**Εργαστήριο 2:** **Υλοποίηση αρχείου καταγραφής στο σύστημα αρχείων FAT του Linux**

**Ευστάθιος Ανδρεόπουλος AM: 4630**

**Γιώργος Χατζηλίγος AM: 4835**

**Contents**

[**ΛΕΙΤΟΥΡΓΙΑ ΣΥΣΤΗΜΑΤΟΣ FAT** 2](#_Toc135765556)

[**ΕΓΓΡΑΦΕΣ ΤΩΝ ΤΡΟΠΟΠΟΙΗΣΕΩΝ ΤΟΥ FAT** 53](#_Toc135765557)

## **ΛΕΙΤΟΥΡΓΙΑ ΣΥΣΤΗΜΑΤΟΣ FAT**

**Ερώτημα**

Για να κατανοήσετε την λειτουργία του συστήματος αρχείων FAT μπορείτε να χρησιμοποιήσετε τις εφαρμογές boot και cptofs ώστε να δείτε τις λειτουργίες που εμπλέκονται κατά την προσπέλαση του αρχείου /tmp/vfatfile. Μπορείτε να εκτελέσετε τις εφαρμογές με τον αποσφαλματωτή gdb προκειμένου να προχωρήσετε βηματικά στις συναρτήσεις που καλούνται κατά την εκτέλεση. Επιπλέον, μπορείτε να τυπώσετε μηνύματα στην κονσόλα προσθέτοντας την συνάρτηση printk(KERN\_INFO "...", ...) στον πηγαίο κώδικα της LKL (π.χ., στον κατάλογο fat/ ή mm/) και μεταγλωττίζοντας ξανά την βιβλιοθήκη και την εφαρμογή (μπορεί να χρειαστεί clean για να γίνουν ορατές οι αλλαγές στο εκτελέσιμο).

**Απάντηση**

Αρχικα επισημενουμε τις εντολες που ακολουθησαμε με επεξηγηση της κάθε εντολης

Σειρά Εντολών :

1. Μέσα στο lkl/lkl source εκτελούμε make -j8 clean ; make –j8 – C tools/lkl clean; make -j8 -C tools/lkl ;
   1. Εκτελεί τις εντολές την μια μετά την άλλη φτιάχνει αρχικά 8 threads στο lkl/ lkl-source και στο tools/lkl
2. cd tools/lkl
   1. πηγαίνει στο φάκελο
3. make test
   1. κάνει μεταγλώττιση το πρόγραμμα
4. ./cptofos -i /tmp/vfatfile -p -t vfat file /
   1. Αντιγράφει το αρχείο file τοπικά στην εικονική μηχανή δηλαδή στο vm στο /to/tmp/vfatfile
5. su root
   1. γινόμαστε διαχειριστής
6. mount -t vfat -o loop /tmp/vfatfile / vfat
   1. Προσπελαση αρχειο απευθειας από το λειτουργικο συστημα του vm
7. Παμε στον φακελο /vfat
8. umount /vfat ώστε να μπορουμε να ξανακανουμε μελλοντικα mount
9. exit για να βγουμε από το root

|  |  |
| --- | --- |
| **Εντολες** | |
| 1. | make -j8 clean ; make –j8 – C tools/lkl clean; make -j8 -C tools/lkl ; |
| 2. | make test |
| 3. | ./cptofos -i /tmp/vfatfile -p -t vfat file / |
| 4. | su root |
| 5. | mount -t vfat -o loop /tmp/vfatfile / vfat |
| 6. | umount |
| 7. |  |

Με αυτές τις εντολές πέτυχαμε

1. την εκτέλεση του fat και μπορούμε να δούμε στο terminal .
2. ποιες λειτουργίες εκτελέστηκαν .
3. να δούμε εάν αντιγράφεται το file στην εικονική μηχανή οπού μετρά θα μπορούμε να προσπελασουμε το συστημα .

**Απασφαλματωση**

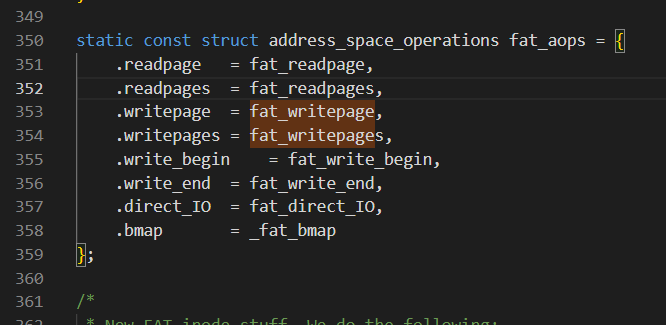
Η απασφαλματωση θα γίνει με την εντολή printk σε οποιαδήποτε σημεία του κώδικα όπως επισημάνθηκε από τις οδηγίες που μας δοθήκαν .Επίσης ,θα υπάρχουν screenshot με σημαντικες αλλαγες οι οποιες εγιναν σε διαφορα αρχεια μεσα στο lkl-source .

Η λογικη που θα χρησιμοποιησουμε για την αποσφαλματωση του κωδικα είναι οι εξης να παμε σε ολες τις συναρτησεις που ελεγχουν το fat και να βαλουμε την την εντολη: printk(KERN\_INFO “Access >> %s: %s ”, \_\_FILE\_\_,\_\_func\_\_);

Επιτυγχανουμε με \_\_FILE\_\_να παιρνουμε το path του φακελου και με \_\_func\_\_ το ονομα της συναρτησης που θα εμφανιστουν όταν εκτελεσουμε την εντολη 3.

Προοθουμε το το αποτλεσμα της εντολης 3 κάθε φορα σε ένα αρχεια txt με την ανακατευθυνση “>>” στο terminal.

struct address\_space\_operations fat\_aops



Generic function implemented

static int fat\_writepage(struct page \*page, struct writeback\_control \*wbc)

{

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    return block\_write\_full\_page(page, fat\_get\_block, wbc);

}

static int fat\_writepages(struct address\_space \*mapping,

              struct writeback\_control \*wbc)

{

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    return mpage\_writepages(mapping, wbc, fat\_get\_block);

}

static int fat\_readpage(struct file \*file, struct page \*page)

{

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    return mpage\_readpage(page, fat\_get\_block);

}

static int fat\_readpages(struct file \*file, struct address\_space \*mapping,

             struct list\_head \*pages, unsigned nr\_pages)

{

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    return mpage\_readpages(mapping, pages, nr\_pages, fat\_get\_block);

}

static void fat\_write\_failed(struct address\_space \*mapping, loff\_t to)

{

    struct inode \*inode = mapping->host;

    if (to > inode->i\_size) {

        truncate\_pagecache(inode, inode->i\_size);

        fat\_truncate\_blocks(inode, inode->i\_size);

    }

}

static int fat\_write\_begin(struct file \*file, struct address\_space \*mapping,

            loff\_t pos, unsigned len, unsigned flags,

            struct page \*\*pagep, void \*\*fsdata)

{

    int err;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    \*pagep = NULL;

    err = cont\_write\_begin(file, mapping, pos, len, flags,

                pagep, fsdata, fat\_get\_block,

                &MSDOS\_I(mapping->host)->mmu\_private);

    if (err < 0)

        fat\_write\_failed(mapping, pos + len);

    return err;

}

static int fat\_write\_end(struct file \*file, struct address\_space \*mapping,

            loff\_t pos, unsigned len, unsigned copied,

            struct page \*pagep, void \*fsdata)

{

    struct inode \*inode = mapping->host;

    int err;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    err = generic\_write\_end(file, mapping, pos, len, copied, pagep, fsdata);

    if (err < len)

        fat\_write\_failed(mapping, pos + len);

    if (!(err < 0) && !(MSDOS\_I(inode)->i\_attrs & ATTR\_ARCH)) {

        inode->i\_mtime = inode->i\_ctime = current\_time(inode);

        MSDOS\_I(inode)->i\_attrs |= ATTR\_ARCH;

        mark\_inode\_dirty(inode);

    }

    return err;

}

static ssize\_t fat\_direct\_IO(struct kiocb \*iocb, struct iov\_iter \*iter)

{

    struct file \*file = iocb->ki\_filp;

    struct address\_space \*mapping = file->f\_mapping;

    struct inode \*inode = mapping->host;

    size\_t count = iov\_iter\_count(iter);

    loff\_t offset = iocb->ki\_pos;

    ssize\_t ret;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    if (iov\_iter\_rw(iter) == WRITE) {

        /\*

         \* FIXME: blockdev\_direct\_IO() doesn't use ->write\_begin(),

         \* so we need to update the ->mmu\_private to block boundary.

         \*

         \* But we must fill the remaining area or hole by nul for

         \* updating ->mmu\_private.

         \*

         \* Return 0, and fallback to normal buffered write.

         \*/

        loff\_t size = offset + count;

        if (MSDOS\_I(inode)->mmu\_private < size)

            return 0;

    }

    /\*

     \* FAT need to use the DIO\_LOCKING for avoiding the race

     \* condition of fat\_get\_block() and ->truncate().

     \*/

    ret = blockdev\_direct\_IO(iocb, inode, iter, fat\_get\_block);

    if (ret < 0 && iov\_iter\_rw(iter) == WRITE)

        fat\_write\_failed(mapping, offset + count);

    return ret;

}

static sector\_t \_fat\_bmap(struct address\_space \*mapping, sector\_t block)

{

    sector\_t blocknr;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    /\* fat\_get\_cluster() assumes the requested blocknr isn't truncated. \*/

    down\_read(&MSDOS\_I(mapping->host)->truncate\_lock);

    blocknr = generic\_block\_bmap(mapping, block, fat\_get\_block\_bmap);

    up\_read(&MSDOS\_I(mapping->host)->truncate\_lock);

    return blocknr;

}

struct super\_operations fat\_sops

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,

};

Generic functions implemented

static void fat\_put\_super(struct super\_block \*sb)

{

    struct msdos\_sb\_info \*sbi = MSDOS\_SB(sb);

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    fat\_set\_state(sb, 0, 0);

    iput(sbi->fsinfo\_inode);

    iput(sbi->fat\_inode);

    call\_rcu(&sbi->rcu, delayed\_free);

}

static struct kmem\_cache \*fat\_inode\_cachep;

static struct inode \*fat\_alloc\_inode(struct super\_block \*sb)

{

    struct msdos\_inode\_info \*ei;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    ei = kmem\_cache\_alloc(fat\_inode\_cachep, GFP\_NOFS);

    if (!ei)

        return NULL;

    init\_rwsem(&ei->truncate\_lock);

    return &ei->vfs\_inode;

}

static void fat\_i\_callback(struct rcu\_head \*head)

{

    struct inode \*inode = container\_of(head, struct inode, i\_rcu);

    kmem\_cache\_free(fat\_inode\_cachep, MSDOS\_I(inode));

}

static void fat\_destroy\_inode(struct inode \*inode)

{

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    call\_rcu(&inode->i\_rcu, fat\_i\_callback);

}

static void \_\_exit fat\_destroy\_inodecache(void)

{

    /\*

     \* Make sure all delayed rcu free inodes are flushed before we

     \* destroy cache.

     \*/

    rcu\_barrier();

    kmem\_cache\_destroy(fat\_inode\_cachep);

}

static int fat\_remount(struct super\_block \*sb, int \*flags, char \*data)

{

    int new\_rdonly;

    struct msdos\_sb\_info \*sbi = MSDOS\_SB(sb);

    \*flags |= MS\_NODIRATIME | (sbi->options.isvfat ? 0 : MS\_NOATIME);

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    sync\_filesystem(sb);

    /\* make sure we update state on remount. \*/

    new\_rdonly = \*flags & MS\_RDONLY;

    if (new\_rdonly != (sb->s\_flags & MS\_RDONLY)) {

        if (new\_rdonly)

            fat\_set\_state(sb, 0, 0);

        else

            fat\_set\_state(sb, 1, 1);

    }

    return 0;

}

static int fat\_statfs(struct dentry \*dentry, struct kstatfs \*buf)

{

    struct super\_block \*sb = dentry->d\_sb;

    struct msdos\_sb\_info \*sbi = MSDOS\_SB(sb);

    u64 id = huge\_encode\_dev(sb->s\_bdev->bd\_dev);

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    /\* If the count of free cluster is still unknown, counts it here. \*/

    if (sbi->free\_clusters == -1 || !sbi->free\_clus\_valid) {

        int err = fat\_count\_free\_clusters(dentry->d\_sb);

        if (err)

            return err;

    }

    buf->f\_type = dentry->d\_sb->s\_magic;

    buf->f\_bsize = sbi->cluster\_size;

    buf->f\_blocks = sbi->max\_cluster - FAT\_START\_ENT;

    buf->f\_bfree = sbi->free\_clusters;

    buf->f\_bavail = sbi->free\_clusters;

    buf->f\_fsid.val[0] = (u32)id;

    buf->f\_fsid.val[1] = (u32)(id >> 32);

    buf->f\_namelen =

        (sbi->options.isvfat ? FAT\_LFN\_LEN : 12) \* NLS\_MAX\_CHARSET\_SIZE;

    return 0;

}

static int \_\_fat\_write\_inode(struct inode \*inode, int wait)

{

    struct super\_block \*sb = inode->i\_sb;

    struct msdos\_sb\_info \*sbi = MSDOS\_SB(sb);

    struct buffer\_head \*bh;

    struct msdos\_dir\_entry \*raw\_entry;

    loff\_t i\_pos;

    sector\_t blocknr;

    int err, offset;

    if (inode->i\_ino == MSDOS\_ROOT\_INO)

        return 0;

retry:

    i\_pos = fat\_i\_pos\_read(sbi, inode);

    if (!i\_pos)

        return 0;

    fat\_get\_blknr\_offset(sbi, i\_pos, &blocknr, &offset);

    bh = sb\_bread(sb, blocknr);

    if (!bh) {

        fat\_msg(sb, KERN\_ERR, "unable to read inode block "

