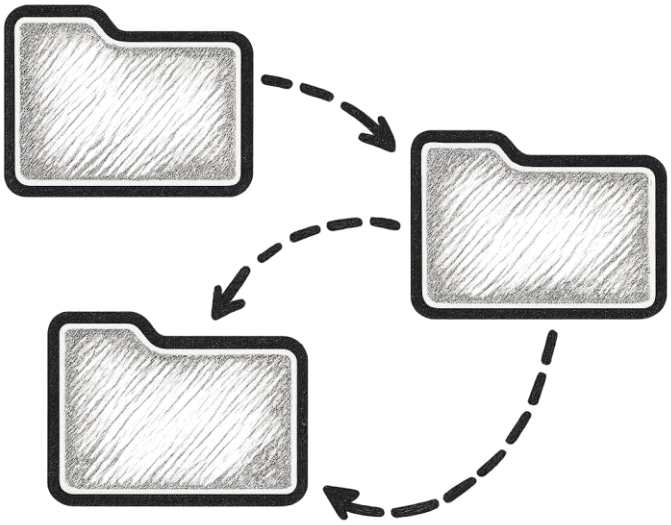
Εργαστήριο 2: Υλοποίηση αρχείου καταγραφής στο σύστημα αρχείων VFAT του Linux



Βασίλειος Παπακυριακού (5324)

Παναγιώτης Θεωδορόπουλος (5230)

Ηλίας Ναούμ (5431)

Αρχεία που τροποποιήσαμε

- buffer.c
- cache.c
- dir.c
- fatent.c
- fat.h
- inode.c
- splice.c
- filemap.c
- namei_vfat.c
- file.c
- read_write.c

Αλλαγές που πραγματοποιήσαμε

- Τοποθέτηση printk() για ιχνηλάτηση των συναρτήσεων εκτέλεσης του VFAT
- Δημιουργία αρχείου καταγραφής journal στο τοπικό σύστημα αρχείων
- Καταγραφή σημαντικών αλλαγών σε πεδία των βασικών δομών του VFAT

Περιγραφή της LKL

Η **Linux Kernel Library** (LKL) είναι μια βιβλιοθήκη που μας επιτρέπει να εκτελούμε λειτουργίες του πυρήνα του Linux από το user space, χωρίς χρειάζεται να “πειράξουμε” απευθείας τον πυρήνα του λειτουργικού συστήματος. Η χρήση της είναι κυρίως για πειραματισμό, δοκιμές και βελτιστοποίηση συστημάτων αρχείων και drivers (όπως το FAT), προσφέροντάς μας ένα ασφαλές περιβάλλον ανάπτυξης.

Μερικά βασικά πλεονεκτήματα της LKL:

Απόδοση: Οι κλήσεις συστήματος εκτελούνται απευθείας από user space, αποφεύγοντας το overhead της μετάβασης σε kernel mode, κάτι που είχε ως αποτέλεσμα μικρότερο χρόνο απόκρισης στις δοκιμές μας.

Ανεξαρτησία από αλλαγές του πυρήνα: Δεν χρειάστηκε να επανεκκινήσουμε ή να επανασυντάξουμε τον kernel κάθε φορά που κάναμε μια αλλαγή, κάτι που εξοικονόμησε σημαντικό χρόνο.

Ασφάλεια: Η δυνατότητα να δουλεύουμε εκτός του ίδιου του πυρήνα μάς έδωσε την ασφάλεια να κάνουμε πιο τολμηρές δοκιμές, χωρίς τον φόβο να γίνει ζημιά σε ολόκληρο το σύστημα.

Περιγραφή του FAT

Το FAT είναι ένα από τα παλαιότερα και ευρύτερα διαδεδομένα συστήματα αρχείων, το οποίο σχεδιάστηκε αρχικά από τη Microsoft για τα συστήματα MS-DOS και χρησιμοποιείται ακόμα ευρέως κυρίως σε φορητές συσκευές και σε ενσωματωμένα συστήματα. Χαρακτηρίζεται από την απλότητα του και τη συμβατότητα του με πολλά λειτουργικά συστήματα.

Δομή - αρχιτεκτονική

Το FAT οργανώνει τον δίσκο σε 4 κύριες περιοχές:

1. **Reserved sector:** Περιλαμβάνει τον Boot Sector και λοιπούς reserved τομείς. Ο Boot Sector περιέχει σημαντικές πληροφορίες για την εκκίνηση του συστήματος αρχείων, π.χ. το μέγεθος των sector, το μέγεθος των cluster και τον αριθμό FAT πινάκων.

