_fsync(journal);
306         sys_fdatasync(journal);
307         printk(KERN_INFO "%s \n",file_buf);
308
309         sprintf(file_buf, "%tent->bhs[1]->b_size %d", fatent->bhs[1]->b_size);
310         sys_write(journal, file_buf,FILE_BUFFER_SIZE);
311         sys_fsync(journal);
312         sys_fdatasync(journal);
313         printk(KERN_INFO "%s \n",file_buf);
314
315         sprintf(file_buf, "%tent->bhs[1]->b_data %p", fatent->bhs[1]->b_data);
316         sys_write(journal, file_buf,FILE_BUFFER_SIZE);
317         sys_fsync(journal);
318         sys_fdatasync(journal);
319         printk(KERN_INFO "%s \n",file_buf);
320
321     sys_close(journal);
322     printk(KERN_INFO "----- WRITING CLUSTER INFO IN THE JOURNAL END-----\n");
323
324     return 0;
325
326 err_brelse:
327     brelse(bhs[0]);
328 err:
329     fat_msg_sb, KERN_ERR, "FAT read failed (blocknr %llu)", (llu)blocknr);
330     return -EIO;
331 }

```

```

361     if (file_opened){
362         printk(KERN_INFO "----- WRITING CLUSTER INFO IN THE JOURNAL START-----\n");
363         sprintf(file_buf, "%tent->bhs[0]->b_state %u", fatent->bhs[0]->b_state);
364         sys_write(journal, file_buf,FILE_BUFFER_SIZE);
365         sys_fsync(journal);
366         sys_fdatasync(journal);
367         printk(KERN_INFO "%s \n",file_buf);
368
369         sprintf(file_buf, "%tent->bhs[0]->b_blocknr %d", fatent->bhs[0]->b_blocknr);
370         sys_write(journal, file_buf,FILE_BUFFER_SIZE);
371         sys_fsync(journal);
372         sys_fdatasync(journal);
373         printk(KERN_INFO "%s \n",file_buf);
374
375         sprintf(file_buf, "%tent->bhs[0]->b_size %d", fatent->bhs[0]->b_size);
376         sys_write(journal, file_buf,FILE_BUFFER_SIZE);
377         sys_fsync(journal);
378         sys_fdatasync(journal);
379         printk(KERN_INFO "%s \n",file_buf);
380
381         sprintf(file_buf, "%tent->bhs[0]->b_data %p", fatent->bhs[0]->b_data);
382         sys_write(journal, file_buf,FILE_BUFFER_SIZE);
383         sys_fsync(journal);
384         sys_fdatasync(journal);
385         printk(KERN_INFO "%s \n",file_buf);
386
387     sys_close(journal);
388     printk(KERN_INFO "----- WRITING CLUSTER INFO IN THE JOURNAL END-----\n");
389
390     if (!fatent->bhs[0]) {
391         fat_msg_sb, KERN_ERR, "FAT read failed (blocknr %llu)",
392         (llu)blocknr);
393         return -EIO;
394     }
395     fatent->nr_bhs = 1;
396     ops->ent_set_ptr(fatent, offset);
397     printk(KERN_INFO "file: fatent.c fat_ent_bread() END\n");
398     return 0;
399 }

```

Συναρτήσεις fat12_ent_get(),fat16_ent_get() και fat32_ent_get()

Οι συναρτήσεις fat12_ent_get(),fat16_ent_get() και fat32_ent_get() χρησιμοποιούνται από το σύστημα αρχείων FAT με σκοπό την ανάθεση του αριθμού που προσδιορίζει το επόμενο cluster. Στο σημείο αυτό η καταγραφή του αριθμού αυτού κρίθηκε μη αναγκαία, καθώς οι Συναρτήσεις fat12_set_ptr(),fat16_set_ptr() και fat32_set_ptr() που αναφέρθηκαν νωρίτερα προσδιορίζουν ακριβώς τον pointer στη θέση μνήμης του cluster και έχουν καταγραφεί στο σημείο αυτό.

Επομένως, οι κλήσεις printk() στο σημείο αυτό έχουν τοποθετηθεί για τη κατανόηση της αλληλεπίδρασης των κλήσεων της δομής File Allocation Table με τις υπόλοιπες δομές.

```

401 static int fat12_ent_get(struct fat_entry *fatent)
402 {
403     u8 **ent12_p = fatent->u.ent12_p;
404     int next;
405     printk(KERN_INFO "file: fatent.c fat12_ent_get() START\n");
406     spin_lock(&fat12_entry_lock);
407     if (fatent->entry & 1)
408         next = (*ent12_p[0] >> 4) | (*ent12_p[1] << 4);
409     else
410         next = (*ent12_p[1] << 8) | *ent12_p[0];
411     spin_unlock(&fat12_entry_lock);
412
413     next &= 0x0fff;
414     if (next >= BAD_FAT12)
415         next = FAT_ENT_EOF;
416     printk(KERN_INFO "file: fatent.c fat12_ent_get() END\n");
417     return next;
418 }
419
420 static int fat16_ent_get(struct fat_entry *fatent)
421 {
422     int next = le16_to_cpu(*fatent->u.ent16_p);
423     WARN_ON((unsigned long)fatent->u.ent16_p & (2 - 1));
424     printk(KERN_INFO "file: fatent.c fat16_ent_get() START\n");
425     if (next >= BAD_FAT16)
426         next = FAT_ENT_EOF;
427     printk(KERN_INFO "file: fatent.c fat16_ent_get() EOF\n");
428     printk(KERN_INFO "file: fatent.c fat16_ent_get() END\n");
429     return next;
430 }
431
432 static int fat32_ent_get(struct fat_entry *fatent)
433 {
434     int next = le32_to_cpu(*fatent->u.ent32_p) & 0xffffffff;
435     WARN_ON((unsigned long)fatent->u.ent32_p & (4 - 1));
436     printk(KERN_INFO "file: fatent.c fat32_ent_get() START\n");
437     if (next >= BAD_FAT32)
438         next = FAT_ENT_EOF;
439     printk(KERN_INFO "file: fatent.c fat32_ent_get() END\n");
440     return next;
441 }

```

Συναρτήσεις fat12_ent_get(),fat16_ent_get() και fat32_ent_get()

Οι συναρτήσεις χρησιμοποιούνται από το σύστημα αρχείων FAT προκειμένου να **επισημανθούν ως dirty** οι δομές των οποίων οι πληροφορίες έχουν καταγραφεί σ αυτό. Η καταγραφή τους κρίθηκε απαραίτητη για το λόγο ότι το σύστημα , σε περίπτωση κατάρρευσης, πρέπει να είναι ικανό να **ρυθμίσει ανάλογα τη συμπεριφορά** του, με τη γνώση εάν τα πεδία έχουν καταγραφεί ή όχι και να προβαίνει στις κατάλληλες ενέργειες.

Ο κώδικας στις παρακάτω εικόνες καταγράφει τις αλλαγές στα πεδία **fatent → bhs[0]** και **fatent → fat_inode** τα οποία επισημαίνονται ως dirty με τη κλήση της συνάρτησης **mark_buffer_dirty_inode()**

```
543 static void fat16_ent_put(struct fat_entry *fatent, int new)
544 {
545     if (new == FAT_ENT_EOF)
546         new = EOF_FAT16;
547
548     printk(KERN_INFO "file: fatent.c fat16_ent_put() START\n");
549     *fatent->u.ent16_p = cpu_to_le16(new);
550
551     printk(KERN_INFO "-----\n");
552     printk(KERN_INFO "Journal opens to write superblock data\n");
553
554     journal = sys_open("journal", O_CREAT | O_WRONLY | O_APPEND, S_IRWXU);
555     printk(KERN_INFO "FILE DESCRIPTOR %d\n", journal);
556
557     if (journal>=0){
558         printk(KERN_INFO "file: fatent.c Journal opened successfully\n");
559         //can define a macro for that
560         file_opened =1;
561     }
562     else{
563         file_opened =0;
564         printk(KERN_INFO "file: fatent.c Could not open journal\n");
565     }
566
567     // ----- ADDED BY ME -----
568     // MARKS THE inode DIRTY SO THE INODE FIELDS ARE CHANGED
569     mark_buffer_dirty_inode(fatent->bhs[0], fatent->fat_inode);
570
571 }
```