               "for updating (i\_pos %lld)", i\_pos);

        return -EIO;

    }

    spin\_lock(&sbi->inode\_hash\_lock);

    if (i\_pos != MSDOS\_I(inode)->i\_pos) {

        spin\_unlock(&sbi->inode\_hash\_lock);

        brelse(bh);

        goto retry;

    }

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    raw\_entry = &((struct msdos\_dir\_entry \*) (bh->b\_data))[offset];

    if (S\_ISDIR(inode->i\_mode))

        raw\_entry->size = 0;

    else

        raw\_entry->size = cpu\_to\_le32(inode->i\_size);

    raw\_entry->attr = fat\_make\_attrs(inode);

    fat\_set\_start(raw\_entry, MSDOS\_I(inode)->i\_logstart);

    fat\_time\_unix2fat(sbi, &inode->i\_mtime, &raw\_entry->time,

              &raw\_entry->date, NULL);

    if (sbi->options.isvfat) {

        \_\_le16 atime;

        fat\_time\_unix2fat(sbi, &inode->i\_ctime, &raw\_entry->ctime,

                  &raw\_entry->cdate, &raw\_entry->ctime\_cs);

        fat\_time\_unix2fat(sbi, &inode->i\_atime, &atime,

                  &raw\_entry->adate, NULL);

    }

    spin\_unlock(&sbi->inode\_hash\_lock);

    mark\_buffer\_dirty(bh);

    err = 0;

    if (wait)

        err = sync\_dirty\_buffer(bh);

    brelse(bh);

    return err;

}

static int fat\_write\_inode(struct inode \*inode, struct writeback\_control \*wbc)

{

    int err;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    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);

    return err;

}

struct fatent\_operations fat12\_ops

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, αριθμός

Περιγραφή που δημιουργήθηκε αυτόματα

Generic functions implemented

static void fat12\_ent\_blocknr(struct super\_block \*sb, int entry,

                  int \*offset, sector\_t \*blocknr)

{

    struct msdos\_sb\_info \*sbi = MSDOS\_SB(sb);

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    int bytes = entry + (entry >> 1);

    WARN\_ON(entry < FAT\_START\_ENT || sbi->max\_cluster <= entry);

    \*offset = bytes & (sb->s\_blocksize - 1);

    \*blocknr = sbi->fat\_start + (bytes >> sb->s\_blocksize\_bits);

}

static void fat12\_ent\_set\_ptr(struct fat\_entry \*fatent, int offset)

{

    struct buffer\_head \*\*bhs = fatent->bhs;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    if (fatent->nr\_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;

    }

}

static int fat12\_ent\_bread(struct super\_block \*sb, struct fat\_entry \*fatent,

               int offset, sector\_t blocknr)

{

    struct buffer\_head \*\*bhs = fatent->bhs;

    WARN\_ON(blocknr < MSDOS\_SB(sb)->fat\_start);

    fatent->fat\_inode = MSDOS\_SB(sb)->fat\_inode;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    bhs[0] = sb\_bread(sb, blocknr);

    if (!bhs[0])

        goto err;

    if ((offset + 1) < sb->s\_blocksize)

        fatent->nr\_bhs = 1;

    else {

*/\* This entry is block boundary, it needs the next block \*/*

        blocknr++;

        bhs[1] = sb\_bread(sb, blocknr);

        if (!bhs[1])

            goto err\_brelse;

        fatent->nr\_bhs = 2;

    }

    fat12\_ent\_set\_ptr(fatent, offset);

    return 0;

err\_brelse:

    brelse(bhs[0]);

err:

    fat\_msg(sb, KERN\_ERR, "FAT read failed (blocknr %llu)", (llu)blocknr);

    return -EIO;

}

static int fat12\_ent\_get(struct fat\_entry \*fatent)

{

    u8 \*\*ent12\_p = fatent->u.ent12\_p;

    int next;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    spin\_lock(&fat12\_entry\_lock);

    if (fatent->entry & 1)

        next = (\*ent12\_p[0] >> 4) | (\*ent12\_p[1] << 4);

    else

        next = (\*ent12\_p[1] << 8) | \*ent12\_p[0];

    spin\_unlock(&fat12\_entry\_lock);

    next &= 0x0fff;

    if (next >= BAD\_FAT12)

        next = FAT\_ENT\_EOF;

    return next;

}

static void fat12\_ent\_put(struct fat\_entry \*fatent, int new)

{

    u8 \*\*ent12\_p = fatent->u.ent12\_p;

    if (new == FAT\_ENT\_EOF)

        new = EOF\_FAT12;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    spin\_lock(&fat12\_entry\_lock);

    if (fatent->entry & 1) {

        \*ent12\_p[0] = (new << 4) | (\*ent12\_p[0] & 0x0f);

        \*ent12\_p[1] = new >> 4;

    } else {

        \*ent12\_p[0] = new & 0xff;

        \*ent12\_p[1] = (\*ent12\_p[1] & 0xf0) | (new >> 8);

    }

    spin\_unlock(&fat12\_entry\_lock);

    mark\_buffer\_dirty\_inode(fatent->bhs[0], fatent->fat\_inode);

    if (fatent->nr\_bhs == 2)

        mark\_buffer\_dirty\_inode(fatent->bhs[1], fatent->fat\_inode);

}

static int fat12\_ent\_next(struct fat\_entry \*fatent)

{

    u8 \*\*ent12\_p = fatent->u.ent12\_p;

    struct buffer\_head \*\*bhs = fatent->bhs;

    u8 \*nextp = ent12\_p[1] + 1 + (fatent->entry & 1);

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    fatent->entry++;

    if (fatent->nr\_bhs == 1) {

        WARN\_ON(ent12\_p[0] > (u8 \*)(bhs[0]->b\_data +

                            (bhs[0]->b\_size - 2)));

        WARN\_ON(ent12\_p[1] > (u8 \*)(bhs[0]->b\_data +

                            (bhs[0]->b\_size - 1)));

        if (nextp < (u8 \*)(bhs[0]->b\_data + (bhs[0]->b\_size - 1))) {

            ent12\_p[0] = nextp - 1;

            ent12\_p[1] = nextp;

            return 1;

        }

    } else {

        WARN\_ON(ent12\_p[0] != (u8 \*)(bhs[0]->b\_data +

                            (bhs[0]->b\_size - 1)));

        WARN\_ON(ent12\_p[1] != (u8 \*)bhs[1]->b\_data);

        ent12\_p[0] = nextp - 1;

        ent12\_p[1] = nextp;

        brelse(bhs[0]);

        bhs[0] = bhs[1];

        fatent->nr\_bhs = 1;

        return 1;

    }

    ent12\_p[0] = NULL;

    ent12\_p[1] = NULL;

    return 0;

}

struct fatent\_operations fat16\_ops

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, αριθμός

Περιγραφή που δημιουργήθηκε αυτόματα

Generic functions implemented

static void fat\_ent\_blocknr(struct super\_block \*sb, int entry,

                int \*offset, sector\_t \*blocknr)

{

    struct msdos\_sb\_info \*sbi = MSDOS\_SB(sb);

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    int bytes = (entry << sbi->fatent\_shift);

    WARN\_ON(entry < FAT\_START\_ENT || sbi->max\_cluster <= entry);

    \*offset = bytes & (sb->s\_blocksize - 1);

    \*blocknr = sbi->fat\_start + (bytes >> sb->s\_blocksize\_bits);

}

static void fat16\_ent\_set\_ptr(struct fat\_entry \*fatent, int offset)

{

    WARN\_ON(offset & (2 - 1));

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    fatent->u.ent16\_p = (\_\_le16 \*)(fatent->bhs[0]->b\_data + offset);

}

static int fat\_ent\_bread(struct super\_block \*sb, struct fat\_entry \*fatent,

             int offset, sector\_t blocknr)

{

    const struct fatent\_operations \*ops = MSDOS\_SB(sb)->fatent\_ops;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    WARN\_ON(blocknr < MSDOS\_SB(sb)->fat\_start);

    fatent->fat\_inode = MSDOS\_SB(sb)->fat\_inode;

    fatent->bhs[0] = sb\_bread(sb, blocknr);

    if (!fatent->bhs[0]) {

        fat\_msg(sb, KERN\_ERR, "FAT read failed (blocknr %llu)",

               (llu)blocknr);

        return -EIO;

    }

    fatent->nr\_bhs = 1;

    ops->ent\_set\_ptr(fatent, offset);

    return 0;

}

static int fat16\_ent\_get(struct fat\_entry \*fatent)

{

    int next = le16\_to\_cpu(\*fatent->u.ent16\_p);

    WARN\_ON((unsigned long)fatent->u.ent16\_p & (2 - 1));

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    if (next >= BAD\_FAT16)

        next = FAT\_ENT\_EOF;

    return next;

}

static void fat16\_ent\_put(struct fat\_entry \*fatent, int new)

{

    if (new == FAT\_ENT\_EOF)

        new = EOF\_FAT16;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    \*fatent->u.ent16\_p = cpu\_to\_le16(new);

    mark\_buffer\_dirty\_inode(fatent->bhs[0], fatent->fat\_inode);

}

static int fat16\_ent\_next(struct fat\_entry \*fatent)

{

    const struct buffer\_head \*bh = fatent->bhs[0];

    fatent->entry++;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    if (fatent->u.ent16\_p < (\_\_le16 \*)(bh->b\_data + (bh->b\_size - 2))) {

        fatent->u.ent16\_p++;

        return 1;

    }

    fatent->u.ent16\_p = NULL;

    return 0;

}

struct fatent\_operations fat32\_ops

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,

};

Generic functions implemented

static void fat\_ent\_blocknr(struct super\_block \*sb, int entry,

                int \*offset, sector\_t \*blocknr)

{

    struct msdos\_sb\_info \*sbi = MSDOS\_SB(sb);

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    int bytes = (entry << sbi->fatent\_shift);

    WARN\_ON(entry < FAT\_START\_ENT || sbi->max\_cluster <= entry);

    \*offset = bytes & (sb->s\_blocksize - 1);

    \*blocknr = sbi->fat\_start + (bytes >> sb->s\_blocksize\_bits);

}

static int fat32\_ent\_next(struct fat\_entry \*fatent)

{

    const struct buffer\_head \*bh = fatent->bhs[0];

    fatent->entry++;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    if (fatent->u.ent32\_p < (\_\_le32 \*)(bh->b\_data + (bh->b\_size - 4))) {

        fatent->u.ent32\_p++;

        return 1;

    }

    fatent->u.ent32\_p = NULL;

    return 0;

}

static int fat\_ent\_bread(struct super\_block \*sb, struct fat\_entry \*fatent,

             int offset, sector\_t blocknr)

{

    const struct fatent\_operations \*ops = MSDOS\_SB(sb)->fatent\_ops;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    WARN\_ON(blocknr < MSDOS\_SB(sb)->fat\_start);

    fatent->fat\_inode = MSDOS\_SB(sb)->fat\_inode;

    fatent->bhs[0] = sb\_bread(sb, blocknr);

    if (!fatent->bhs[0]) {

        fat\_msg(sb, KERN\_ERR, "FAT read failed (blocknr %llu)",

               (llu)blocknr);

        return -EIO;

    }

    fatent->nr\_bhs = 1;

    ops->ent\_set\_ptr(fatent, offset);

    return 0;

}

static int fat32\_ent\_get(struct fat\_entry \*fatent)

{

    int next = le32\_to\_cpu(\*fatent->u.ent32\_p) & 0x0fffffff;

    WARN\_ON((unsigned long)fatent->u.ent32\_p & (4 - 1));

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    if (next >= BAD\_FAT32)

        next = FAT\_ENT\_EOF;

    return next;

}

static void fat32\_ent\_put(struct fat\_entry \*fatent, int new)

{

    WARN\_ON(new & 0xf0000000);

    new |= le32\_to\_cpu(\*fatent->u.ent32\_p) & ~0x0fffffff;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    \*fatent->u.ent32\_p = cpu\_to\_le32(new);

    mark\_buffer\_dirty\_inode(fatent->bhs[0], fatent->fat\_inode);

}

Struct file\_operations fat\_file\_operations

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,

#ifdef CONFIG\_COMPAT

    .compat\_ioctl   = fat\_generic\_compat\_ioctl,

#endif

    .fsync      = fat\_file\_fsync,

    .splice\_read    = generic\_file\_splice\_read,

    .fallocate  = fat\_fallocate,

};

Generic functions implemented

#ifdef CONFIG\_COMPAT

static long fat\_generic\_compat\_ioctl(struct file \*filp, unsigned int cmd,

                      unsigned long arg)

{

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    return fat\_generic\_ioctl(filp, cmd, (unsigned long)compat\_ptr(arg));

}

#endif

int fat\_file\_fsync(struct file \*filp, loff\_t start, loff\_t end, int datasync)

{

    struct inode \*inode = filp->f\_mapping->host;

    int res, err;

    res = generic\_file\_fsync(filp, start, end, datasync);

    err = sync\_mapping\_buffers(MSDOS\_SB(inode->i\_sb)->fat\_inode->i\_mapping);

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    return res ? res : err;

}

static long fat\_fallocate(struct file \*file, int mode,

              loff\_t offset, loff\_t len)