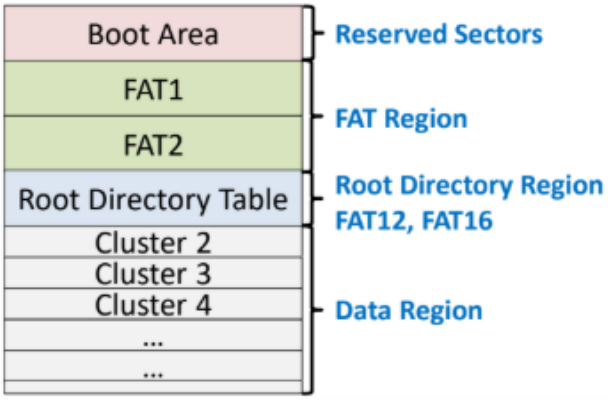
2. **FAT Region:** Περιέχει έναν ή περισσότερους πίνακες FAT (File Allocation Table). Κάθε πίνακας FAT είναι ουσιαστικά ένας πίνακας καταχωρήσεων για τα clusters του δίσκου. Κάθε εγγραφή δείχνει είτε τον αριθμό του επόμενου cluster είτε περιέχει μια ειδική τιμή (π.χ. 0xFFF για τέλος αρχείου, 0x000 για ελεύθερο cluster, 0xFF7 για κατεστραμμένο).

3. **Root Directory Region (για FAT12 και FAT16 μόνο):** Πρόκειται για μια στατικά κατανεμημένη περιοχή όπου αποθηκεύονται οι εγγραφές των αρχείων και των καταλόγων του root directory, με πληροφορίες όπως το όνομα, το μέγεθος, την ημερομηνία τελευταίας τροποποίησης και η αρχική θέση του στον FAT. Στο FAT32 αυτή η περιοχή αποθηκεύεται δυναμικά μέσα στο Data Region.

4. **Data Region:** Είναι το βασικό μέρος του δίσκου όπου αποθηκεύονται πραγματικά τα αρχεία και οι φάκελοι. Κάθε αρχείο διαιρείται σε μικρά κομμάτια που λέγονται

clusters και τα οποία είναι ομάδες από μικρότερα κομμάτια που λέγονται sectors. Έτσι τα δεδομένα κάθε αρχείου τοποθετούνται σε μια ή περισσότερες τέτοιες ομάδες στον δίσκο.

Στο σύστημα FAT, η αποθήκευση αρχείων γίνεται με τη δέσμευση ελεύθερων clusters, τα οποία συνδέονται μεταξύ τους μέσω του πίνακα FAT σαν αλυσίδα (linked list). Το Directory Entry (Dentry) κάθε αρχείου περιέχει μεταδεδομένα και τον αριθμό του πρώτου cluster. Για την ανάκτηση, η διαδικασία ξεκινά από αυτό το cluster και ακολουθείται η αλυσίδα μέσω του FAT μέχρι να εντοπιστεί το τέλος του αρχείου.



Πειραματική Μελέτη του VFAT μέσω LKL

Στο πλαίσιο της εργασίας μας, χρησιμοποιήσαμε την εικονική μηχανή Debian που μας δίνεται η οποία περιέχει τον πηγαίο κώδικα του Linux προσαρμοσμένο ώστε να εκτελείτε σε επίπεδο χρήστη μέσω της LKL. Ο στόχος μας ήταν η καταγραφή αλλαγών που πραγματοποιούνται στο σύστημα αρχείων VFAT ώστε να διασφαλιστεί η δυνατότητα επαναφοράς σε συνεπή κατάσταση σε περίπτωση δυσλειτουργίας. Αυτό επιτυγχάνεται με την αποθήκευση δεδομένων και μεταδεδομένων σε ειδικό αρχείο καταγραφής, ώστε το σύστημα να μπορεί να ανακτήσει την τελευταία έγκυρη κατάσταση.

Ο κώδικας της LKL είναι οργανωμένος σε φακέλους ανάλογα με τη λειτουργία που υλοποιεί κάθε τμήμα. Για παράδειγμα:

- ✓ Ο φάκελος fs/ περιλαμβάνει την υλοποίηση του VFAT και συναφών λειτουργιών συστήματος αρχείων.

- ✓ Ο φάκελος mm/ ασχολείται με τη διαχείριση μνήμης.