```
442 static void fat12_ent_put(struct fat_entry *fatent, int new)
443 {
444     u8 **ent12_p = fatent->u.ent12_p;
445
446     if (new == FAT_ENT_EOF)
447         new = EOF_FAT12;
448
449     printk(KERN_INFO "file: fatent.c fat12_ent_put() START\n");
450     spin_lock(&fat12_entry_lock);
451     if (fatent->entry & 1) {
452         *ent12_p[0] = (new << 4) | (*ent12_p[0] & 0x0f);
453         *ent12_p[1] = new >> 4;
454     } else {
455         *ent12_p[0] = new & 0xff;
456         *ent12_p[1] = (*ent12_p[1] & 0xf0) | (new >> 8);
457     }
458     spin_unlock(&fat12_entry_lock);
459
460     mark_buffer_dirty_inode(fatent->bhs[0], fatent->fat_inode);
461     printk(KERN_INFO "-----\n");
462     printk(KERN_INFO "Journal opens to write superblock data\n");
463
464     journal = sys_open("journal", O_CREAT | O_WRONLY | O_APPEND, S_IRWXU);
465     printk(KERN_INFO "FILE DESCRIPTOR %d\n", journal);
466
467     if (journal>=0){
468         printk(KERN_INFO "file: fatent.c Journal opened successfully\n");
469         //can define a macro for that
470         file_opened =1;
471     }
472     else{
473         file_opened =0;
474         printk(KERN_INFO "file: fatent.c Could not open journal\n");
475     }
476
477 }
```

```
572
573     if (file_opened){
574         printk(KERN_INFO "----- WRITING CLUSTER INFO IN THE JOURNAL START-----\n");
575         printk(KERN_INFO "-----AFTER MARKED DIRTY-----\n");
576         sprintf(file_buf, "fatent->bhs[0]->b state %u", fatent->bhs[0]->b_state);
577         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
578         sys_fsync(journal);
579         sys_fdatasync(journal);
580         printk(KERN_INFO "%s\n", file_buf);
581
582         sprintf(file_buf, "fatent->bhs[0]->b blocknr %d", fatent->bhs[0]->b_blocknr);
583         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
584         sys_fsync(journal);
585         sys_fdatasync(journal);
586         printk(KERN_INFO "%s\n", file_buf);
587
588         sprintf(file_buf, "fatent->bhs[0]->b size %d", fatent->bhs[0]->b_size);
589         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
590         sys_fsync(journal);
591         sys_fdatasync(journal);
592         printk(KERN_INFO "%s\n", file_buf);
593
594         sprintf(file_buf, "fatent->bhs[0]->b data %p", fatent->bhs[0]->b_data);
595         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
596         sys_fsync(journal);
597         sys_fdatasync(journal);
598         printk(KERN_INFO "%s\n", file_buf);
599
600         sprintf(file_buf, "fatent->fat_inode->i mode %u", fatent->fat_inode->i_mode);
601         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
602         sys_fsync(journal);
603         sys_fdatasync(journal);
604         printk(KERN_INFO "%s\n", file_buf);
605
606         sprintf(file_buf, "fatent->fat_inode->i state %u", fatent->fat_inode->i_state);
607
608
609         sprintf(file_buf, "fatent->fat_inode->i flags %u", fatent->fat_inode->i_flags);
610         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
611         sys_fsync(journal);
612         sys_fdatasync(journal);
613         printk(KERN_INFO "%s\n", file_buf);
614
615     }
616
617     printk(KERN_INFO "-----\n");
618     sys_close(journal);
619
620     printk(KERN_INFO "file: fatent.c fat16_ent_put() END\n");
621
622 }
```

```
479
480     if (file_opened){
481         printk(KERN_INFO "----- WRITING CLUSTER INFO IN THE JOURNAL START-----\n");
482         printk(KERN_INFO "-----AFTER MARKED DIRTY-----\n");
483         sprintf(file_buf, "fatent->bhs[0]->b state %u", fatent->bhs[0]->b_state);
484         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
485         sys_fsync(journal);
486         sys_fdatasync(journal);
487         printk(KERN_INFO "%s\n", file_buf);
488
489         sprintf(file_buf, "fatent->bhs[0]->b blocknr %d", fatent->bhs[0]->b_blocknr);
490         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
491         sys_fsync(journal);
492         sys_fdatasync(journal);
493         printk(KERN_INFO "%s\n", file_buf);
494
495         sprintf(file_buf, "fatent->bhs[0]->b size %d", fatent->bhs[0]->b_size);
496         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
497         sys_fsync(journal);
498         sys_fdatasync(journal);
499         printk(KERN_INFO "%s\n", file_buf);
500
501         sprintf(file_buf, "fatent->bhs[0]->b data %p", fatent->bhs[0]->b_data);
502         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
503         sys_fsync(journal);
504         sys_fdatasync(journal);
505         printk(KERN_INFO "%s\n", file_buf);
506
507         sprintf(file_buf, "fatent->fat_inode->i mode %u", fatent->fat_inode->i_mode);
508         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
509         sys_fsync(journal);
510         sys_fdatasync(journal);
511         printk(KERN_INFO "%s\n", file_buf);
512
513         sprintf(file_buf, "fatent->fat_inode->i state %u", fatent->fat_inode->i_state);
514         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
515         sys_fsync(journal);
516         sys_fdatasync(journal);
517         printk(KERN_INFO "%s\n", file_buf);
518
519         sprintf(file_buf, "fatent->fat_inode->i flags %u", fatent->fat_inode->i_flags);
520         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
521         sys_fsync(journal);
522         sys_fdatasync(journal);
523         printk(KERN_INFO "%s\n", file_buf);
524
525         sprintf(file_buf, "fatent->fat_inode->dirtied when %u", fatent->fat_inode->i_flags);
526         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
527         sys_fsync(journal);
528         sys_fdatasync(journal);
529         printk(KERN_INFO "%s\n", file_buf);
530
531         sprintf(file_buf, "fatent->fat_inode->dirtied_time_when %u", fatent->fat_inode->i_flags);
532         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
533         sys_fsync(journal);
534         sys_fdatasync(journal);
535         printk(KERN_INFO "%s\n", file_buf);
536
537 }
```

```

536     printk(KERN_INFO "-----\n");
537     sys_close(journal);
538     if (fatent->nr_bhs == 2)
539         mark_buffer_dirty_inode(fatent->bhs[1], fatent->fat_inode);
540     printk(KERN_INFO "file: inode.c fat12_ent_put() END\n");
541 }

```

```

636 static void fat32_ent_put(struct fat_entry *fatent, int new)
637 {
638     WARN_ON(new & 0xf0000000);
639     new |= le32_to_cpu(*fatent->u.ent32_p) & ~0x0fffffff;
640     *fatent->u.ent32_p = cpu_to_le32(new);
641     mark_buffer_dirty_inode(fatent->bhs[0], fatent->fat_inode);
642     printk(KERN_INFO "-----\n");
643     printk(KERN_INFO "Journal opens to write superblock data\n");
644
645
646     journal = sys_open("journal", O_CREAT|O_WRONLY|O_APPEND, S_IRWXU);
647     printk(KERN_INFO "FILE DESCRIPTOR %d \n", journal);
648
649     if (journal>=0){
650         printk(KERN_INFO "file: fatent.c Journal opened successfully\n");
651         //can define a macro for that
652         file_opened =1;
653     }
654     else{
655         file_opened =0;
656         printk(KERN_INFO "file: fatent.c Could not open journal\n");
657     }

```

```

659     if (file_opened){
660         printk(KERN_INFO "----- WRITING CLUSTER INFO IN THE JOURNAL START-----\n");
661         printk(KERN_INFO "-----AFTER MARKED DIRTY-----\n");
662         sprintf(file_buf,"fatent->bhs[0]->b_state %u",fatent->bhs[0]->b_state);
663         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
664         sys_fsync(journal);
665         sys_fdatasync(journal);
666         printk(KERN_INFO "%s \n",file_buf);
667
668         sprintf(file_buf,"fatent->bhs[0]->b_blocknr %d",fatent->bhs[0]->b_blocknr);
669         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
670         sys_fsync(journal);
671         sys_fdatasync(journal);
672         printk(KERN_INFO "%s \n",file_buf);
673
674         sprintf(file_buf,"fatent->bhs[0]->b_size %d",fatent->bhs[0]->b_size);
675         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
676         sys_fsync(journal);
677         sys_fdatasync(journal);
678         printk(KERN_INFO "%s \n",file_buf);
679
680         sprintf(file_buf,"fatent->bhs[0]->b_data %p",fatent->bhs[0]->b_data);
681         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
682         sys_fsync(journal);
683         sys_fdatasync(journal);
684         printk(KERN_INFO "%s \n",file_buf);
685
686         sprintf(file_buf,"fatent->fat_inode->i_mode %u",fatent->fat_inode->i_mode);
687         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
688         sys_fsync(journal);
689         sys_fdatasync(journal);
690         printk(KERN_INFO "%s \n",file_buf);
691
692         sprintf(file_buf,"fatent->fat_inode->i_state %u",fatent->fat_inode->i_state);
693         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
694         sys_fsync(journal);
695         sys_fdatasync(journal);
696         printk(KERN_INFO "%s \n",file_buf);
697
698         sprintf(file_buf,"fatent->fat_inode->i_flags %u",fatent->fat_inode->i_flags);
699         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
700         sys_fsync(journal);
701         sys_fdatasync(journal);
702         printk(KERN_INFO "%s \n",file_buf);
703
704         sprintf(file_buf,"fatent->fat_inode->dirtied_when %u",fatent->fat_inode->i_flags);
705         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
706         sys_fsync(journal);
707         sys_fdatasync(journal);
708         printk(KERN_INFO "%s \n",file_buf);
709
710         sprintf(file_buf,"fatent->fat_inode->dirtied_time_when %u",fatent->fat_inode->i_flags);
711         sys_write(journal,file_buf,FILE_BUFFER_SIZE);
712         sys_fsync(journal);
713         sys_fdatasync(journal);
714         printk(KERN_INFO "%s \n",file_buf);
715     }

```