{

    int nr\_cluster; */\* Number of clusters to be allocated \*/*

    loff\_t mm\_bytes; */\* Number of bytes to be allocated for file \*/*

    loff\_t ondisksize; */\* block aligned on-disk size in bytes\*/*

    struct inode \*inode = file->f\_mapping->host;

    struct super\_block \*sb = inode->i\_sb;

    struct msdos\_sb\_info \*sbi = MSDOS\_SB(sb);

    int err = 0;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

*/\* No support for hole punch or other fallocate flags. \*/*

    if (mode & ~FALLOC\_FL\_KEEP\_SIZE)

        return -EOPNOTSUPP;

*/\* No support for dir \*/*

    if (!S\_ISREG(inode->i\_mode))

        return -EOPNOTSUPP;

    inode\_lock(inode);

    if (mode & FALLOC\_FL\_KEEP\_SIZE) {

        ondisksize = inode->i\_blocks << 9;

        if ((offset + len) <= ondisksize)

            goto error;

*/\* First compute the number of clusters to be allocated \*/*

        mm\_bytes = offset + len - ondisksize;

        nr\_cluster = (mm\_bytes + (sbi->cluster\_size - 1)) >>

            sbi->cluster\_bits;

*/\* Start the allocation.We are not zeroing out the clusters \*/*

        while (nr\_cluster-- > 0) {

            err = fat\_add\_cluster(inode);

            if (err)

                goto error;

        }

    } else {

        if ((offset + len) <= i\_size\_read(inode))

            goto error;

*/\* This is just an expanding truncate \*/*

        err = fat\_cont\_expand(inode, (offset + len));

    }

error:

    inode\_unlock(inode);

    return err;

}

struct inode\_operations fat\_file\_inode\_operations

const struct inode\_operations fat\_file\_inode\_operations = {

    .setattr    = fat\_setattr,

    .getattr    = fat\_getattr,

};

Generic functions implemented

int fat\_setattr(struct dentry \*dentry, struct iattr \*attr)

{

    struct msdos\_sb\_info \*sbi = MSDOS\_SB(dentry->d\_sb);

    struct inode \*inode = d\_inode(dentry);

    unsigned int ia\_valid;

    int error;

*/\* Check for setting the inode time. \*/*

    ia\_valid = attr->ia\_valid;

    if (ia\_valid & TIMES\_SET\_FLAGS) {

        if (fat\_allow\_set\_time(sbi, inode))

            attr->ia\_valid &= ~TIMES\_SET\_FLAGS;

    }

    error = setattr\_prepare(dentry, attr);

    attr->ia\_valid = ia\_valid;

    if (error) {

        if (sbi->options.quiet)

            error = 0;

        goto out;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    }

*/\**

*\* Expand the file. Since inode\_setattr() updates ->i\_size*

*\* before calling the ->truncate(), but FAT needs to fill the*

*\* hole before it. XXX: this is no longer true with new truncate*

*\* sequence.*

*\*/*

    if (attr->ia\_valid & ATTR\_SIZE) {

        inode\_dio\_wait(inode);

        if (attr->ia\_size > inode->i\_size) {

            error = fat\_cont\_expand(inode, attr->ia\_size);

            if (error || attr->ia\_valid == ATTR\_SIZE)

                goto out;

            attr->ia\_valid &= ~ATTR\_SIZE;

        }

    }

    if (((attr->ia\_valid & ATTR\_UID) &&

         (!uid\_eq(attr->ia\_uid, sbi->options.fs\_uid))) ||

        ((attr->ia\_valid & ATTR\_GID) &&

         (!gid\_eq(attr->ia\_gid, sbi->options.fs\_gid))) ||

        ((attr->ia\_valid & ATTR\_MODE) &&

         (attr->ia\_mode & ~FAT\_VALID\_MODE)))

        error = -EPERM;

    if (error) {

        if (sbi->options.quiet)

            error = 0;

        goto out;

    }

*/\**

*\* We don't return -EPERM here. Yes, strange, but this is too*

*\* old behavior.*

*\*/*

    if (attr->ia\_valid & ATTR\_MODE) {

        if (fat\_sanitize\_mode(sbi, inode, &attr->ia\_mode) < 0)

            attr->ia\_valid &= ~ATTR\_MODE;

    }

    if (attr->ia\_valid & ATTR\_SIZE) {

        error = fat\_block\_truncate\_page(inode, attr->ia\_size);

        if (error)

            goto out;

        down\_write(&MSDOS\_I(inode)->truncate\_lock);

        truncate\_setsize(inode, attr->ia\_size);

        fat\_truncate\_blocks(inode, attr->ia\_size);

        up\_write(&MSDOS\_I(inode)->truncate\_lock);

    }

    setattr\_copy(inode, attr);

    mark\_inode\_dirty(inode);

out:

    return error;

}

int fat\_getattr(const struct path \*path, struct kstat \*stat,

        u32 request\_mask, unsigned int flags)

{

    struct inode \*inode = d\_inode(path->dentry);

    generic\_fillattr(inode, stat);

    stat->blksize = MSDOS\_SB(inode->i\_sb)->cluster\_size;

    if (MSDOS\_SB(inode->i\_sb)->options.nfs == FAT\_NFS\_NOSTALE\_RO) {

*/\* Use i\_pos for ino. This is used as fileid of nfs. \*/*

        stat->ino = fat\_i\_pos\_read(MSDOS\_SB(inode->i\_sb), inode);

    }

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    return 0;

}

EXPORT\_SYMBOL\_GPL(fat\_getattr);

inode\_operations msdos\_dir\_inode\_operations

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,

};

Generic functions implemented

static int msdos\_create(struct inode \*dir, struct dentry \*dentry, umode\_t mode,

            bool excl)

{

    struct super\_block \*sb = dir->i\_sb;

    struct inode \*inode = NULL;

    struct fat\_slot\_info sinfo;

    struct timespec ts;

    unsigned char msdos\_name[MSDOS\_NAME];

    int err, is\_hid;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    mutex\_lock(&MSDOS\_SB(sb)->s\_lock);

    err = msdos\_format\_name(dentry->d\_name.name, dentry->d\_name.len,

                msdos\_name, &MSDOS\_SB(sb)->options);

    if (err)

        goto out;

    is\_hid = (dentry->d\_name.name[0] == '.') && (msdos\_name[0] != '.');

*/\* Have to do it due to foo vs. .foo conflicts \*/*

    if (!fat\_scan(dir, msdos\_name, &sinfo)) {

        brelse(sinfo.bh);

        err = -EINVAL;

        goto out;

    }

    ts = current\_time(dir);

    err = msdos\_add\_entry(dir, msdos\_name, 0, is\_hid, 0, &ts, &sinfo);

    if (err)

        goto out;

    inode = fat\_build\_inode(sb, sinfo.de, sinfo.i\_pos);

    brelse(sinfo.bh);

    if (IS\_ERR(inode)) {

        err = PTR\_ERR(inode);

        goto out;

    }

    inode->i\_mtime = inode->i\_atime = inode->i\_ctime = ts;

*/\* timestamp is already written, so mark\_inode\_dirty() is unneeded. \*/*

    d\_instantiate(dentry, inode);

out:

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    if (!err)

        err = fat\_flush\_inodes(sb, dir, inode);

    return err;

}

static struct dentry \*msdos\_lookup(struct inode \*dir, struct dentry \*dentry,

                   unsigned int flags)

{

    struct super\_block \*sb = dir->i\_sb;

    struct fat\_slot\_info sinfo;

    struct inode \*inode;

    int err;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    mutex\_lock(&MSDOS\_SB(sb)->s\_lock);

    err = msdos\_find(dir, dentry->d\_name.name, dentry->d\_name.len, &sinfo);

    switch (err) {

    case -ENOENT:

        inode = NULL;

        break;

    case 0:

        inode = fat\_build\_inode(sb, sinfo.de, sinfo.i\_pos);

        brelse(sinfo.bh);

        break;

    default:

        inode = ERR\_PTR(err);

    }

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    return d\_splice\_alias(inode, dentry);

}

static int msdos\_unlink(struct inode \*dir, struct dentry \*dentry)

{

    struct inode \*inode = d\_inode(dentry);

    struct super\_block \*sb = inode->i\_sb;

    struct fat\_slot\_info sinfo;

    int err;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    mutex\_lock(&MSDOS\_SB(sb)->s\_lock);

    err = msdos\_find(dir, dentry->d\_name.name, dentry->d\_name.len, &sinfo);

    if (err)

        goto out;

    err = fat\_remove\_entries(dir, &sinfo); */\* and releases bh \*/*

    if (err)

        goto out;

    clear\_nlink(inode);

    inode->i\_ctime = current\_time(inode);

    fat\_detach(inode);

out:

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    if (!err)

        err = fat\_flush\_inodes(sb, dir, inode);

    return err;

}

static int msdos\_mkdir(struct inode \*dir, struct dentry \*dentry, umode\_t mode)

{

    struct super\_block \*sb = dir->i\_sb;

    struct fat\_slot\_info sinfo;

    struct inode \*inode;

    unsigned char msdos\_name[MSDOS\_NAME];

    struct timespec ts;

    int err, is\_hid, cluster;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    mutex\_lock(&MSDOS\_SB(sb)->s\_lock);

    err = msdos\_format\_name(dentry->d\_name.name, dentry->d\_name.len,

                msdos\_name, &MSDOS\_SB(sb)->options);

    if (err)

        goto out;

    is\_hid = (dentry->d\_name.name[0] == '.') && (msdos\_name[0] != '.');

*/\* foo vs .foo situation \*/*

    if (!fat\_scan(dir, msdos\_name, &sinfo)) {

        brelse(sinfo.bh);

        err = -EINVAL;

        goto out;

    }

    ts = current\_time(dir);

    cluster = fat\_alloc\_new\_dir(dir, &ts);

    if (cluster < 0) {

        err = cluster;

        goto out;

    }

    err = msdos\_add\_entry(dir, msdos\_name, 1, is\_hid, cluster, &ts, &sinfo);

    if (err)

        goto out\_free;

    inc\_nlink(dir);

    inode = fat\_build\_inode(sb, sinfo.de, sinfo.i\_pos);

    brelse(sinfo.bh);

    if (IS\_ERR(inode)) {

        err = PTR\_ERR(inode);

*/\* the directory was completed, just return a error \*/*

        goto out;

    }

    set\_nlink(inode, 2);

    inode->i\_mtime = inode->i\_atime = inode->i\_ctime = ts;

*/\* timestamp is already written, so mark\_inode\_dirty() is unneeded. \*/*

    d\_instantiate(dentry, inode);

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    fat\_flush\_inodes(sb, dir, inode);

    return 0;

out\_free:

    fat\_free\_clusters(dir, cluster);

out:

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    return err;

}

static int msdos\_rmdir(struct inode \*dir, struct dentry \*dentry)

{

    struct super\_block \*sb = dir->i\_sb;

    struct inode \*inode = d\_inode(dentry);

    struct fat\_slot\_info sinfo;

    int err;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    mutex\_lock(&MSDOS\_SB(sb)->s\_lock);

*/\**

*\* Check whether the directory is not in use, then check*

*\* whether it is empty.*

*\*/*

    err = fat\_dir\_empty(inode);

    if (err)

        goto out;

    err = msdos\_find(dir, dentry->d\_name.name, dentry->d\_name.len, &sinfo);

    if (err)

        goto out;

    err = fat\_remove\_entries(dir, &sinfo); */\* and releases bh \*/*

    if (err)

        goto out;

    drop\_nlink(dir);

    clear\_nlink(inode);

    inode->i\_ctime = current\_time(inode);

    fat\_detach(inode);

out:

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    if (!err)

        err = fat\_flush\_inodes(sb, dir, inode);

    return err;

}

static int do\_msdos\_rename(struct inode \*old\_dir, unsigned char \*old\_name,

               struct dentry \*old\_dentry,

               struct inode \*new\_dir, unsigned char \*new\_name,

               struct dentry \*new\_dentry, int is\_hid)

{

    struct buffer\_head \*dotdot\_bh;

    struct msdos\_dir\_entry \*dotdot\_de;

    struct inode \*old\_inode, \*new\_inode;

    struct fat\_slot\_info old\_sinfo, sinfo;

    struct timespec ts;

    loff\_t new\_i\_pos;

    int err, old\_attrs, is\_dir, update\_dotdot, corrupt = 0;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    old\_sinfo.bh = sinfo.bh = dotdot\_bh = NULL;

    old\_inode = d\_inode(old\_dentry);

    new\_inode = d\_inode(new\_dentry);

    err = fat\_scan(old\_dir, old\_name, &old\_sinfo);

    if (err) {

        err = -EIO;

        goto out;

    }

    is\_dir = S\_ISDIR(old\_inode->i\_mode);

    update\_dotdot = (is\_dir && old\_dir != new\_dir);

    if (update\_dotdot) {

        if (fat\_get\_dotdot\_entry(old\_inode, &dotdot\_bh, &dotdot\_de)) {

            err = -EIO;

            goto out;

        }

    }

    old\_attrs = MSDOS\_I(old\_inode)->i\_attrs;

    err = fat\_scan(new\_dir, new\_name, &sinfo);

    if (!err) {

        if (!new\_inode) {

*/\* "foo" -> ".foo" case. just change the ATTR\_HIDDEN \*/*

            if (sinfo.de != old\_sinfo.de) {

                err = -EINVAL;

                goto out;

            }

            if (is\_hid)

                MSDOS\_I(old\_inode)->i\_attrs |= ATTR\_HIDDEN;

            else

                MSDOS\_I(old\_inode)->i\_attrs &= ~ATTR\_HIDDEN;

            if (IS\_DIRSYNC(old\_dir)) {

                err = fat\_sync\_inode(old\_inode);

                if (err) {

                    MSDOS\_I(old\_inode)->i\_attrs = old\_attrs;

                    goto out;