- ✓ Ο φάκελος tools/ έχει μέσα βοηθητικά εργαλεία για την επικοινωνία του LKL με άλλα στοιχεία όπως το vfatfile

Στη μελέτη μας βασιστήκαμε κυρίως στις φροντιστηριακές διαφάνειες και στις ενδεικτικές δομές που μας δίνονται στην εκφώνηση ώστε να εντοπίσουμε τα κρίσιμα σημεία στον κώδικα και να εισάγουμε τις κατάλληλες εντολές printk() ώστε να παρακολουθήσουμε σε πραγματικό χρόνο τη ροή εκτέλεσης και τις αλλαγές που πραγματοποιούνται στις κρίσιμες δομές.

Αρχικά εντοπίσαμε τα βασικά υποσυστήματα όπως **Superblock**, **Inode**, **εγγραφή αρχείων**, **κατάλογοι**, **μνήμη** και τα μελετήσαμε (περαιτέρω ανάλυση στη συνέχεια).

Βασικές δομές του VFS και οι λειτουργίες τους:

1. **Superblock:** Είναι η κεντρική δομή που περιέχει κρίσιμες πληροφορίες για το file system, όπως ο τύπος, το μέγεθος κλπ. Με βάση την εκφώνηση θα ασχοληθούμε με τις λειτουργίες της βασικής δομής superblock για το FAT οι οποίες βρίσκονται στο **struct super_operations fat_sops** στο αρχείο **inode.c** (fs/fat/inode.c):

    a. **alloc_inode = fat_alloc_inode**: Δεσμεύει και αρχικοποιεί ένα νέο inode για το FAT.
        - printk() με κόκκινο χρώμα στη γραμμή 789 στο αρχείο inode.c

    b. **free_inode = fat_free_inode**: Αποδεσμεύει και καθαρίζει ένα inode που δεν χρειάζεται πλέον.
        - printk() με κόκκινο χρώμα στη γραμμή 812 στο αρχείο inode.c

    c. **write_inode = fat_write_inode**: Γράφει την κατάσταση του inode από τη μνήμη πίσω στον δίσκο.
        - printk() με κόκκινο χρώμα στη γραμμή 977 στο αρχείο inode.c

    d. **evict_inode = fat_evict_inode**: Απομακρύνει ένα inode από την cache και καθαρίζει τους πόρους.
        - printk() με κόκκινο χρώμα στη γραμμή 686 στο αρχείο inode.c

    e. **put_super = fat_put_super**: Καλείται κατά το unmount για να απελευθερώσει πόρους του superblock.
        - printk() με κόκκινο χρώμα στη γραμμή 771 στο αρχείο inode.c

f. **statfs = fat_statfs**: Παρέχει στατιστικά στοιχεία του filesystem (π.χ. συνολικός αριθμός blocks και inodes).
   - printk() με κόκκινο χρώμα στη γραμμή 894 στο αρχείο inode.c

g. **remount_fs = fat_remount**: Χειρίζεται την αλλαγή επιλογών mount σε ήδη τρέχον FAT.
   - printk() με κόκκινο χρώμα στη γραμμή 872 στο αρχείο inode.c

h. **show_options = fat_show_options**: Εμφανίζει τις επιλογές mount που χρησιμοποιήθηκαν για το FAT.
   - printk() με κόκκινο χρώμα στη γραμμή 1018 στο αρχείο inode.c

2. **Μνήμη:** Οι λειτουργίες που σχετίζονται με τη διαχείριση της μνήμης (χώρου διευθύνσεων) για το FAT driver στο Linux είναι ορισμένες μέσα στη δομή **struct address_space_operations fat_aops**, η οποία βρίσκεται στο αρχείο **inode.c** (fs/fat/inode.c). Μερικές λειτουργίες βρέθηκαν και στο αρχείο **buffer.c** (fs/buffer.c):

   a. **dirty_folio = block_dirty_folio**: Δηλώνει ότι μια σελίδα έχει αλλάξει και πρέπει να αποθηκευτεί.
      - printk() με πράσινο χρώμα στη γραμμή 622 στο αρχείο buffer.c

   b. **invalidate_folio = block_invalidate_folio**: Απορρίπτει τα δεδομένα μιας σελίδας που δεν είναι πλέον χρήσιμα.
      - printk() με πράσινο χρώμα στη γραμμή 1491 στο αρχείο buffer.c

   c. **read_folio = fat_read_folio**: Φέρνει μια σελίδα δεδομένων από τον δίσκο στη μνήμη.
      - printk() με πράσινο χρώμα στη γραμμή 222 στο αρχείο inode.c

   d. **readahead = fat_readahead**: Διαβάζει προκαταβολικά επόμενες σελίδες για ταχύτερη πρόσβαση.
      - printk() με πράσινο χρώμα στη γραμμή 229 στο αρχείο inode.c