```

716     printk(KERN_INFO "-----\n");
717     sys_close(journal);
718 }

```

Στις εικόνες που ακολουθούν παρουσιάζεται **ο τρόπος** που ο κώδικας της βιβλιοθήκης LKL **αξιοποιεί τις συναρτήσεις** του αρχείου **fatent.c** ως εξής:

Αρχικά, η **έναρξης της πραγματοποίησης των εγγραφών** στο σύστημα αρχείων FAT γίνεται με τη συνάρτηση **fat_write_begin()** του αρχείου **inode.c**, η οποία αναφέρθηκε στο σχετικό κομμάτι της αναφοράς.

Για **κάθε cluster** που θα δεσμευτεί για την εγγραφή, καλείται η συνάρτηση **fat_add_cluster()** και στη συνέχεια ο έλεγχος των για τη πραγματοποίηση των εγγραφών ανατίθεται στη **δομή File Allocation Table** με τη χρήση των συναρτήσεων του αρχείου **fatent.c**.

Συγκεκριμένα, καλείται η συνάρτηση **fat_ent_blocknr()** η οποία δείχνει σε ποιο cluster έγινε εγγραφή τη δεδομένη χρονική στιγμή, **πόσα clusters** είναι ακόμα **διαθέσιμα προς εγγραφή** και την **ενημερωμένη κατάσταση** του τρέχοντος cluster, δηλαδή εάν είναι έγκυρο προς εγγραφή ή όχι.

Στη συνέχεια γίνεται κλήση της συνάρτησης **fat_ent_bread()** η οποία δίνει πληροφορίες σχετικές με τη κατάσταση του cluster που είναι πεδία του **struct buffer_head**, το block από το οποίο ξεκινά η εγγραφή, το μέγεθος της εγγραφής που συνεπάγεται και το πλήθος των clusters που έχουν δεσμευτεί για αυτή αλλά και τον pointer στο τρέχον cluster.

Ακολουθεί κλήση της συνάρτησης **fat16_ent_set_ptr()** και **fat16_ent_get()** με την χρήση των οποίων το σύστημα αρχείων FAT **ενημερώνεται** για τον **pointer του επόμενου cluster** και η διαδικασία της εγγραφής σε ένα cluster ολοκληρώνεται με τη κλήση της κλήση της συνάρτησης **fat16_ent_put()**, επισημαίνοντας **ως dirty το cluster αυτό**, αφότου **ολοκληρωθεί** η εγγραφή.

Η κλήσεις αυτές **γίνονται επαναληπτικά** μέχρις ότου να **καταχωρηθεί το αρχείο στο σύστημα αρχείων FAT**, δηλαδή να γίνουν εγγραφές σε ένα πεπερασμένο αριθμό από clusters.

Αξίζει να επισημανθεί ότι η ομάδα από κλήσεις που αναφέρθηκε αφορά το σύστημα αρχείων **FAT16** που παρατηρήθηκε ότι χρησιμοποιείται στη παρούσα εργαστηριακή άσκηση. Οι κλήσεις αυτές ενδεχομένως να διαφοροποιούνται για τα υπόλοιπα συστήματα αρχείων αλλά σε κάθε περίπτωση οι κλήσεις αυτές θα είναι από τον κώδικα του αρχείου fatent.c

```

0.019447] ----- file: inode.c fat_write_begin START -----
0.019451] file: inode.c fat_add_cluster START
0.019453] file: fatent.c fat_alloc_clusters START
0.019455]
0.019457] Journal opens to write cluster data
0.019459] FILE DESCRIPTOR 0
0.019469] file: fatent.c Journal opened successfully
0.019471] ----- WRITING WHICH FAT ENTRY WAS PARSED START -----
0.019473] fantent.entry 3
0.019476] ----- WRITING WHICH FAT ENTRY WAS PARSED END -----
0.019480] file: fatent.c fat_ent_blocknr() START
0.019489] FILE DESCRIPTOR 2
0.019492] file: inode.c Journal opened successfully
0.019494] ----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START -----
0.019496] fatent->u.ent12_p 4
0.019498] ----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START -----
0.019503] ----- WRITING SUPERBLOCK INFO REGARDING CLUSTERS START -----
0.019505] fsbi->free_clusters 1
0.019508] sbi->free_clusters valid 0
0.019510] ----- WRITING SUPERBLOCK INFO REGARDING CLUSTERS END -----
0.019510]
0.019514] file: fatent.c fat_ent_blocknr() END
0.019516]
0.019518] Journal opens to write superblock data
0.019528] FILE DESCRIPTOR 2
0.019530] file: fatent.c Journal opened successfully
0.019532]
0.019534] file: fatent.c fat_ent_bread() START
0.019536] fatent->bhs[0]->b_state 43
0.019539] fatent->bhs[0]->b_blocknr 4
0.019544] fatent->bhs[0]->b_size 512
0.019547] fatent->bhs[0]->b_data 000007fcc05c00800
0.019549] ----- WRITING CLUSTER INFO IN THE JOURNAL END-----
0.019551] file: fatent.c fat16_ent_set_ptr() START
0.019552]
0.019562] FILE DESCRIPTOR 2
0.019564] file: inode.c Journal opened successfully
0.019566] ----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START -----
0.019569] fatent->u.ent16_p 96471046
0.019571] ----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START -----
0.019571]
0.019573] file: fatent.c fat16_ent_set_ptr() END

```

```

0.019577] file: fatent.c fat_ent_bread() END
0.019577]
0.019579] file: fatent.c fat16_ent_get() START
0.019581] file: fatent.c fat16_ent_get() EOF
0.019583] file: fatent.c fat16_ent_get() END
0.019589] file: fatent.c fat16_ent_put() START
0.019592] -----
0.019594] Journal opens to write superblock data
0.019603] FILE DESCRIPTOR 2
0.019606] file: fatent.c Journal opened successfully
0.019608] ----- WRITING CLUSTER INFO IN THE JOURNAL START-----
0.019610] -----AFTER MARKED DIRTY-----
0.019612] fatent->bhs[0]->b_state 43
0.019615] fatent->bhs[0]->b_blocknr 4
0.019618] fatent->bhs[0]->b_size 512
0.019621] fatent->bhs[0]->b_data 000007fcc05c00800
0.019623] fatent->fat_inode->i_mode 0
0.019626] fatent->fat_inode->i_state 0
0.019629] fatent->fat_inode->i_flags 0
0.019631] fatent->fat_inode->dirtied_when 0
0.019634] fatent->fat_inode->dirtied_time_when 0
0.019636] -----
0.019638] file: fatent.c fat16_ent_put() END
0.019638]
0.019641] ----- WRITING LAST FAT ENTRY THAT WAS INSERTED START -----
0.019643] sbi->prev_free 3
0.019645] ----- WRITING LAST FAT ENTRY THAT WAS INSERTED END -----
0.019648] file: fatent.c fat_alloc_clusters END
0.019650] file: inode.c fat_add_cluster END
0.019652] file: inode.c fat_add_cluster START
0.019655] file: fatent.c fat_alloc_clusters START
0.019656]
0.019658] Journal opens to write cluster data
0.019668] FILE DESCRIPTOR 0
0.019670] file: fatent.c Journal opened successfully
0.019672]
0.019674] ----- WRITING WHICH FAT ENTRY WAS PARSED START -----
0.019677] fantent.entry 4
0.019679] ----- WRITING WHICH FAT ENTRY WAS PARSED END -----
0.019681] file: fatent.c fat_ent_blocknr() START

```

Αρχείο namei_vfat.c

Στο αρχείο **namei_vfat.c** ορίζονται οι λειτουργίες που σχετίζονται με τους καταλόγους (**directories**), καθώς και με λειτουργίες του συστήματος **VFAT**. Οι λειτουργίες αυτές συνιστούν πεδία του **struct vfat_dir_inode_operations** και πρόκειται για τις εξής συναρτήσεις που παρουσιάζονται στη παρακάτω εικόνα