                }

            } else

                mark\_inode\_dirty(old\_inode);

            old\_dir->i\_version++;

            old\_dir->i\_ctime = old\_dir->i\_mtime = current\_time(old\_dir);

            if (IS\_DIRSYNC(old\_dir))

                (void)fat\_sync\_inode(old\_dir);

            else

                mark\_inode\_dirty(old\_dir);

            goto out;

        }

    }

    ts = current\_time(old\_inode);

    if (new\_inode) {

        if (err)

            goto out;

        if (is\_dir) {

            err = fat\_dir\_empty(new\_inode);

            if (err)

                goto out;

        }

        new\_i\_pos = MSDOS\_I(new\_inode)->i\_pos;

        fat\_detach(new\_inode);

    } else {

        err = msdos\_add\_entry(new\_dir, new\_name, is\_dir, is\_hid, 0,

                      &ts, &sinfo);

        if (err)

            goto out;

        new\_i\_pos = sinfo.i\_pos;

    }

    new\_dir->i\_version++;

    fat\_detach(old\_inode);

    fat\_attach(old\_inode, new\_i\_pos);

    if (is\_hid)

        MSDOS\_I(old\_inode)->i\_attrs |= ATTR\_HIDDEN;

    else

        MSDOS\_I(old\_inode)->i\_attrs &= ~ATTR\_HIDDEN;

    if (IS\_DIRSYNC(new\_dir)) {

        err = fat\_sync\_inode(old\_inode);

        if (err)

            goto error\_inode;

    } else

        mark\_inode\_dirty(old\_inode);

    if (update\_dotdot) {

        fat\_set\_start(dotdot\_de, MSDOS\_I(new\_dir)->i\_logstart);

        mark\_buffer\_dirty\_inode(dotdot\_bh, old\_inode);

        if (IS\_DIRSYNC(new\_dir)) {

            err = sync\_dirty\_buffer(dotdot\_bh);

            if (err)

                goto error\_dotdot;

        }

        drop\_nlink(old\_dir);

        if (!new\_inode)

            inc\_nlink(new\_dir);

    }

    err = fat\_remove\_entries(old\_dir, &old\_sinfo); */\* and releases bh \*/*

    old\_sinfo.bh = NULL;

    if (err)

        goto error\_dotdot;

    old\_dir->i\_version++;

    old\_dir->i\_ctime = old\_dir->i\_mtime = ts;

    if (IS\_DIRSYNC(old\_dir))

        (void)fat\_sync\_inode(old\_dir);

    else

        mark\_inode\_dirty(old\_dir);

    if (new\_inode) {

        drop\_nlink(new\_inode);

        if (is\_dir)

            drop\_nlink(new\_inode);

        new\_inode->i\_ctime = ts;

    }

out:

    brelse(sinfo.bh);

    brelse(dotdot\_bh);

    brelse(old\_sinfo.bh);

    return err;

error\_dotdot:

*/\* data cluster is shared, serious corruption \*/*

    corrupt = 1;

    if (update\_dotdot) {

        fat\_set\_start(dotdot\_de, MSDOS\_I(old\_dir)->i\_logstart);

        mark\_buffer\_dirty\_inode(dotdot\_bh, old\_inode);

        corrupt |= sync\_dirty\_buffer(dotdot\_bh);

    }

error\_inode:

    fat\_detach(old\_inode);

    fat\_attach(old\_inode, old\_sinfo.i\_pos);

    MSDOS\_I(old\_inode)->i\_attrs = old\_attrs;

    if (new\_inode) {

        fat\_attach(new\_inode, new\_i\_pos);

        if (corrupt)

            corrupt |= fat\_sync\_inode(new\_inode);

    } else {

*/\**

*\* If new entry was not sharing the data cluster, it*

*\* shouldn't be serious corruption.*

*\*/*

        int err2 = fat\_remove\_entries(new\_dir, &sinfo);

        if (corrupt)

            corrupt |= err2;

        sinfo.bh = NULL;

    }

    if (corrupt < 0) {

        fat\_fs\_error(new\_dir->i\_sb,

                 "%s: Filesystem corrupted (i\_pos %lld)",

                 \_\_func\_\_, sinfo.i\_pos);

    }

    goto out;

}

struct inode\_operations vfat\_dir\_inode\_operations

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,

};

Generic functions implemented

static struct dentry \*vfat\_lookup(struct inode \*dir, struct dentry \*dentry,

                  unsigned int flags)

{

    struct super\_block \*sb = dir->i\_sb;

    struct fat\_slot\_info sinfo;

    struct inode \*inode;

    struct dentry \*alias;

    int err;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    mutex\_lock(&MSDOS\_SB(sb)->s\_lock);

    err = vfat\_find(dir, &dentry->d\_name, &sinfo);

    if (err) {

        if (err == -ENOENT) {

            inode = NULL;

            goto out;

        }

        goto error;

    }

    inode = fat\_build\_inode(sb, sinfo.de, sinfo.i\_pos);

    brelse(sinfo.bh);

    if (IS\_ERR(inode)) {

        err = PTR\_ERR(inode);

        goto error;

    }

    alias = d\_find\_alias(inode);

*/\**

*\* Checking "alias->d\_parent == dentry->d\_parent" to make sure*

*\* FS is not corrupted (especially double linked dir).*

*\*/*

    if (alias && alias->d\_parent == dentry->d\_parent &&

        !vfat\_d\_anon\_disconn(alias)) {

*/\**

*\* This inode has non anonymous-DCACHE\_DISCONNECTED*

*\* dentry. This means, the user did ->lookup() by an*

*\* another name (longname vs 8.3 alias of it) in past.*

*\**

*\* Switch to new one for reason of locality if possible.*

*\*/*

        BUG\_ON(d\_unhashed(alias));

        if (!S\_ISDIR(inode->i\_mode))

            d\_move(alias, dentry);

        iput(inode);

        mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

        return alias;

    } else

        dput(alias);

out:

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    if (!inode)

        vfat\_d\_version\_set(dentry, dir->i\_version);

    return d\_splice\_alias(inode, dentry);

error:

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    return ERR\_PTR(err);

}

static int vfat\_create(struct inode \*dir, struct dentry \*dentry, umode\_t mode,

               bool excl)

{

    struct super\_block \*sb = dir->i\_sb;

    struct inode \*inode;

    struct fat\_slot\_info sinfo;

    struct timespec ts;

    int err;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    mutex\_lock(&MSDOS\_SB(sb)->s\_lock);

    ts = current\_time(dir);

    err = vfat\_add\_entry(dir, &dentry->d\_name, 0, 0, &ts, &sinfo);

    if (err)

        goto out;

    dir->i\_version++;

    inode = fat\_build\_inode(sb, sinfo.de, sinfo.i\_pos);

    brelse(sinfo.bh);

    if (IS\_ERR(inode)) {

        err = PTR\_ERR(inode);

        goto out;

    }

    inode->i\_version++;

    inode->i\_mtime = inode->i\_atime = inode->i\_ctime = ts;

*/\* timestamp is already written, so mark\_inode\_dirty() is unneeded. \*/*

    d\_instantiate(dentry, inode);

out:

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    return err;

}

static int vfat\_rmdir(struct inode \*dir, struct dentry \*dentry)

{

    struct inode \*inode = d\_inode(dentry);

    struct super\_block \*sb = dir->i\_sb;

    struct fat\_slot\_info sinfo;

    int err;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    mutex\_lock(&MSDOS\_SB(sb)->s\_lock);

    err = fat\_dir\_empty(inode);

    if (err)

        goto out;

    err = vfat\_find(dir, &dentry->d\_name, &sinfo);

    if (err)

        goto out;

    err = fat\_remove\_entries(dir, &sinfo); */\* and releases bh \*/*

    if (err)

        goto out;

    drop\_nlink(dir);

    clear\_nlink(inode);

    inode->i\_mtime = inode->i\_atime = current\_time(inode);

    fat\_detach(inode);

    vfat\_d\_version\_set(dentry, dir->i\_version);

out:

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    return err;

}

static int vfat\_unlink(struct inode \*dir, struct dentry \*dentry)

{

    struct inode \*inode = d\_inode(dentry);

    struct super\_block \*sb = dir->i\_sb;

    struct fat\_slot\_info sinfo;

    int err;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    mutex\_lock(&MSDOS\_SB(sb)->s\_lock);

    err = vfat\_find(dir, &dentry->d\_name, &sinfo);

    if (err)

        goto out;

    err = fat\_remove\_entries(dir, &sinfo); */\* and releases bh \*/*

    if (err)

        goto out;

    clear\_nlink(inode);

    inode->i\_mtime = inode->i\_atime = current\_time(inode);

    fat\_detach(inode);

    vfat\_d\_version\_set(dentry, dir->i\_version);

out:

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    return err;

}

static int vfat\_mkdir(struct inode \*dir, struct dentry \*dentry, umode\_t mode)

{

    struct super\_block \*sb = dir->i\_sb;

    struct inode \*inode;

    struct fat\_slot\_info sinfo;

    struct timespec ts;

    int err, cluster;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    mutex\_lock(&MSDOS\_SB(sb)->s\_lock);

    ts = current\_time(dir);

    cluster = fat\_alloc\_new\_dir(dir, &ts);

    if (cluster < 0) {

        err = cluster;

        goto out;

    }

    err = vfat\_add\_entry(dir, &dentry->d\_name, 1, cluster, &ts, &sinfo);

    if (err)

        goto out\_free;

    dir->i\_version++;

    inc\_nlink(dir);

    inode = fat\_build\_inode(sb, sinfo.de, sinfo.i\_pos);

    brelse(sinfo.bh);

    if (IS\_ERR(inode)) {

        err = PTR\_ERR(inode);

*/\* the directory was completed, just return a error \*/*

        goto out;

    }

    inode->i\_version++;

    set\_nlink(inode, 2);

    inode->i\_mtime = inode->i\_atime = inode->i\_ctime = ts;

*/\* timestamp is already written, so mark\_inode\_dirty() is unneeded. \*/*

    d\_instantiate(dentry, inode);

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    return 0;

out\_free:

    fat\_free\_clusters(dir, cluster);

out:

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    return err;

}

static int vfat\_rename(struct inode \*old\_dir, struct dentry \*old\_dentry,

               struct inode \*new\_dir, struct dentry \*new\_dentry,

               unsigned int flags)

{

    struct buffer\_head \*dotdot\_bh;

    struct msdos\_dir\_entry \*dotdot\_de;

    struct inode \*old\_inode, \*new\_inode;

    struct fat\_slot\_info old\_sinfo, sinfo;

    struct timespec ts;

    loff\_t new\_i\_pos;

    int err, is\_dir, update\_dotdot, corrupt = 0;

    struct super\_block \*sb = old\_dir->i\_sb;

    printk(KERN\_INFO "Access >> %s: %s\n",\_\_FILE\_\_,\_\_func\_\_);

    if (flags & ~RENAME\_NOREPLACE)

        return -EINVAL;

    old\_sinfo.bh = sinfo.bh = dotdot\_bh = NULL;

    old\_inode = d\_inode(old\_dentry);

    new\_inode = d\_inode(new\_dentry);

    mutex\_lock(&MSDOS\_SB(sb)->s\_lock);

    err = vfat\_find(old\_dir, &old\_dentry->d\_name, &old\_sinfo);

    if (err)

        goto out;

    is\_dir = S\_ISDIR(old\_inode->i\_mode);

    update\_dotdot = (is\_dir && old\_dir != new\_dir);

    if (update\_dotdot) {

        if (fat\_get\_dotdot\_entry(old\_inode, &dotdot\_bh, &dotdot\_de)) {

            err = -EIO;

            goto out;

        }

    }

    ts = current\_time(old\_dir);

    if (new\_inode) {

        if (is\_dir) {

            err = fat\_dir\_empty(new\_inode);

            if (err)

                goto out;

        }

        new\_i\_pos = MSDOS\_I(new\_inode)->i\_pos;

        fat\_detach(new\_inode);

    } else {

        err = vfat\_add\_entry(new\_dir, &new\_dentry->d\_name, is\_dir, 0,

                     &ts, &sinfo);

        if (err)

            goto out;

        new\_i\_pos = sinfo.i\_pos;

    }

    new\_dir->i\_version++;

    fat\_detach(old\_inode);

    fat\_attach(old\_inode, new\_i\_pos);

    if (IS\_DIRSYNC(new\_dir)) {

        err = fat\_sync\_inode(old\_inode);

        if (err)

            goto error\_inode;

    } else

        mark\_inode\_dirty(old\_inode);

    if (update\_dotdot) {

        fat\_set\_start(dotdot\_de, MSDOS\_I(new\_dir)->i\_logstart);

        mark\_buffer\_dirty\_inode(dotdot\_bh, old\_inode);

        if (IS\_DIRSYNC(new\_dir)) {

            err = sync\_dirty\_buffer(dotdot\_bh);

            if (err)

                goto error\_dotdot;

        }

        drop\_nlink(old\_dir);

        if (!new\_inode)

            inc\_nlink(new\_dir);

    }

    err = fat\_remove\_entries(old\_dir, &old\_sinfo); */\* and releases bh \*/*

    old\_sinfo.bh = NULL;

    if (err)

        goto error\_dotdot;

    old\_dir->i\_version++;

    old\_dir->i\_ctime = old\_dir->i\_mtime = ts;

    if (IS\_DIRSYNC(old\_dir))

        (void)fat\_sync\_inode(old\_dir);

    else

        mark\_inode\_dirty(old\_dir);

    if (new\_inode) {

        drop\_nlink(new\_inode);

        if (is\_dir)

            drop\_nlink(new\_inode);

        new\_inode->i\_ctime = ts;

    }

out:

    brelse(sinfo.bh);

    brelse(dotdot\_bh);

    brelse(old\_sinfo.bh);

    mutex\_unlock(&MSDOS\_SB(sb)->s\_lock);

    return err;

error\_dotdot:

*/\* data cluster is shared, serious corruption \*/*

    corrupt = 1;

    if (update\_dotdot) {

        fat\_set\_start(dotdot\_de, MSDOS\_I(old\_dir)->i\_logstart);

        mark\_buffer\_dirty\_inode(dotdot\_bh, old\_inode);

        corrupt |= sync\_dirty\_buffer(dotdot\_bh);

    }

error\_inode:

    fat\_detach(old\_inode);

    fat\_attach(old\_inode, old\_sinfo.i\_pos);

    if (new\_inode) {

        fat\_attach(new\_inode, new\_i\_pos);

        if (corrupt)

            corrupt |= fat\_sync\_inode(new\_inode);

    } else {

*/\**

*\* If new entry was not sharing the data cluster, it*

*\* shouldn't be serious corruption.*

*\*/*

        int err2 = fat\_remove\_entries(new\_dir, &sinfo);

        if (corrupt)

            corrupt |= err2;

        sinfo.bh = NULL;

    }

    if (corrupt < 0) {

        fat\_fs\_error(new\_dir->i\_sb,

                 "%s: Filesystem corrupted (i\_pos %lld)",

                 \_\_func\_\_, sinfo.i\_pos);

    }

    goto out;

}

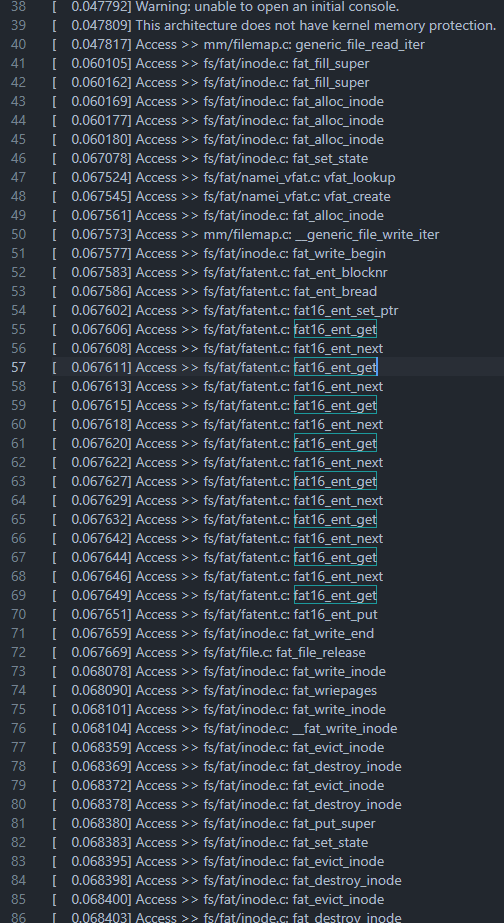
Μετά τις αλλαγές που έγιναν σε κάθε ένα από τα παραπάνω αρχεία και αφού εκτελέσαμε τις εντολές χρησιμοποιήσαμε διαφορετικά αρχεία (Διαφορετικού μεγέθους) στην εντολή 3 ώστε να δούμε διαφορετικά κατά την αποτελέσματα κατά την προσομοίωση .Τα αποτελέσματα της προσομοίωσης είναι τα εξής :

1.Μικρο αρχείο : maths.c αποτέλεσμα result2.txt

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, λογισμικό, λογισμικό πολυμέσων

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 1 maths.c



Κάνει αντιγραφή του maths.c στο νέο fs και ενημερώνονται οι εγγραφές για να αποθηκεύσουν όλα τα clusetrs που χρειάζεται το maths.c

Απελευθέρωση πόρων(μνήμης) που χρησιμοποιήθηκαν για την δημιουργία των κόμβων inode.

Αφορά λειτουργίες του superblock .Εισαγωγή πληροφοριών στο superblock δέσμευση μνήμης για τον κόμβο inode

Τελειώνει την εγγραφή

Εικόνα 2 Αποτελεσμα maths.c

2. Μεσαίο μέγεθος αρχείο : opsystem.pdf αποτέλεσμα result3.txt

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, σχεδίαση

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 3 Εισαγωγη/Αρχικοιποιηση μεταβλητων και δεσμευση μνημης για την δημιουργια inode που αφορα τα Superblock

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, κατάλογος

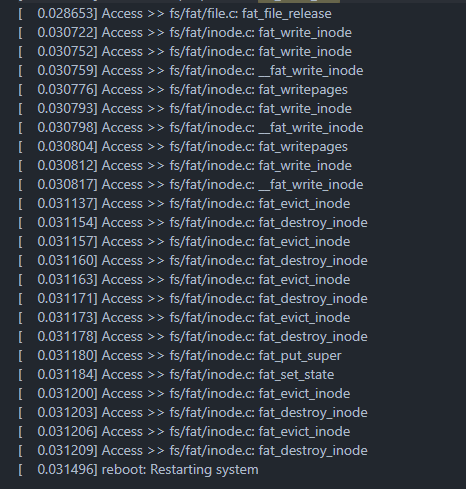
Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 4 1η αντιγραφη του opsystems.pdf στο νέο fs

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, κατάλογος, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 5 2η αντιγραφή του opsystems.pdf στο νέο fs



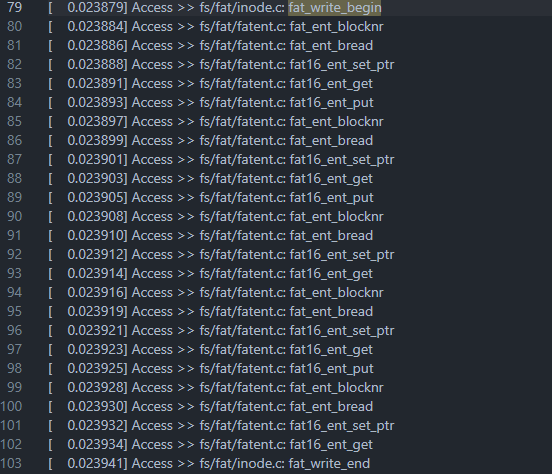
Εικόνα 6 Απελευθέρωση πόρων(μνήμης) που χρησιμοποιήθηκαν για την δημιουργία των κόμβων inode.

3.Μεγαλο αρχείο : lklfuse.c αποτέλεσμα result1.txt

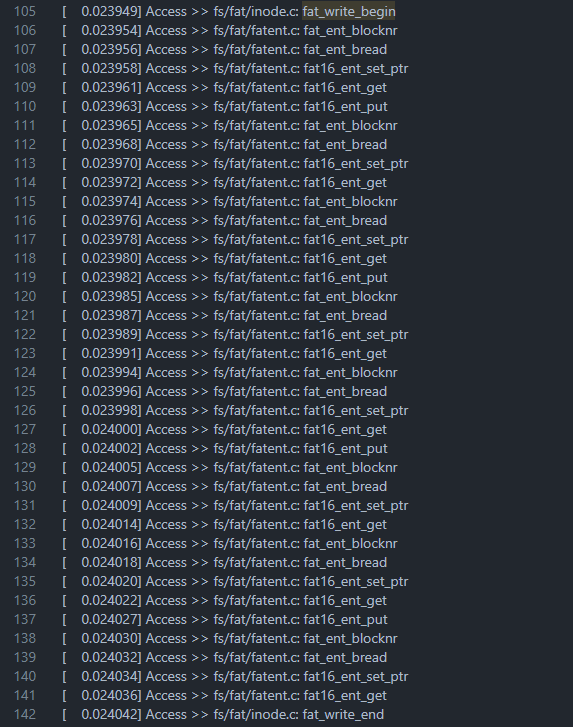
Εικόνα που περιέχει στιγμιότυπο οθόνης, κείμενο, μοτίβο, σχεδίαση

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 7 Αφορά λειτουργίες του superblock .Εισαγωγή πληροφοριών στο superblock δέσμευση μνήμης για τον κόμβο inode



Εικόνα 8 1η αντιγραφη του opsystems.pdf στο νέο fs



Εικόνα 9 2η αντιγραφη του opsystems.pdf στο νέο fs

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 10 3η αντιγραφη του opsystems.pdf στο νέο fs

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, κατάλογος

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 11 4η αντιγραφη του opsystems.pdf στο νέο fs

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα 12 Απελευθέρωση πόρων(μνήμης) που χρησιμοποιήθηκαν για την δημιουργία των κόμβων inode.

## **ΕΓΓΡΑΦΕΣ ΤΩΝ ΤΡΟΠΟΠΟΙΗΣΕΩΝ ΤΟΥ FAT**

**Ερώτημα**

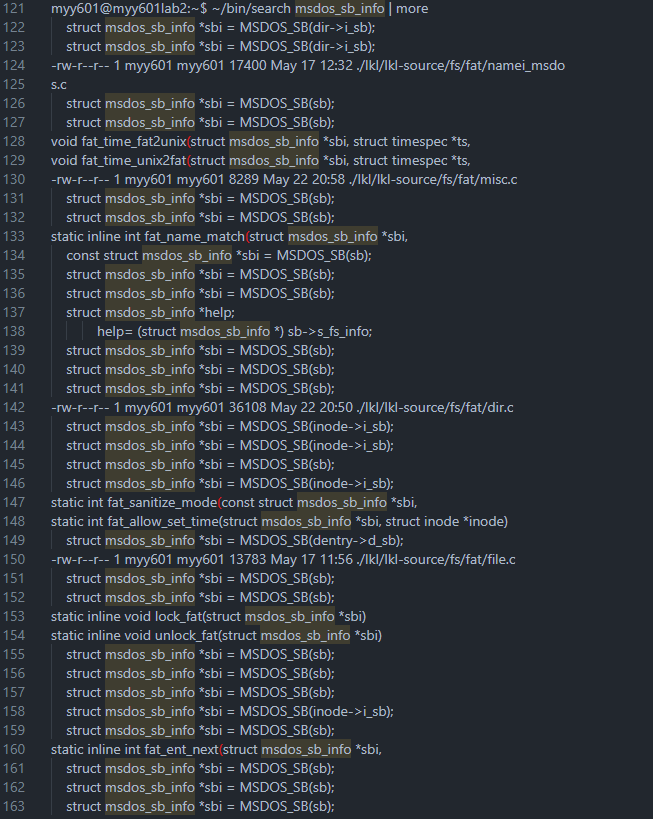
iii. Μια πιο ρεαλιστική υλοποίηση της καταγραφής θα χρησιμοποιήσει τις εσωτερικές κλήσεις του LKL (π.χ., ., sys\_open, sys\_write, κλπ) σε αρχείο (π.χ., /journal) που είναι αποθηκευμένο στο ίδιο το σύστημα FAT που παρακολουθείτε. Απαιτείται να καλέσετε τις συναρτήσεις αυτές από τον πηγαίο κώδικα της LKL σε θέσεις όπου πραγματοποιούνται οι τροποποιήσεις των δομών. Οι εγγραφές θα πρέπει να αντιστοιχούν σε εσωτερικές δομές του FAT όπως είναι το superblock, το file allocation table, τα directory entries και τα δεδομένα αρχείων.

**Απάντηση**

Για την καταγραφή των εσωτερικών κλήσεων μέσα στα αρχεία χρησιμοποιήσαμε τις συναρτήσεις οι οποίες βρίσκονται #include<linux/sys\_calls.h > και έχουν τις συναρτήσεις ο οποίες γράφουν με την sys\_write() στο αρχείο /journal και ανοίγει το αρχείο sys\_open() το journal .Έτσι , για να δούμε όμως τα αποτελέσματα που αντίστοιχων εσωτερικών δομών του fat όπως superblock, το file allocation table, τα directory entries και τα δεδομένα αρχείων του και το τι συμβαίνει μέσα σε κάθε συνάρτηση χρησιμοποιούμε printk() και sprintf() όπου γίνεται αλλαγή σε κάποια μεταβλητή της δομής .Στη συνέχεια θα υπάρξουν screenshot και εξήγηση τις κάθε συνάρτησης .

Superblock

Χρησιμοποιήσαμε την εντολή msdos\_sb\_info | more στο terminal ώστε να δούμε που χρησιμοποιείται η συγκεκριμένη μεταβλητή . Έτσι θα σημειώσουμε με όπου δούμε την msdos\_sb\_info να χρησιμοποιείται σε άλλο αρχείο .Μετα θα δείξουμε σε ποια αρχεία αυτή αλλάζει.





Μέσα στο /lkl/lkl-source/fs/fat/namei\_msdos.c βλέπουμε να δημιουργείται δυο τέτοια struct msdos\_sb\_info αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /lkl/lkl-source/fs/fat/misc.c βλέπουμε να δημιουργείται τέσσερα τέτοια struct msdos\_sb\_info αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /lkl/lkl-source/fs/fat/dir.c βλέπουμε να δημιουργείται και χρησιμοποιούνται 11 τέτοια struct msdos\_sb\_info αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /lkl/lkl-source/fs/fat/inode.c βλέπουμε να υπάρχουν σε συγκεκριμένα πεδία της δομής οπότε τώρα χρειάζεται να τα καταγράψουμε .Πιο συγκεκριμένα στην μέθοδο fat\_fill\_super().Οποτε τώρα χρειάζεται να χρησιμοποιήσουμε την sys\_write(),sys\_open για την καταγραφή των αλλαγών .