```

1130 static const struct inode_operations vfat_dir_inode_operations = {
1131     .create    = vfat_create,
1132     .lookup    = vfat_lookup,
1133     .unlink    = vfat_unlink,
1134     .mkdir     = vfat_mkdir,
1135     .rmdir     = vfat_rmdir,
1136     .rename    = vfat_rename,
1137     .setattr   = fat_setattr,
1138     .getattr   = fat_getattr,
1139 };
1140

```

Συνάρτηση vfat_create()

Η συνάρτηση **vfat_create()** είναι υπεύθυνη για την αρχικοποίηση του inode του αρχείου καθώς και για το **directory entry (dentry)** του.
Ο κώδικας που προϊδεάζει για τις ενέργειες αυτές είναι:

- η συνάρτηση **vfat_add_entry()** στη γραμμή 815
- η συνάρτηση **fat_build_inode()** στη γραμμή 820
- η συνάρτηση **d_instantiate()** στη γραμμή 831

Συνεπώς, οι αλλαγές που κρίθηκαν απαραίτητες προς καταγραφή είναι αυτές που πραγματοποιούνται στα structs dentry και inode.

```

789 static int vfat_create(struct inode *dir, struct dentry *dentry, umode_t mode,
790                      :           bool excl)
791 {
792     struct super_block *sb = dir->i_sb;
793     struct inode *inode;
794     struct fat_slot_info sinfo;
795     struct timespec ts;
796     int err;
797     printk(KERN_INFO "file: namei_fat.c vfat_create START\n\n");
798     inode_journal = sys_open("journal", O_CREAT | O_WRONLY | O_APPEND, S_IRWXU);
799     printk(KERN_INFO "FILE DESCRIPTOR %d \n", inode_journal);
800
801     if (inode_journal >= 0){
802         printk(KERN_INFO "file: namei_vfat.c Journal opened successfully\n");
803         //can define a macro for that
804         file_opened = 1;
805     }
806     else{
807         file_opened = 0;
808         printk(KERN_INFO "file: namei_vfat.c Could not open journal\n");
809     }
810
811     mutex_lock(&MSDOS_SB(sb)->s_lock);
812     ts = current_time(dir);
813     err = vfat_add_entry(dir, &dentry->d_name, 0, 0, &ts, &sinfo);
814     if (err)
815         goto out;
816     dir->i_version++;
817     inode = fat_build_inode(sb, sinfo.de, sinfo.i_pos);
818     brelse(sinfo.bh);
819     if (IS_ERR(inode)) {
820         err = PTR_ERR(inode);
821         goto out;
822     }
823     inode->i_version++;
824     inode->i_mtime = inode->i_atime = inode->i_ctime = ts;
825     /* timestamp is already written, so mark_inode_dirty() is unneeded. */
826

```

```

831     d_instantiate(dentry, inode);
832
833     if (file_opened){
834         sprintf(file_buf,"inode->i_version %u",inode->i_version);
835         sys_write(inode_journal,file_buf,FILE_BUFFER_SIZE);
836         sys_fsync(inode_journal);
837         sys_fdatasync(inode_journal);
838         printk(KERN_INFO "%s \n",file_buf);
839
840         sprintf(file_buf,"inode->i_mtime %hd",inode->i_mtime.tv_nsec);
841         sys_write(inode_journal,file_buf,FILE_BUFFER_SIZE);
842         sys_fsync(inode_journal);
843         sys_fdatasync(inode_journal);
844         printk(KERN_INFO "%s \n",file_buf);
845
846         sprintf(file_buf,"inode->i_atime %hd",inode->i_atime.tv_nsec);
847         sys_write(inode_journal,file_buf,FILE_BUFFER_SIZE);
848         sys_fsync(inode_journal);
849         sys_fdatasync(inode_journal);
850         printk(KERN_INFO "%s \n",file_buf); 99%
851
852         sprintf(file_buf,"inode->i_ctime %hd",inode->i_ctime.tv_nsec);
853         sys_write(inode_journal,file_buf,FILE_BUFFER_SIZE);
854         sys_fsync(inode_journal);
855         sys_fdatasync(inode_journal);
856         printk(KERN_INFO "%s \n",file_buf);
857
858         printk(KERN_INFO "----- WRITING DENTRY INFO NOW -----");
859
860         sprintf(file_buf,"dentry->d_flags %x",dentry->d_flags);
861         sys_write(inode_journal,file_buf,FILE_BUFFER_SIZE);
862         sys_fsync(inode_journal);
863         sys_fdatasync(inode_journal);
864         printk(KERN_INFO "%x \n",file_buf);
865
866         sprintf(file_buf,"dentry->d_name.name %s",dentry->d_name.name);
867         sys_write(inode_journal,file_buf,FILE_BUFFER_SIZE);
868         sys_fsync(inode_journal);
869         sys_fdatasync(inode_journal);
870         printk(KERN_INFO "%s \n",file_buf);
871
872         sprintf(file_buf,"dentry->d_iname %s",dentry->d_iname);
873         sys_write(inode_journal,file_buf,FILE_BUFFER_SIZE);
874         sys_fsync(inode_journal);
875         sys_fdatasync(inode_journal);
876         printk(KERN_INFO "%s \n",file_buf);
877
878     }
879
880     out:
881     mutex_unlock(&MSDOS_SB(sb)->s_lock);
882     printk(KERN_INFO "\nfile: namei_fat.c vfat_create END\n");
883     sys_close(inode_journal);
884     return err;
885 }
```

```

777     out:
778         mutex_unlock(&MSDOS_SB(sb)->s_lock);
779         if (!inode)
780             vfat_d_version_set(dentry, dir->i_version);
781         printk(KERN_INFO "file: namei_fat.c vfat_lookup END\n");
782         return d_splice_alias(inode, dentry);
783
784     error:
785         mutex_unlock(&MSDOS_SB(sb)->s_lock);
786         printk(KERN_INFO "file: namei_fat.c vfat_lookup END\n");
787         return ERR_PTR(err);
788 }
```

Συνάρτηση vfat_lookup()

Η συνάρτηση **vfat_lookup()** χρησιμοποιείται από το σύστημα αρχείων με σκοπό την εύρεση ενός αρχείου, καλώντας τη συνάρτηση **vfat_find()** με όρισμα το **όνομα του αρχείου**, δηλαδή το path που ακολουθεί το σύστημα για να καταλήξει σε αυτό. Η συνάρτηση αυτή δε τροποποιεί με οποιοδήποτε τρόπο τη δομή dentry και συνεπώς δεν κρίθηκε απαραίτητη η καταγραφή των αλλαγών σε αυτό το σημείο.

Ωστόσο, η χρήση της συνάρτησης vfat_find() αποτέλεσε σημείο για την εξέταση του κώδικα της.

```
727     static struct dentry *vfat_lookup(struct inode *dir, struct dentry *dentry,
728                                         unsigned int flags)
729 {
730     struct super_block *sb = dir->i_sb;
731     struct fat_slot_info sinfo;
732     struct inode *inode;
733     struct dentry *alias;
734     int err;
735     printk(KERN_INFO "file: namei_fat.c vfat_lookup START\n");
736     mutex_lock(&MSDOS_SB(sb)->s_lock);
737
738     err = vfat_find(dir, &dentry->d_name, &sinfo);
739     if (err) {
740         if (err == -ENOENT) {
741             inode = NULL;
742             goto out;
743         }
744         goto error;
745     }
```

Η συνάρτηση **vfat_find()**, όπως και η συνάρτηση **vfat_lookup()**, δεν τροποποιεί με οποιοδήποτε τρόπο τη δομή dentry και αποτελεί συνέχεια της αναζήτησης του αρχείου στο σύστημα αρχείων.

Ο ρόλος, λοιπόν, των κλήσεων printk() στις δύο αυτές συναρτήσεις είναι για καθοδήγηση στη κατανόηση της αλληλεπίδρασης του κώδικα του αρχείου με τις υπόλοιπες δομές της βιβλιοθήκης LKL.