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, λογισμικό

Περιγραφή που δημιουργήθηκε αυτόματα

Αλλαγές πεδίων της msdos\_sb\_info

Μέσα στο /lkl/lkl-source/fs/fat/nfs.c βλέπουμε να δημιουργείται και χρησιμοποιούνται struct msdos\_sb\_info αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /lkl/lkl-source/fs/fat/cache.c βλέπουμε να δημιουργείται και χρησιμοποιούνται struct msdos\_sb\_info αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Επίσης μια δομή που σχετίζεται με τα superblock είναι η msdos\_fs.h η οποια βρισκεται στο /lkl/lkl-source/include/uapi/linux/msdos\_fs.h . Πιο συγκεκριμένα χρησιμοποιείται γιατί συνδέεται με τον superblock, που είναι μια δομή δεδομένων που αντιπροσωπεύει τα συνολικά metadata ενός συστήματος αρχείων αποθήκευση και διαχείριση πληροφοριών που αφορούν ένα σύστημα αρχείων FAT .

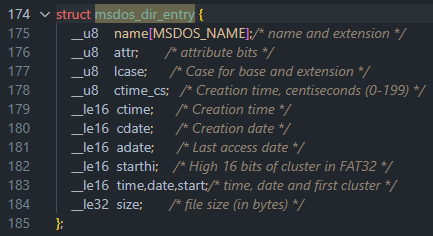
Directory Entry

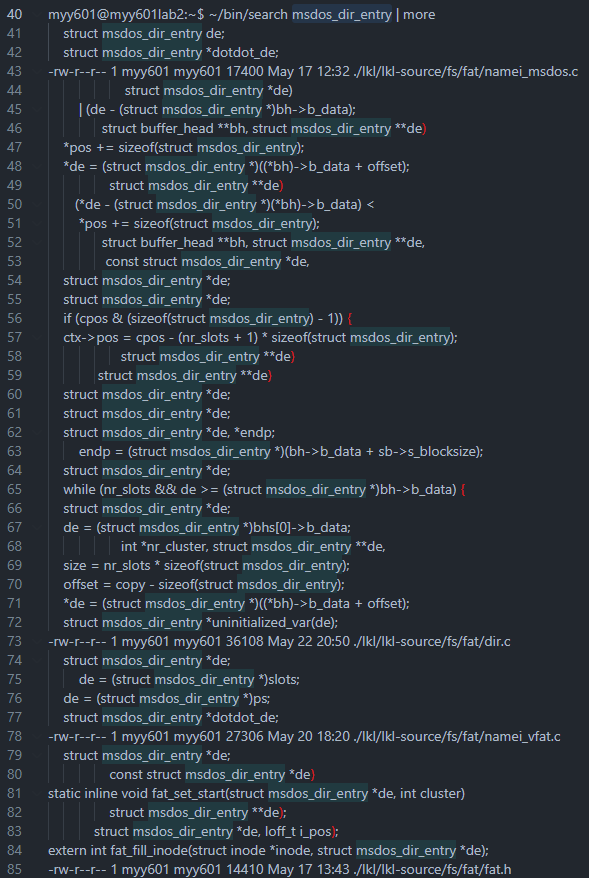
Χρησιμοποιήσαμε την εντολή fat\_slot\_info | more ώστε να δούμε που χρησιμοποιείται η συγκεκριμένη μεταβλητή . Έτσι θα σημειώσουμε με όπου δούμε την fat\_slot\_info να χρησιμοποιείται σε άλλο αρχείο .Μετα θα δείξουμε σε ποια αρχεία αυτή αλλάζει.

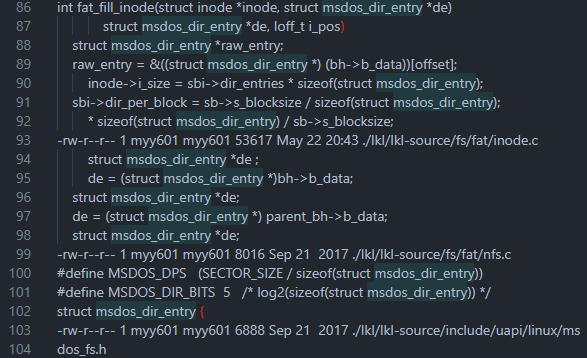
Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

Χρησιμοποιήσαμε την εντολή msdos\_dir\_entry| more ώστε να δούμε που χρησιμοποιείται η συγκεκριμένη μεταβλητή . Έτσι θα σημειώσουμε με όπου δούμε την msdos\_dir\_entry να χρησιμοποιείται σε άλλο αρχείο .Μετα θα δείξουμε σε ποια αρχεία αυτή αλλάζει. Χρησιμοποιείται για την αποθήκευση πληροφοριών σχετικά με ένα αρχείο ή φάκελο, όπως το όνομά του, τα χαρακτηριστικά του, το μέγεθος, τις χρονοσφραγίδες και τη θέση του μέσα στο αρχείο συστήματος.







Μέσα στο /lkl/lkl-source/fs/fat/namei\_msdos.c βλέπουμε να δημιουργείται και χρησιμοποιούνται struct msdos\_dir\_entry αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

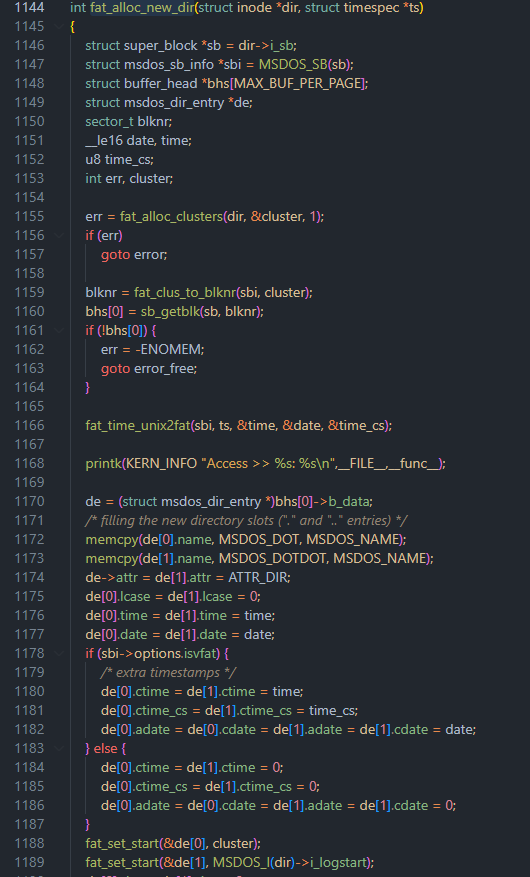
Μέσα στο /lkl/lkl-source/fs/fat/namei\_msdos.c βλέπουμε να δημιουργείται και χρησιμοποιούνται struct msdos\_dir\_entry αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /lkl/lkl-source/fs/fat/fat.h βλέπουμε να δημιουργείται και χρησιμοποιούνται struct msdos\_dir\_entry αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

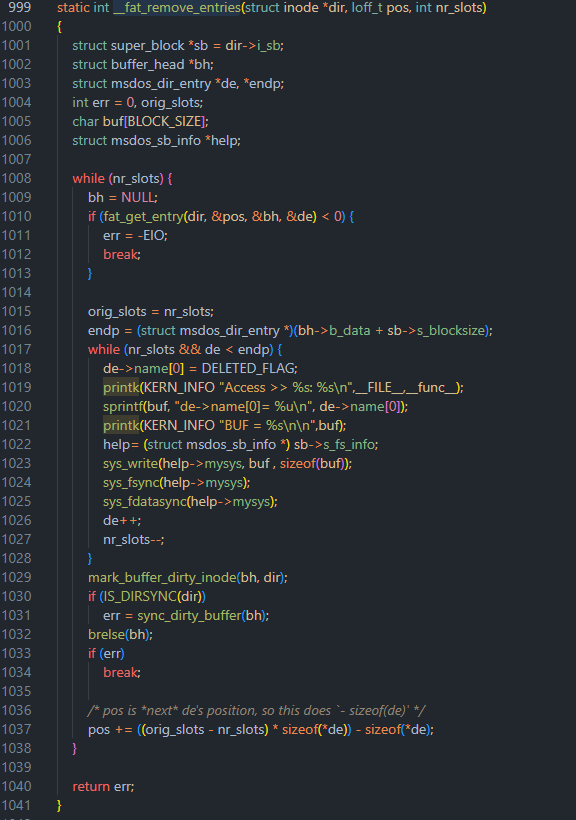
Μέσα στο /lkl/lkl-source/fs/fat/nfs.c βλέπουμε να δημιουργείται και χρησιμοποιούνται struct msdos\_dir\_entry αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /lkl/lkl-source/include/uapi/linux/msdos\_fs.h βλέπουμε να δημιουργείται και χρησιμοποιούνται struct msdos\_dir\_entry αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /lkl/lkl-source/fs/fat/dir.c βλέπουμε να υπάρχουν σε συγκεκριμένα πεδία της δομής οπότε τώρα χρειάζεται να τα καταγράψουμε .Πιο συγκεκριμένα στην μέθοδο\_\_fat\_remove\_entries() και στην fat\_alloc\_new\_dir() .Οπότε τώρα χρειάζεται να χρησιμοποιήσουμε την sys\_write(),sys\_open για την καταγραφή των αλλαγών .



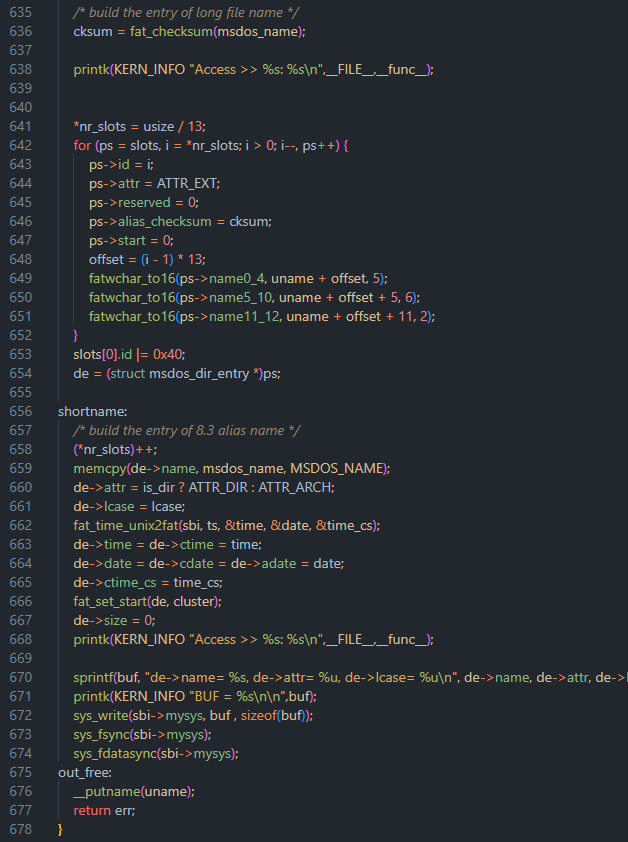
Αλλαγες σε πεδια της δομης msdos\_dir\_entry de.attr



Αλλαγες σε πεδια της δομης msdos\_dir\_entry de και endp

Μέσα στο ./lkl/lkl-source/fs/fat/namei\_vfat.c βλέπουμε να υπάρχουν σε συγκεκριμένα πεδία της δομής οπότε τώρα χρειάζεται να τα καταγράψουμε .Πιο συγκεκριμένα στην μέθοδο vfat\_build\_slots().Οπότε τώρα χρειάζεται να χρησιμοποιήσουμε την sys\_write(),sys\_open για την καταγραφή των αλλαγών .





Αλλαγες σε πεδια της δομης msdos\_dir\_entry de όπως attr,lcase …

Μέσα στο /lkl/lkl-source/fs/fat/inode.c βλέπουμε να υπάρχουν σε συγκεκριμένα πεδία της δομής οπότε τώρα χρειάζεται να τα καταγράψουμε .Πιο συγκεκριμένα στην μέθοδο \_\_fat\_write\_inode() . Οπότε τώρα χρειάζεται να χρησιμοποιήσουμε την sys\_write(),sys\_open για την καταγραφή των αλλαγών . 

Αλλαγες σε πεδια της δομης msdos\_dir\_entry raw\_entry ->size , raw\_entry ->attr.

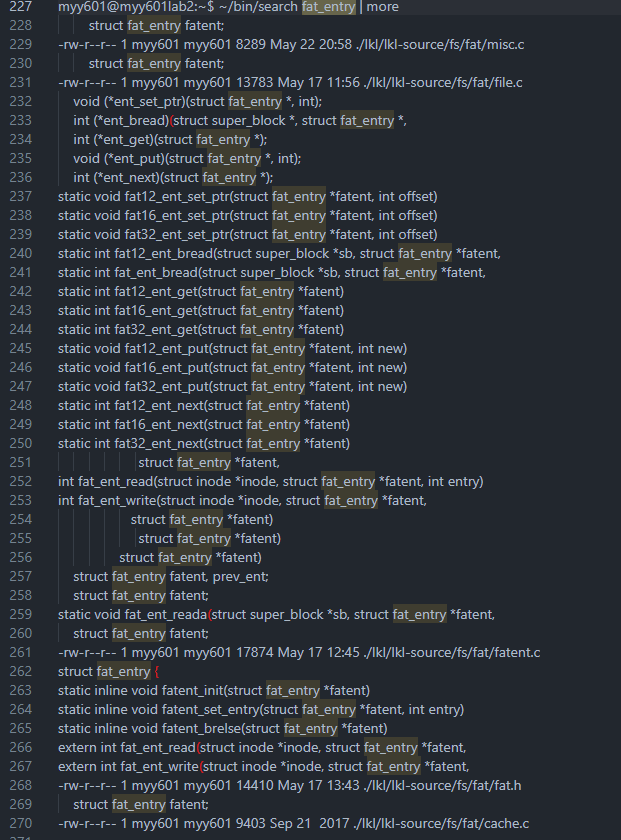
Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

File Allocation Table

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, λογισμικό

Περιγραφή που δημιουργήθηκε αυτόματα



Μέσα στο /lkl/lkl-source/fs/fat/misc.c βλέπουμε να δημιουργείται struct fat\_entry αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /lkl/lkl-source/fs/fat/file.c βλέπουμε να δημιουργείται struct fat\_entry αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /lkl/lkl-source/fs/fat/fatetnt.c βλέπουμε να δημιουργείται struct fat\_entry αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /lkl/lkl-source/fs/fat/cache.c βλέπουμε να δημιουργείται struct fat\_entry αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Επίσης μια δομή που σχετίζεται με τα file allocation table είναι η msdos\_fs.h η οποια βρισκεται στο /lkl/lkl-source/include/uapi/linux/msdos\_fs.h . Πιο συγκεκριμένα αναφέρεται στο αρχείο συστή-ματος. Πιο συγκεκριμενα ο κώδικας msdos\_fs είναι υπεύθυνος για την χειρισμό των λειτουργιών που σχετίζονται με την προσάρτηση, ανάγνωση και εγγραφή δεδομένων σε αποθηκευτικές συσκευές που έχουν μορφοποιηθεί με το αρχείο συστήματος του MS-DOS. Είναι ένα απλό και απευθείας αρχείο συστήματος που χρησιμοποιεί μια πίνακα κατανομής αρχείων για να παρακολουθεί την τοποθεσία των αρχείων στον δίσκο.

Οι δομές που σχετίζονται πιο συγκεκριμένα με κάποια είναι οι εξης : \_\_fat\_dirent,fat\_boot\_ struct

1. \_\_fat\_dirent: Αυτή η δομή αναπαριστά μια καταχώρηση καταλόγου στο αρχείο συστήματος FAT. Περιλαμβάνει πληροφορίες σχετικά με ένα συγκεκριμένο αρχείο ή φάκελο, όπως το όνομά του, τα χαρακτηριστικά του, το μέγεθός του και την τοποθεσία του στον δίσκοsector,fat\_boot\_fsinfo

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

1. fat\_boot\_sector: Αυτή η δομή αναπαριστά τον τομέα εκκίνησης του αρχείου συστήματος FAT. Ο τομέας εκκίνησης είναι ο πρώτος τομέας ενός δίσκου ή μιας διαμέρισης και περιέχει σημαντικές πληροφορίες για την αρχικοποίηση του αρχείου συστήματος. Περιέχει λεπτομέρειες όπως τον τύπο του αρχείου συστήματος, τον αριθμό των πινάκων κατανομής αρχείων, το μέγεθος κάθε πίνακα κατανομής αρχείων και άλλες παράμετρους που απαιτούνται για τη σωστή λειτουργία του αρχείου συστήματος.

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά

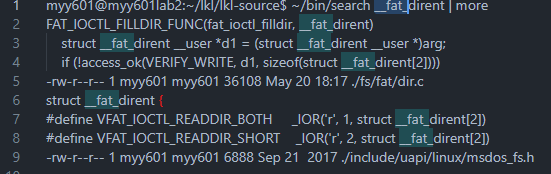
Περιγραφή που δημιουργήθηκε αυτόματα

fat\_boot\_fsinfo: Αυτή η δομή αναπαριστά τον τομέα πληροφοριών του συστήματος αρχείων (FSInfo) στο αρχείο συστήματος FAT. Ο τομέας FSInfo περιλαμβάνει πρόσθετες πληροφορίες σχετικά με το αρχείο συστήματος, όπως ο αριθμός των ελεύθερων κατανομών, η τελευταία διατεθείσα κατανομή και άλλα στατιστικά στοιχεία. Λειτουργεί ως μνήμη cache για συχνά χρησιμοποιούμενες πληροφορίες του αρχείου συστήματος, επιτρέποντας ταχύτερες λειτουργίες στο αρχείο συστήματος FAT.

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, αριθμός

Περιγραφή που δημιουργήθηκε αυτόματα

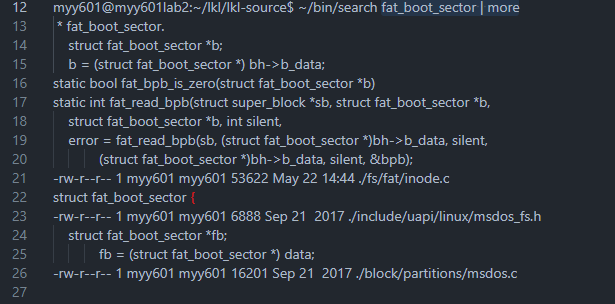
Χρησιμοποιήσαμε την εντολή \_\_fat\_dirent| more ώστε να δούμε που χρησιμοποιείται η συγκεκριμένη μεταβλητή . Έτσι θα σημειώσουμε με όπου δούμε την \_\_fat\_dirent να χρησιμοποιείται σε άλλο αρχείο .Μετα θα δείξουμε σε ποια αρχεία αυτή αλλάζει.



Μέσα στο /fs/fat/dir.c βλέπουμε να δημιουργείται struct \_\_fat\_dirent αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /include/uapi/linux/msdos\_fs.h βλέπουμε να δημιουργείται struct \_\_fat\_dirent αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

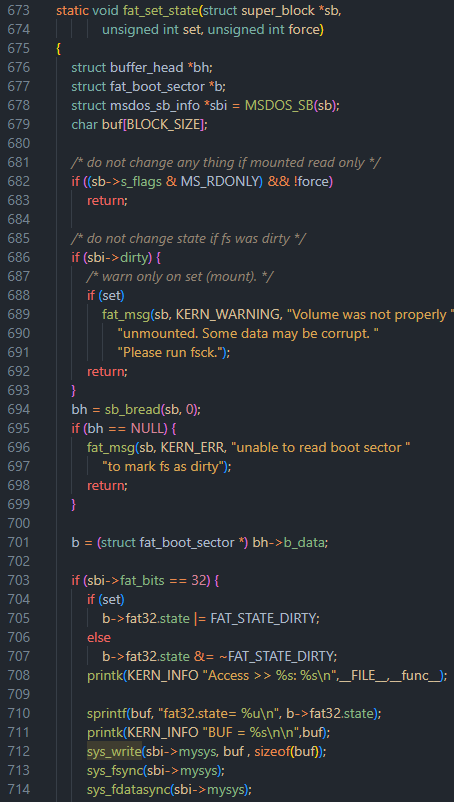
Χρησιμοποιήσαμε την εντολή fat\_boot\_sector | more ώστε να δούμε που χρησιμοποιείται η συγκεκριμένη μεταβλητή . Έτσι θα σημειώσουμε με όπου δούμε την fat\_boot\_sector να χρησιμοποιείται σε άλλο αρχείο .Μετα θα δείξουμε σε ποια αρχεία αυτή αλλάζει.



Μέσα στο /include/uapi/linux/msdos\_fs.h βλέπουμε να δημιουργείται struct fat\_boot\_sector αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

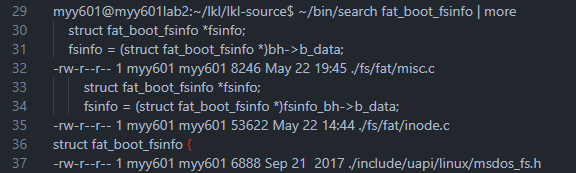
Μέσα στο /block/partitions/msdos.c βλέπουμε να δημιουργείται struct fat\_boot\_sector αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /lkl/lkl-source/fs/fat/inode.c βλέπουμε να υπάρχουν αλλαγες σε συγκεκριμένα πεδία της δομής οπότε τώρα χρειάζεται να τα καταγράψουμε .Πιο συγκεκριμένα στην μέθοδο fat\_set\_state().Οποτε τώρα χρειάζεται να χρησιμοποιήσουμε την sys\_write(),sys\_open για την καταγραφή των αλλαγών .



Αλλαγες στα πεδια της δομης fat\_boot\_sector

Χρησιμοποιήσαμε την εντολή fat\_boot\_fsinfo | more ώστε να δούμε που χρησιμοποιείται η συγκεκριμένη μεταβλητή . Έτσι θα σημειώσουμε με όπου δούμε την fat\_boot\_fsinfo να χρησιμοποιείται σε άλλο αρχείο .Μετα θα δείξουμε σε ποια αρχεία αυτή αλλάζει.

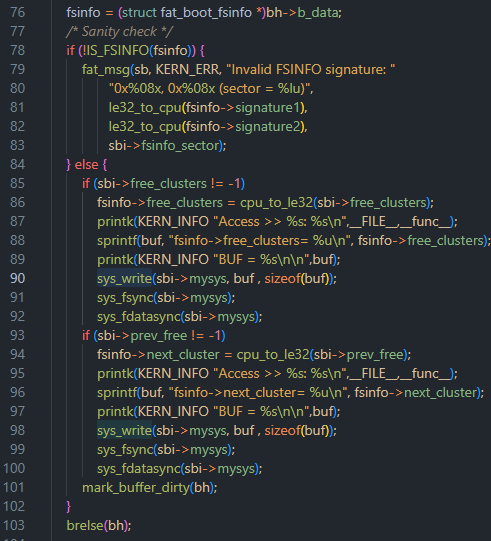


Μέσα στο /fs/fat/inode.c βλέπουμε να δημιουργείται struct fat\_boot\_fsinfo αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

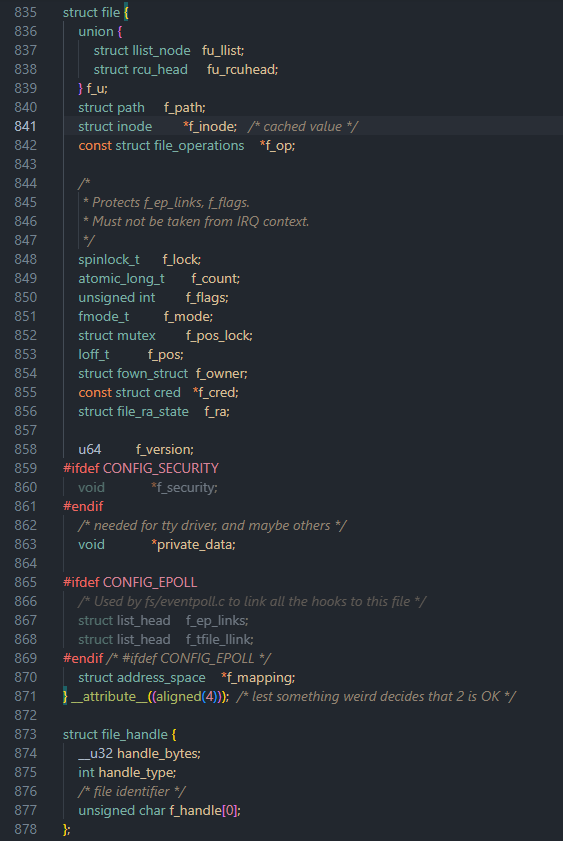
Μέσα στο /include/uapi/linux/msdos\_fs.h βλέπουμε να δημιουργείται struct fat\_boot\_fsinfo αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο ./fs/fat/misc.c βλέπουμε να υπάρχουν αλλαγές σε συγκεκριμένα πεδία της δομής οπότε τώρα χρειάζεται να τα καταγράψουμε .Πιο συγκεκριμένα στην μέθοδο fat\_clusters\_flush().Οποτε τώρα χρειάζεται να χρησιμοποιήσουμε την sys\_write,sys\_open για την καταγραφή των αλλαγών .

Αλλαγές στα πεδία της δομής fat\_boot\_fsinfo

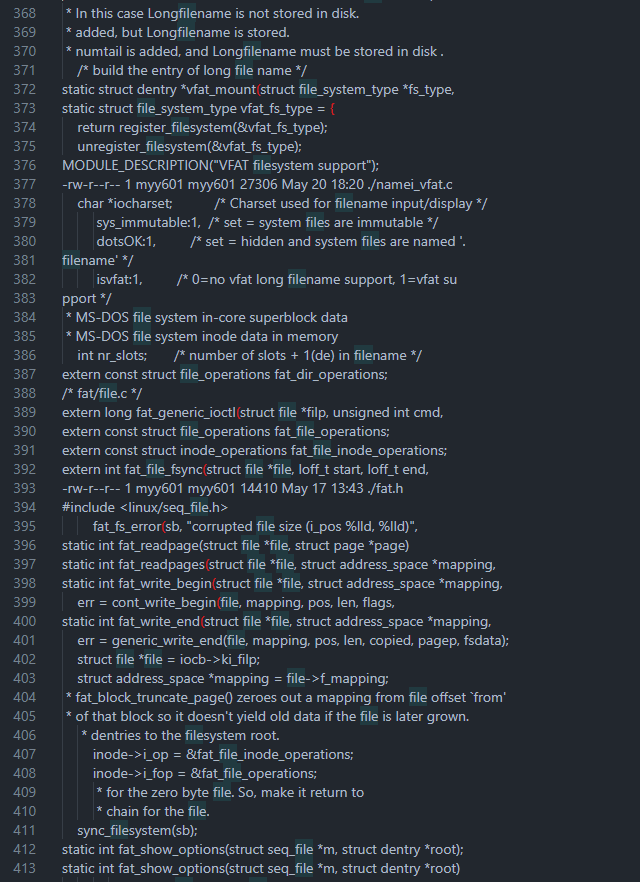


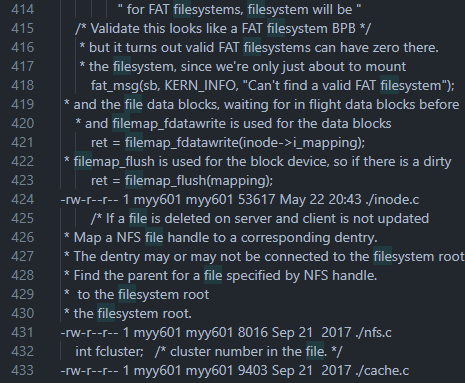
File











Επίσης μια δομή που σχετίζεται με τα file είναι η msdos\_dir\_slot.h η οποια βρισκεται στο /lkl/lkl-source/include/uapi/linux/msdos\_fs.h . Πιο συγκεκριμενα είναι μια δομή δεδομένων που χρησιμοποιείται στο αρχείο συστήματος MS-DOS για να αναπαριστά μια καταχώρηση καταλόγου μέσα σε έναν φάκελο. Χρησιμοποιείται ειδικά στα αρχεία συστήματος FAT12 και FAT16.