```
707     static int vfat_find(struct inode *dir, const struct qstr *qname,
708                           struct fat_slot_info *sinfo)
709 {
710     printk(KERN_INFO "file: namei_fat.c vfat_find START\n");
711     unsigned int len = vfat_striptail_len(qname);
712     if (len == 0)
713         return -ENOENT;
714     printk(KERN_INFO "file: namei_fat.c vfat_find END\n");
715     return fat_search_long(dir, qname->name, len, sinfo);
716 }
717
```

Συνάρτηση vfat_mkdir()

Η συνάρτηση **vfat_mkdir()** χρησιμοποιείται από το σύστημα αρχείων με σκοπό την **δημιουργία ενός νέου καταλόγου (directory)** σε ένα δεδομένο **μονοπάτι (path)**. Η συνάρτηση αυτή δε τροποποιεί με οποιοδήποτε τρόπο τη δομή dentry ή τη δομή inode που δίδονται ως ορίσματα σε αυτή και συνεπώς δεν κρίθηκε απαραίτητη η καταγραφή των αλλαγών σε αυτό το σημείο.

Ο ρόλος, λοιπόν, των κλήσεων printk() στις δύο αυτές συναρτήσεις είναι για καθοδήγηση στη κατανόηση της αλληλεπίδρασης του κώδικα του αρχείου με τις υπόλοιπες δομές της βιβλιοθήκης LKL.

```

945 static int vfat_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode)
946 {
947     struct super_block *sb = dir->i_sb;
948     struct inode *inode;
949     struct fat_slot_info sinfo;
950     struct timespec ts;
951     int err, cluster;
952
953     printk(KERN_INFO "file: namei_fat.c vfat_mkdir START\n");
954     mutex_lock(&MSDOS_SB(sb)->s_lock);
955
956     ts = current_time(dir);
957     cluster = fat_alloc_new_dir(dir, &ts);
958     if (cluster < 0) {
959         err = cluster;
960         goto out;
961     }
962     err = vfat_add_entry(dir, &dentry->d_name, 1, cluster, &ts, &sinfo);
963     if (err)
964         goto out_free;
965     dir->i_version++;
966     inc_nlink(dir);
967
968     inode = fat_build_inode(sb, sinfo.de, sinfo.i_pos);
969     brelse(sinfo.bh);
970     if (IS_ERR(inode)) {
971         err = PTR_ERR(inode);
972         /* the directory was completed, just return a error */
973         goto out;
974     }
975     inode->i_version++;
976     set_nlink(inode, 2);
977     inode->i_mtime = inode->i_atime = inode->i_ctime = ts;
978     /* timestamp is already written, so mark_inode_dirty() is unneeded. */
979
980     d_instantiate(dentry, inode);
981
982     mutex_unlock(&MSDOS_SB(sb)->s_lock);
983     printk(KERN_INFO "file: namei_fat.c vfat_mkdir END\n");
984     return 0;
985
986 out_free:
987     fat_free_clusters(dir, cluster);
988 out:
989     mutex_unlock(&MSDOS_SB(sb)->s_lock);
990     return err;
991 }
```

Η συνάρτηση **vfat_rmdir()** χρησιμοποιείται από το σύστημα αρχείων με σκοπό τη **διαγραφή ενός καταλόγου (directory)**, δοθέντος του **path** που είναι αποθηκευμένος. Η συνάρτηση αυτή δε τροποποιεί με οποιοδήποτε τρόπο τη δομή dentry ή τη δομή inode που δίδονται ως ορίσματα σε αυτή και συνεπώς δεν κρίθηκε απαραίτητη η καταγραφή των αλλαγών σε αυτό το σημείο.

Ο ρόλος, λοιπόν, των κλήσεων printk() στις δύο αυτές συναρτήσεις είναι για καθοδήγηση στη κατανόηση της αλληλεπίδρασης του κώδικα του αρχείου με τις υπόλοιπες δομές της βιβλιοθήκης LKL.

```

888 static int vfat_rmdir(struct inode *dir, struct dentry *dentry)
889 {
890     struct inode *inode = d_inode(dentry);
891     struct super_block *sb = dir->i_sb;
892     struct fat_slot_info sinfo;
893     int err;
894
895     mutex_lock(&MSDOS_SB(sb)->s_lock);
896     printk(KERN_INFO "file: namei_fat.c vfat_rmdir START\n");
897     err = fat_dir_empty(inode);
898     if (err)
899         goto out;
900     err = vfat_find(dir, &dentry->d_name, &sinfo);
901     if (err)
902         goto out;
903
904     err = fat_remove_entries(dir, &sinfo); /* and releases bh */
905     if (err)
906         goto out;
907     drop_nlink(dir);
908
909     clear_nlink(inode);
910     inode->i_mtime = inode->i_atime = current_time(inode);
911     fat_detach(inode);
912     vfat_d_version_set(dentry, dir->i_version);
913
914     mutex_unlock(&MSDOS_SB(sb)->s_lock);
915     printk(KERN_INFO "file: namei_fat.c vfat_rmdir END\n");
916     return err;
917 }
```

Οι παρακάτω συναρτήσεις, παρόλο που **δεν αποτελούν πεδία του struct** **vfat_dir_inode_operations**, κρίθηκε αναγκαία η προσθήκη των κλήσεων `printk()` σε αυτές για τη κατανόηση της αλληλεπίδρασης του με το σύστημα. Στη συνάρτηση **vfat_fill_super()**, ακολουθήθηκε η πορεία του κώδικα στη συνάρτηση **fat_fill_super()** του **αρχείου inode.c** και προστέθηκε το γνωστό πλέον **μοτίβο κώδικα** που καταγράφει στο αρχείο καταγραφής **journal τις αλλαγές στα πεδία των δομών** του συστήματος αρχείων.

```

1125 static const struct inode_operations vfat_dir_inode_operations = {
1126     .create      = vfat_create,
1127     .lookup      = vfat_lookup,
1128     .unlink      = vfat_unlink,
1129     .mkdir       = vfat_mkdir,
1130     .rmdir       = vfat_rmdir,
1131     .rename      = vfat_rename,
1132     .setattr     = fat_setattr,
1133     .getattr     = fat_getattr,
1134 };
1135
1136
1137 static void setup(struct super_block *sb)
1138 {
1139     MSDOS_SB(sb)->dir_ops = &vfat_dir_inode_operations;
1140     printk(KERN_INFO "file: namei_fat.c setup\n");
1141     if (MSDOS_SB(sb)->options.name_check != 's')
1142         sb->s_d_op = &vfat_ci_dentry_ops;
1143     else
1144         sb->s_d_op = &vfat_dentry_ops;
1145 }
1146
1147 static int vfat_fill_super(struct super_block *sb, void *data, int silent)
1148 {
1149     printk(KERN_INFO "file: namei_vfat.c vfat_fill_super START");
1150     return fat_fill_super(sb, data, silent, 1, setup);
1151 }
1152
1153 static struct dentry *vfat_mount(struct file_system_type *fs_type,
1154                                   int flags, const char *dev_name,
1155                                   void *data)
1156 {
1157     printk(KERN_INFO "file: namei_vfat.c vfat_mount START");
1158     return mount_bdev(fs_type, flags, dev_name, data, vfat_fill_super);
1159 }
1160
1161 static struct file_system_type vfat_fs_type = {
1162     .owner        = THIS_MODULE,
1163     .name         = "vfat",
1164     .mount        = vfat_mount,
1165     .kill_sb      = kill_block_super,
1166     .fs_flags     = FS_REQUIRES_DEV,
1167 };

```

Συνάρτηση **fat_fill_super()**

Η συνάρτηση αυτή χρησιμοποιείται για το **προσδιορισμό της πληροφορίας** της δομής **superblock**, στην οποία είναι και οργανωμένα τα inodes του εκάστοτε αρχείου. Η καταγραφή των πληροφοριών αυτών είναι ζωτικής σημασίας για το σύστημα αρχείων FAT και, συνεπώς, έχουν προστεθεί οι κλήσεις **sys_***() για τη καταγραφή των αλλαγών στα πεδία του **superblock**.

Τα πεδία του **struct msdos_sb_info** είναι εμφανή στη παρακάτω εικόνα, ύστερα από αναζήτηση του στο **documentation** της βιβλιοθήκης LKL

msdos_sb_info Struct Reference