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά

Περιγραφή που δημιουργήθηκε αυτόματα

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, λογισμικό

Περιγραφή που δημιουργήθηκε αυτόματα

Μέσα στο /lkl/lkl-source/fs/fat/dir.c βλέπουμε να δημιουργείται msdos\_dir\_slot αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /lkl/lkl-source/include/uapi/linux/msdos\_fs.h βλέπουμε να δημιουργείται msdos\_dir\_slot αλλά δεν αλλάζει κάτι που περιέχει η δομή . Αρά , δεν χρειάζεται να καταγράψουμε κάτι μέσα στο αρχείο journal με τις συναρτήσεις sys\_write , sys\_open.

Μέσα στο /lkl/lkl-source/fs/fat/namei\_vfat.c βλέπουμε να υπάρχουν αλλαγες σε συγκεκριμένα πεδία της δομής οπότε τώρα χρειάζεται να τα καταγράψουμε .Πιο συγκεκριμένα στην μέθοδο vfat\_build\_slots() .Οποτε τώρα χρειάζεται να χρησιμοποιήσουμε την sys\_write(),sys\_open για την καταγραφή των αλλαγών .

Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, λογισμικό

Περιγραφή που δημιουργήθηκε αυτόματα

Αλλαγές στα πεδία της δομής msdos\_dir\_slot π.χ. το ps->id , ps->attrr …

## **ΕΛΕΓΧΟΣ ΕΓΓΡΑΦΩΝ**

**Ερώτημα**

Είστε υπεύθυνοι να υλοποιήσετε ελέγχους που δείχνουν ότι οι εγγραφές αντιστοιχούν στις αναμενόμενες τροποποιήσεις από διάφορες εκτελέσεις εφαρμογών που προκαλούν δημιουργία αρχείων και καταλόγων στο σύστημα αρχείων FAT (π.χ., χρησιμοποιώντας tests/boot ή cptofs).

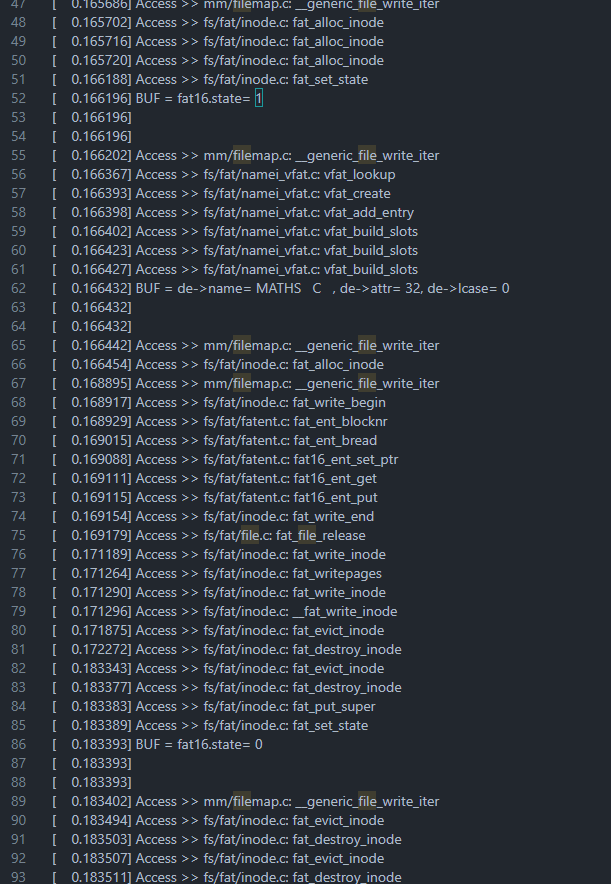
**Απάντηση**

Για την υλοποίηση θα χρησιμοποιήσουμε πάλι την εντολη 3 ./cptofos -i /tmp/vfatfile -p -t vfat file / ώστε να δούμε τα αποτελέσματα της εκτελεσης από διαφορα αρχεια με τα παλια printk του ερωτηματος 1 και τα καινουρια του ερωτηματος . Επισης , χρησιμοποιουμε την >> ώστε να τα κρατησουμε τα αποτελεσματα σε ένα αρχειο txt.

Θα χρησιμοποιησουμε παλι τα αρχεια τα οποια ειχαμε χρησιμοποιησουμε και προηγουμενως δηλάδη το maths.c , lkl\_fuse.c , opsystem.pdf .

Μικρο αρχειο maths.c





## **ΕΛΕΓΧΟΣ ΕΓΓΡΑΦΩΝ**

**Ερώτημα**

Είστε υπεύθυνοι να υλοποιήσετε ελέγχους που δείχνουν ότι οι εγγραφές αντιστοιχούν στις αναμενόμενες τροποποιήσεις από διάφορες εκτελέσεις εφαρμογών που προκαλούν δημιουργία αρχείων και καταλόγων στο σύστημα αρχείων FAT (π.χ., χρησιμοποιώντας tests/boot ή cptofs).

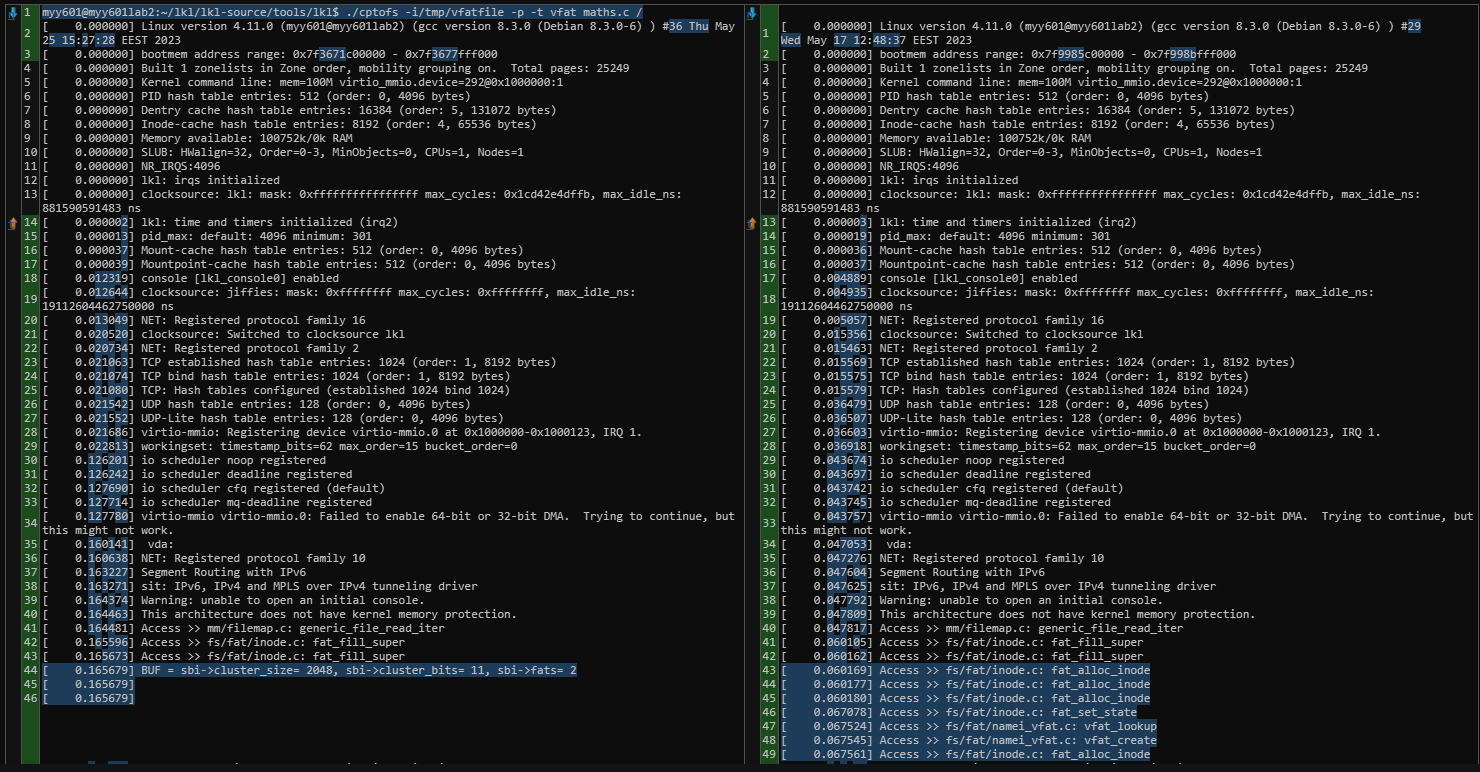
**Απάντηση**

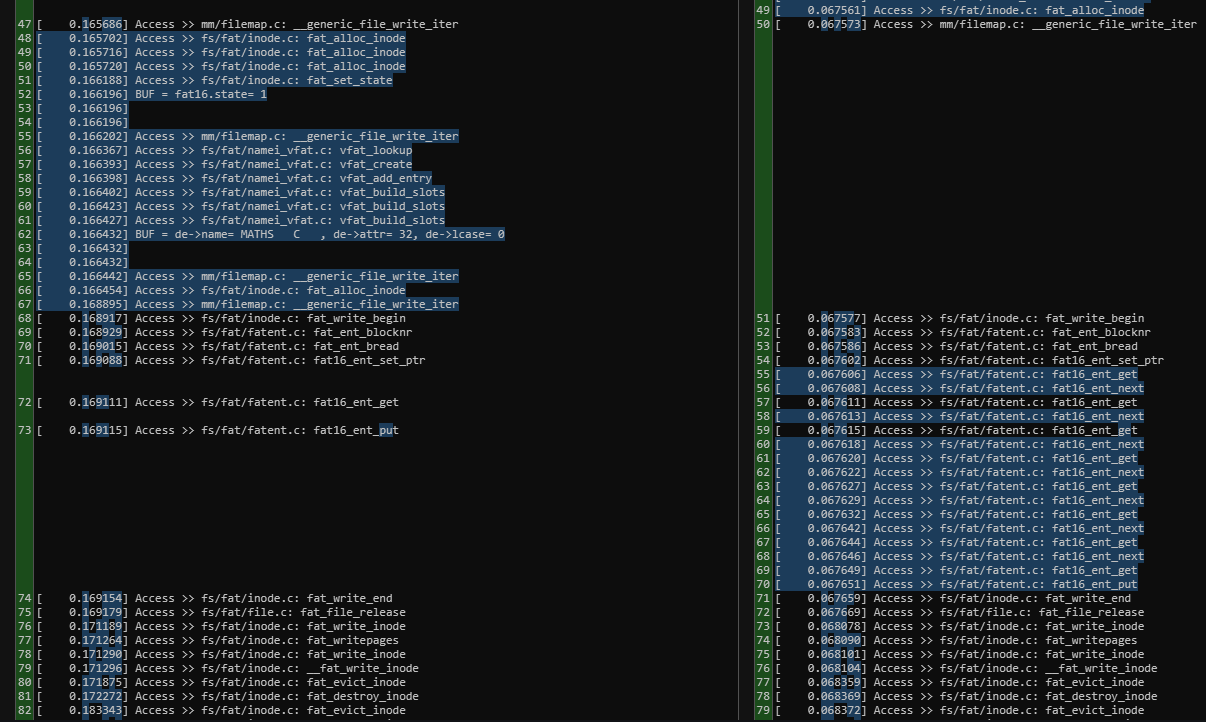
Για την υλοποίηση θα χρησιμοποιήσουμε πάλι την εντολη 3 ./cptofos -i /tmp/vfatfile -p -t vfat file / ώστε να δούμε τα αποτελέσματα της εκτελεσης από διαφορα αρχεια με τα παλια printk του ερωτηματος 1 και τα καινουρια του ερωτηματος . Επισης , χρησιμοποιουμε την >> ώστε να τα κρατησουμε τα αποτελεσματα σε ένα αρχειο txt.

Θα χρησιμοποιησουμε τα αρχεια που ειχαμε και στο πρωτο ερωτημα δηλαδη το maths.c,lklfuse.c,opsystems.pdf

Μετα θα βαλουμε τα δυο αρχειο σε ένα πρόγαμά τα οποία δειχνει τις αλλαγες τις οποιες εχουν μεταξυ τους τα δυο αρχεια .

Το πρωτο αρχειο που θα συγκρινουμε μεταξυ τους είναι το maths.c. Παρατηρουμε ότι εχουν γινει πολλες αλλαγες σε σχεση με πριν . Αυτές αφορουν τα τα οποια εχουμε βαλει sprintf() οπου εκει ελεγχουμε καποιες τιμες από καποιες συγκεκριμενες δομες οι οποιες μας ενδιαφερουν . Κυριως αφορουν superblock, το file allocation table, τα directory entries και τα δεδομένα αρχείων;





Εικόνα που περιέχει κείμενο, στιγμιότυπο οθόνης, γραμματοσειρά, λογισμικό πολυμέσων

Περιγραφή που δημιουργήθηκε αυτόματα