```
#include <fat.h>
```

Data Fields

```
    unsigned short sec_per_clus
    unsigned short cluster_bits
        unsigned int cluster_size
    unsigned char fats
    unsigned char fat_bits
    unsigned short fat_start
    unsigned long fat_length
    unsigned long dir_start
    unsigned short dir_entries
    unsigned long data_start
    unsigned long max_cluster
    unsigned long root_cluster
    unsigned long fsinfo_sector
    struct mutex fat_lock
    struct mutex nfs_build_inode_lock
    struct mutex s_lock
    unsigned int prev_free
    unsigned int free_clusters
    unsigned int free_clus_valid
    struct fat_mount_options options
        struct nls_table * nls_disk
        struct nls_table * nls_io
        const void * dir_ops
            int dir_per_block
            int dir_per_block_bits
    unsigned int vol_id
        int fatent_shift
    const struct fatent_operations * fatent_ops
        struct inode * fat_inode
        struct inode * fsinfo_inode
    struct ratelimit_state ratelimit
        spinlock_t inode_hash_lock
    struct hlist_head inode_hashtable [FAT_HASH_SIZE]
        spinlock_t dir_hash_lock
    struct hlist_head dir_hashtable [FAT_HASH_SIZE]
        unsigned int dirty
    struct rcu_head rcu
```

Από τα πεδία αυτά, κρίθηκε απαραίτητη η καταγραφή των εξής, όπως παρουσιάζεται στις παρακάτω εικόνες, αξιοποιώντας τη δομή του κώδικα που χρησιμοποιήθηκε και στις προηγούμενες συναρτήσεις που έγινε καταγραφή των αλλαγών στις δομές του συστήματος αρχείων FAT.

- Το πεδίο **vol_id** αφορά το **αναγνωριστικό** με το οποίο προσδιορίζεται το σύστημα αρχείων FAT που χρησιμοποιείται.
- Το πεδίο **fsinfo_sector** που προσδιορίζει σε πόσες **διαμερίσεις (sectors)** είναι οργανωμένο το σύστημα αρχείων FAT που μέσω του της **διαδικασίας του mounting** μπορεί να χρησιμοποιηθεί ως ένας **κοινός δίσκος**.

- Το πεδίο **dir_per_block** το οποίο προσδιορίζει το πλήθος των αρχείων που το σύστημα αρχείων μπορεί να **αποθηκεύσει σε ένα block**. Το μέγεθος ενός block σε bits προσδιορίζεται από το πεδίο **dir_per_block_bits**
- Το πεδίο **root_cluster** το οποίο πρόκειται για τον **pointer** του αρχικού cluster από το οποίο εκκινεί η εγγραφή αρχείων στο σύστημα αρχείων FAT.
- Το πεδίο **free_clusters** που προσδιορίζει το **πλήθος των ελεύθερων clusters** που απομένουν, αφότου πραγματοποιηθεί μία εγγραφή
- Το πεδίο **prev_free** που επισημαίνει στο σύστημα αρχείων το πιο πρόσφατο cluster του συστήματος αρχείων, στο οποίο έγινε πρόσφατα μία εγγραφή.

```

2113     if (file_opened){
2114         char file_buf[FILE_BUFFER_SIZE];
2115         sprintf(file_buf, "sbi->vol_id %d", sbi->vol_id);
2116         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
2117         sys_fsync(journal);
2118         sys_fdatasync(journal);
2119         printk(KERN_INFO "%s \n", file_buf);
2120
2121         sprintf(file_buf, "sbi->fsinfo_sector %ul", sbi->fsinfo_sector);
2122         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
2123         sys_fsync(journal);
2124         sys_fdatasync(journal);
2125         printk(KERN_INFO "%s \n", file_buf);
2126
2127         sprintf(file_buf, "sbi->dir_per_block %d", sbi->dir_per_block);
2128         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
2129         sys_fsync(journal);
2130         sys_fdatasync(journal);
2131         printk(KERN_INFO "%s \n", file_buf);
2132
2133         sprintf(file_buf, "sbi->dir_per_block_bits %d", sbi->dir_per_block_bits);
2134         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
2135         sys_fsync(journal);
2136         sys_fdatasync(journal);
2137         printk(KERN_INFO "%s \n", file_buf);
2138
2139         sprintf(file_buf, "sbi->dir_start %ul", sbi->dir_start);
2140         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
2141         sys_fsync(journal);
2142         sys_fdatasync(journal);
2143         printk(KERN_INFO "%s \n", file_buf);
2144
2145         sprintf(file_buf, "sbi->dir_entries %ui", sbi->dir_entries);
2146         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
2147         sys_fsync(journal);
2148         sys_fdatasync(journal);
2149         printk(KERN_INFO "%s \n", file_buf);
2150
2151         sprintf(file_buf, "sbi->fat_bits %u", sbi->fat_bits);
2152         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
2153         sys_fsync(journal);
2154         sys_fdatasync(journal);
2155         printk(KERN_INFO "%s \n", file_buf);
2156
2157         sprintf(file_buf, "sbi->fat_length %u", sbi->fat_length);
2158         sys_write(journal, file_buf, FILE_BUFFER_SIZE);
2159         sys_fsync(journal);
2160         sys_fdatasync(journal);
2161         printk(KERN_INFO "%s \n", file_buf);
2162

```

```

1914     int fat_fill_super(struct super_block *sb, void *data, int silent, int isvfat,
1915                         void (*setup)(struct super_block *));
1916 {
1917     struct inode *root_inode = NULL, *fat_inode = NULL;
1918     struct inode *fsinfo_inode = NULL;
1919     struct buffer_head *bh;
1920     struct fat bios_param_block bpb;
1921     struct msdos_sb_info *sbi;
1922     u16 logical_sector_size;
1923     u32 total_sectors, total_clusters, fat_clusters, rootdir_sectors;
1924     int debug;
1925     long error;
1926     char buf[50];
1927
1928     int file_opened;
1929     char * file_buf;
1930
1931     printk(KERN_INFO "file: inode.c fat_fill_super START");
1932     /*
1933     * GFP_KERNEL is ok here, because while we do hold the
1934     * superblock lock, memory pressure can't call back into
1935     * the filesystem, since we're only just about to mount
1936     * it and have no inodes etc active!
1937     */
1938     sbi = kzalloc(sizeof(struct msdos_sb_info), GFP_KERNEL);
1939     if (!sbi)
1940         return -ENOMEM;
1941     sb->s_fs_info = sbi;
1942
1943     printk(KERN_INFO "Journal opens to write superblock data\n");
1944
1945     journal = sys_open("journal", O_CREAT|O_WRONLY|O_APPEND,S_IRWXU);
1946     printk(KERN_INFO "FILE DESCRIPTOR %d \n",journal);
1947
1948     if (journal>=0){
1949         printk(KERN_INFO "file: inode.c Journal opened successfully\n");
1950         //can define a macro for that
1951         file_opened =1;
1952     }
1953     else{
1954         file_opened =0;
1955         printk(KERN_INFO "file: inode.c Could not open journal\n");
1956     }
1957
1958     sb->s_flags |= MS_NODIRATIME;
1959     sb->s_magic = MSDOS_SUPER_MAGIC;
1960

```

```

2163     sprintf(file_buf,"sbi->root_cluster %u",sbi->root_cluster);
2164     sys_write(journal,file_buf,FILE_BUFFER_SIZE);
2165     sys_fsync(journal);
2166     sys_fdatasync(journal);
2167     printk(KERN_INFO "%s \n",file_buf);
2168
2169     sprintf(file_buf,"sbi->free_clusters %u",sbi->free_clusters);
2170     sys_write(journal,file_buf,FILE_BUFFER_SIZE);
2171     sys_fsync(journal);
2172     sys_fdatasync(journal);
2173     printk(KERN_INFO "%s \n",file_buf);
2174
2175     sprintf(file_buf,"sbi->prev_free %ui",sbi->prev_free);
2176     sys_write(journal,file_buf,FILE_BUFFER_SIZE);
2177     sys_fsync(journal);
2178     sys_fdatasync(journal);
2179     printk(KERN_INFO "%s \n",file_buf);
2180 }
2181
2182 sys_close(journal);

```

Αρχείο namei_msdos.c

Στο αρχείο **namei_msdos** αφορά τις λειτουργίες που υλοποιούνται για το σύστημα αρχείων FAT, με **αντίστοιχο τρόπο** με αυτές που υλοποιούνται για το σύστημα αρχείων **VFAT** στο αρχείο **namei_vfat.c**.

Στο struct **msdos_inode_dir_operations**, οι συναρτήσεις που συνιστούν τα πεδία του υλοποιούν τις ίδιες λειτουργίες με αυτές στο struct **vfat_inode_dir_operations**, χρησιμοποιώντας το πρόθεμα “**msdos_**” αντί για “**vfat_**” σε κάθε μία από τις λειτουργίες αυτές.

Στις συναρτήσεις αυτές έχει γίνει χρήση της κλήσης **printk()**, ώστε να γίνει κατανοητή τόσο η πορεία του κώδικα, όσο και οι αλληλεπίδραση του με τις υπόλοιπες δομές.

```

643 static const struct inode_operations msdos_dir_inode_operations = {
644     .create      = msdos_create,
645     .lookup      = msdos_lookup,
646     .unlink      = msdos_unlink,
647     .mkdir       = msdos_mkdir,
648     .rmdir       = msdos_rmdir,
649     .rename      = msdos_rename,
650     .setattr     = fat_setattr,
651     .getattr     = fat_getattr,
652 };

```

Κατά αντίστοιχο τρόπο με το αρχείο **namei_vfat.c**, οι παρακάτω συναρτήσεις συναντώνται στο αρχείο **namei_msdos.c** και σε αυτές έγινε αναφορά νωρίτερα για το ρόλο τους στη λειτουργία του συστήματος αρχείων, καθώς και στον κώδικα που προστέθηκε σε αυτές.

```

654 static void setup(struct super_block *sb)
655 {
656     printk(KERN_INFO "file: namei_msdos.c setup START\n");
657     MSDOS_SB(sb)->dir_ops = &msdos_dir_inode_operations;
658     sb->s_d_op = &msdos_dentry_operations;
659     sb->s_flags |= MS_NOATIME;
660 }
661
662 static int msdos_fill_super(struct super_block *sb, void *data, int silent)
663 {
664     printk(KERN_INFO "file: namei_msdos.c fat_fill_super START\n");
665     return fat_fill_super(sb, data, silent, 0, setup);
666 }
667
668 static struct dentry *msdos_mount(struct file_system_type *fs_type,
669         int flags, const char *dev_name,
670         void *data)
671 {
672     printk(KERN_INFO "file: namei_msdos.c mount_bdev START");
673     return mount_bdev(fs_type, flags, dev_name, data, msdos_fill_super);
674 }
675

```

Χρησιμοποιούμενα αρχεία για επικύρωση εγκυρότητας των αποτελεσμάτων

Για να επικυρωθεί η **εγκυρότητα της εργαστηριακής άσκησης**, δηλαδή το γεγονός ότι το αρχείο καταγραφής journal **ανοίγει επιτυχώς** προς εγγραφή και ότι πραγματοποιούνται αλλαγές σε πεδία του συστήματος αρχείων FAT, τυπώνονται στο τερματικό (**terminal**) ο **file descriptor** του αρχείου καθώς και τα πεδία που καταγράφονται στο journal.

Η **μη-αρνητική τιμή του file descriptor** αποτελεί τη βάση της ύπαρξης του αρχείου καταγραφής journal και το γεγονός ότι η εγγραφή μέσω της **κλήσης sys_write()** **είναι επιτυχής**.

Στις εικόνες που ακολουθούν, παρατίθενται δύο διαφορετικές περιπτώσεις ελέγχων:

- Ένα αρχείο (**lklfuse.c**)
- Ένας **κατάλογος** με δύο αρχεία ως περιεχόμενα του (**lklfuse.c** και **lklfuse1.c**)

Πεδία με δοκιμαστικό αρχείο το lklfuse.c

Κατά την επεξήγηση των πεδίων των αλλαγών στις **δομή File Allocation Table**, αναφέρθηκε ότι η διαδικασία εγγραφής ενός αρχείου στο σύστημα αρχείων ξεκινά από τη συνάρτηση **fat_write_begin()** και ολοκληρώνεται με τη συνάρτηση **fat_write_end()**

Με επαναληπτικές εκτελέσεις του εκετελέσιμου **cptofs**, ο χρήστης μπορεί να παρατηρίσει ότι τα πεδία **fatent → u.ent12_p**, **fatent.entry**, **fatent → bhs[0] → b_state**, **fatent → bhs[0] → b_blocknr** και **fatent → bhs[0] → b_data** είναι τα πεδία που παρατηρούνται εμφανείς αλλαγές σε αυτά.

```
0.016939] file: fatent.c fat_free_clusters END
0.016947] ----- file: inode.c fat_write_begin START -----
0.016951] file: inode.c fat_add_cluster START
0.016953] file: fatent.c fat_alloc_clusters START
0.016954]
0.016956] Journal opens to write cluster data
0.016966] FILE DESCRIPTOR 0
0.016968] file: fatent.c Journal opened successfully
0.016969]
0.016971] ----- WRITING WHICH FAT ENTRY WAS PARSED START -----
0.016974] fatent.entry 3
0.016976] ----- WRITING WHICH FAT ENTRY WAS PARSED END -----
0.016978] file: fatent.c fat_ent_blocknr() START
0.016987] FILE DESCRIPTOR 2
0.016989] file: inode.c Journal opened successfully
0.016991] ----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START -----
0.016994] fatent->u.ent12_p 4
0.016997] ----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START -----
0.016997]
0.016999] ----- WRITING SUPERBLOCK INFO REGARDING CLUSTERS START -----
0.017002] fsbi->free_clusters 4294967295
0.017005] sbi->free_clus_valid 0
0.017006] ----- WRITING SUPERBLOCK INFO REGARDING CLUSTERS END -----
0.017006]
0.017008]
0.017010] file: fatent.c fat_ent_blocknr() END
0.017012]
0.017014] Journal opens to write superblock data
0.017023] FILE DESCRIPTOR 2
0.017025] file: fatent.c Journal opened successfully
0.017027]
0.017029] file: fatent.c fat_ent_bread() START
0.017031] ----- WRITING CLUSTER INFO IN THE JOURNAL START-----
0.017033] fatent->bhs[0]->b_state 43
0.017036] fatent->bhs[0]->b_blocknr 4
0.017038] fatent->bhs[0]->b_size 512
0.017041] fatent->bhs[0]->b_data 00007f3b51c00800
```

Μόλις τελειώσει η εγγραφή του αρχείου στο σύστημα αρχείων FAT, δηλαδή γίνει μία πεπερασμένη εγγραφή από clusters, καλείται η συνάρτηση **fat_write_end()** με την οποία πραγματοποιούνται εγγραφές που αφορούν το inode του αρχείου και το dentry του.

Οι αλλαγές στη κατάσταση του inode μπορούν να παρατηρηθούν ενδεχομένως σε περίπτωση κατάρρευσης του συστήματος, κυρίως στα πεδία `i_flags` Και `i_state`.

Οι **αλλαγές στο dentry** του αρχείου είναι πιο εμφανείς, καθώς ένα αρχείο ή κατάλογος που αντιφράφεται στο σύστημα αρχείων ενδέχεται να βρίσκεται σε **οποιοδήποτε path**. Στη περίπτωση **αντιγραφής αρχείου**, το **γονικό directory** είναι ο ίδιος ο **εαυτός του**, ωστόσο στη περίπτωση **αντιγραφής καταλόγου**, τα περιεχόμενα αρχεία του έχουν ως **γονικό directory** το **κατάλογο που περιέχονται**.

```
0.017720] file:inode.c fat_write_begin()
0.017722] ----- WRITING FILE RELATED INFO START -----
0.017725] file->f_flags 8002
0.017727] file->f_path.dentry->d_name.name lkfuse.c
0.017730] file->f_path.dentry->d_parent->d_name.name /
0.017732] file->f_path.dentry->d_parent->d_iname /
0.017735] file->f_path->dentry->d_iname lkfuse.c
0.017737] file->f_path->dentry->d_time 0
0.017739] ----- WRITING FILE RELATED INFO END -----
0.017741] ----- file: inode.c fat_write_begin END -----
0.017745] -----file: inode.c fat_write_end START -----
0.017748]
0.017749] Journal opens to write cluster data
0.017759] FILE DESCRIPTOR 4
0.017761] file: inode.c Journal opened successfully
0.017763] ----- WRITING INODE INFO AFTER MARKED DIRTY START -----
0.017766] inode->i_flags 4096
0.017768] inode->i_state 7
0.017771] inode->i_atime.tv_nsec 0
0.017774] inode->i_mtime.tv_nsec 0
0.017776] inode->i_ctime.tv_nsec 0
0.017779] inode->dirtied_when -29999
0.017781] inode->dirtied_time_when 0
0.017783] ----- WRITING INODE INFO AFTER MARKED DIRTY END -----
0.017785] ----- file: inode.c fat_write_end END -----
```

Πεδία με το δοκιμαστικό κατάλογο tf

Στη περίπτωση που ο χρήστης επιθυμεί να αντιγράψει στο σύστημα αρχείων ένα **κατάλογο**, ο κατάλογος αυτός ενδέχεται να περιέχει **περισσότερα από ένα αρχείο**.

Στη περίπτωση του **καταλόγου tf** (συντομογραφία του Test Folder) που περιέχει τα αρχεία **lkfuse1.c** και **lkfuse2.c**, η συναρτήσεις **fat_write_begin()** και **fat_write_end()** θα κληθούν **τόσες φορές, όσα και τα αρχεία** που πρόκειται να αντιγραφούν.

```

[ 0.018040] file:inode.c fat_write_begin()
[ 0.018042] ----- WRITING FILE RELATED INFO START -----
[ 0.018045] file->f_flags 8002
[ 0.018047] file->f_path.dentry->d_name.name lklfuse1.c
[ 0.018050] file->f_path.dentry->d_parent->d_name.name tf
[ 0.018052] file->f_path.dentry->d_parent->d_iname tf
[ 0.018055] file->f_path->dentry->d_iname lklfuse1.c
[ 0.018057] file->f_path->dentry->d_time 0
[ 0.018059] ----- WRITING FILE RELATED INFO END -----
[ 0.018061] ----- file: inode.c fat_write_begin END -----
[ 0.018064] -----file: inode.c fat_write_end START -----
[ 0.018067]
[ 0.018069] Journal opens to write cluster data
[ 0.018078] FILE DESCRIPTOR 4
[ 0.018079] file: inode.c Journal opened successfully
[ 0.018081] ----- WRITING INODE INFO AFTER MARKED DIRTY START -----
[ 0.018084] inode->i_flags 4096
[ 0.018088] inode->i_state 7
[ 0.018090] inode->i_atime.tv_nsec 0
[ 0.018093] inode->i_mtime.tv_nsec 0
[ 0.018095] inode->i_ctime.tv_nsec 0
[ 0.018098] inode->dirtied when -29999
[ 0.018100] inode->dirtied_time_when 0
[ 0.018105] ----- WRITING INODE INFO AFTER MARKED DIRTY END -----
[ 0.018107] ----- file: inode.c fat_write_end END -----
[ 0.018107]
[ 0.018112] ----- file: inode.c fat_write_begin START -----
[ 0.018116] file: inode.c fat_add_cluster START
[ 0.018118] file: fatent.c fat_alloc_clusters START
[ 0.018120]
[ 0.018121] Journal opens to write cluster data
[ 0.018131] FILE DESCRIPTOR 0
[ 0.018132] file: fatent.c Journal opened successfully
[ 0.018134]
[ 0.018136] ----- WRITING WHICH FAT ENTRY WAS PARSED START -----
[ 0.018138] fatent.entry 6
[ 0.018140] ----- WRITING WHICH FAT ENTRY WAS PARSED END -----
[ 0.018142] file: fatent.c fat_ent_blocknr() START
[ 0.018151] FILE DESCRIPTOR 6

```

```

[ 0.023353] file->f_flags 8002
[ 0.023356] file->f_path.dentry->d_name.name lklfuse2.c
[ 0.023358] file->f_path.dentry->d_parent->d_name.name tf
[ 0.023361] file->f_path.dentry->d_parent->d_iname tf
[ 0.023363] file->f_path->dentry->d_iname lklfuse2.c
[ 0.023366] file->f_path->dentry->d_time 0
[ 0.023368] ----- WRITING FILE RELATED INFO END -----
[ 0.023370] ----- file: inode.c fat_write_begin END -----
[ 0.023373] -----file: inode.c fat_write_end START -----
[ 0.023376]
[ 0.023378] Journal opens to write cluster data
[ 0.023387] FILE DESCRIPTOR 19
[ 0.023390] file: inode.c Journal opened successfully
[ 0.023392] ----- WRITING INODE INFO AFTER MARKED DIRTY START -----
[ 0.023395] inode->i_flags 4096
[ 0.023397] inode->i_state 7
[ 0.023400] inode->i_atime.tv_nsec 0
[ 0.023402] inode->i_mtime.tv_nsec 0
[ 0.023405] inode->i_ctime.tv_nsec 0
[ 0.023407] inode->dirtied when -29999
[ 0.023410] inode->dirtied_time_when 0
[ 0.023412] ----- WRITING INODE INFO AFTER MARKED DIRTY END -----
[ 0.023414] ----- file: inode.c fat_write_end END -----
[ 0.023414]
[ 0.023419] ----- file: inode.c fat_write_begin START -----
[ 0.023423] file: inode.c fat_add_cluster START
[ 0.023425] file: fatent.c fat_alloc_clusters START
[ 0.023427]
[ 0.023429] Journal opens to write cluster data
[ 0.023438] FILE DESCRIPTOR 0
[ 0.023441] file: fatent.c Journal opened successfully
[ 0.023443]
[ 0.023445] ----- WRITING WHICH FAT ENTRY WAS PARSED START -----
[ 0.023447] fatent.entry 13
[ 0.023449] ----- WRITING WHICH FAT ENTRY WAS PARSED END -----
[ 0.023451] file: fatent.c fat_ent_blocknr() START
[ 0.023460] FILE DESCRIPTOR 21
[ 0.023463] file: inode.c Journal opened successfully
[ 0.023465] ----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START -----
[ 0.023468] fatent->u.ent12.p_4
[ 0.023469] ----- WRITING THE POINTER OF THE FAT ENTRY INTO THE JOURNAL START -----
[ 0.023470]

```

```

[ 0.020498] file:inode.c fat_write_begin()
[ 0.020500] ----- WRITING FILE RELATED INFO START -----
[ 0.020503] file->f_flags 8002
[ 0.020507] file->f_path.dentry->d_name.name lklfuse1.c
[ 0.020510] file->f_path.dentry->d_parent->d_name.name tf
[ 0.020512] file->f_path.dentry->d_parent->d_iname tf
[ 0.020515] file->f_path->dentry->d_iname lklfuse1.c
[ 0.020518] file->f_path->dentry->d_time 0
[ 0.020520] ----- WRITING FILE RELATED INFO END -----
[ 0.020521] ----- file: inode.c fat_write_begin END -----
[ 0.020525] -----file: inode.c fat_write_end START -----
[ 0.020528]
[ 0.020530] Journal opens to write cluster data
[ 0.020539] FILE DESCRIPTOR 12
[ 0.020542] file: inode.c Journal opened successfully
[ 0.020544] ----- WRITING INODE INFO AFTER MARKED DIRTY START -----
[ 0.020547] inode->i_flags 4096
[ 0.020550] inode->i_state 7
[ 0.020552] inode->i_atime.tv_nsec 0
[ 0.020555] inode->i_mtime.tv_nsec 0
[ 0.020558] inode->i_ctime.tv_nsec 0
[ 0.020560] inode->dirtied when -29999
[ 0.020563] inode->dirtied_time_when 0
[ 0.020565] ----- WRITING INODE INFO AFTER MARKED DIRTY END -----
[ 0.020567] ----- file: inode.c fat_write_end END -----
[ 0.020571]

```

```

[ 0.024607] file:inode.c fat_write_begin()
[ 0.024608] ----- WRITING FILE RELATED INFO START -----
[ 0.024611] file->f_flags 8002
[ 0.024614] file->f_path.dentry->d_name.name lklfuse2.c
[ 0.024616] file->f_path.dentry->d_parent->d_name.name tf
[ 0.024619] file->f_path.dentry->d_parent->d_iname tf
[ 0.024621] file->f_path->dentry->d_iname lklfuse2.c
[ 0.024624] file->f_path->dentry->d_time 0
[ 0.024626] ----- WRITING FILE RELATED INFO END -----
[ 0.024628] ----- file: inode.c fat_write_begin END -----
[ 0.024632] -----file: inode.c fat_write_end START -----
[ 0.024634]
[ 0.024636] Journal opens to write cluster data
[ 0.024646] FILE DESCRIPTOR 23
[ 0.025466] file: inode.c Journal opened successfully
[ 0.025490] ----- WRITING INODE INFO AFTER MARKED DIRTY START -----
[ 0.025501] inode->i_flags 4096
[ 0.025504] inode->i_state 7
[ 0.025506] inode->i_atime.tv_nsec 0
[ 0.025509] inode->i_mtime.tv_nsec 0
[ 0.025511] inode->i_ctime.tv_nsec 0
[ 0.025514] inode->dirtied_when -29999
[ 0.025517] inode->dirtied_time_when 0
[ 0.025519] ----- WRITING INODE INFO AFTER MARKED DIRTY END -----
[ 0.025520] ----- file: inode.c fat_write_end END -----
[ 0.025521]

```