   e. **writepage = fat_writepage**: Στέλνει μία σελίδα από τη μνήμη πίσω στον δίσκο.
      - printk() με πράσινο χρώμα στη γραμμή 205 στο αρχείο inode.c

f. **writepages = fat_writepages**: Γράφει πολλές σελίδες μαζί στον δίσκο.
   - printk() με πράσινο χρώμα στη γραμμή 214 στο αρχείο inode.c

g. **write_begin = fat_write_begin**: Ξεκινά μια εγγραφή σε σελίδα.
   - printk() με πράσινο χρώμα στη γραμμή 253 στο αρχείο inode.c

h. **write_end = fat_write_end**: Ολοκληρώνει την εγγραφή και ενημερώνει τα δεδομένα.
   - printk() με πράσινο χρώμα στη γραμμή 272 στο αρχείο inode.c

i. **direct_IO = fat_direct_IO**: Επιτρέπει άμεση εγγραφή ή ανάγνωση, χωρίς χρήση της ενδιάμεσης cache.
   - printk() με πράσινο χρώμα στη γραμμή 294 στο αρχείο inode.c

j. **bmap = _fat_bmap**: Χαρτογραφεί έναν λογικό αριθμό block σε έναν φυσικό αριθμό block στο δίσκο, βοηθώντας στην καλύτερη ανάγνωση και εγγραφή στη μνήμη.
   - printk() με πράσινο χρώμα στη γραμμή 352 στο αρχείο inode.c

3. **Εγγραφές FAT:** Οι λειτουργίες που σχετίζονται με την επεξεργασία των εγγραφών FAT στον FAT driver του Linux υλοποιούνται μέσα από τη δομή **struct fatent_operations fat12/16/32_ops** του αρχείου **fs/fat/fatent.c**. Για σκοπούς απλοποίησης και λόγω της μεγάλης ομοιότητας μεταξύ τους (καθώς συχνά χρησιμοποιούν τα ίδια πεδία ή συναρτήσεις με παρόμοια υλοποίηση), επικεντρωνόμαστε στην ανάλυση του fat12_ops. Ωστόσο, για πλήρη κατανόηση του συστήματος, καταγράφουμε επίσης πού ακριβώς τοποθετούνται τα διαγνωστικά μηνύματα printk για όλα τα πεδία που υπάρχουν σε κάθε μία από τις παραλλαγές των παραπάνω struct.

   i. **struct fatent_operations fat12_ops** πεδία και λειτουργίες:
      a. **.ent_blocknr = fat12_ent_blocknr:** Επιστρέφει τον αριθμό του block στο οποίο αντιστοιχεί μια συγκεκριμένη εγγραφή FAT.
         - printk() με μπλε χρώμα στη γραμμή 34 στο αρχείο fatent.c
      b. **.ent_set_ptr = fat12_ent_set_ptr:** Αυτή η λειτουργία τοποθετεί τον δείκτη FAT στη σωστή θέση μέσα στην εγγραφή.
         - printk() με μπλε χρώμα στη γραμμή 62 στο αρχείο fatent.c

c. **.ent_bread = fat12_ent_bread:** Είναι μια λειτουργία που διαβάζει το μπλοκ του δίσκου όπου βρίσκεται μια συγκεκριμένη εγγραφή FAT.
   - printk() με μπλε χρώμα στη γραμμή 104 στο αρχείο fatent.c

d. **.ent_get = fat12_ent_get:** Αυτή η λειτουργία επιστρέφει την τρέχουσα τιμή της εγγραφής FAT στην οποία δείχνει ο δείκτης.
   - printk() με μπλε χρώμα στη γραμμή 163 στο αρχείο fatent.c

e. **.ent_put = fat12_ent_put:** Αυτή η λειτουργία αλλάζει την τιμή μιας εγγραφής FAT.
   - printk() με μπλε χρώμα στη γραμμή 212 στο αρχείο fatent.c

f. **.ent_next = fat12_ent_next:** Η λειτουργία αυτή μετακινεί τον δείκτη στην επόμενη εγγραφή FAT.
   - printk() με μπλε χρώμα στη γραμμή 263 στο αρχείο fatent.c

ii. **struct fatent_operations fat16_ops** τοποθέτηση printk():

a. .ent_blocknr=fat_ent_blocknr
   - printk() με μπλε χρώμα στη γραμμή 48 στο αρχείο fatent.c

b. .ent_set_ptr=fat16_ent_set_ptr
   - printk() με μπλε χρώμα στη γραμμή 81 στο αρχείο fatent.c

c. .ent_bread= fat_ent_bread
   - printk() με μπλε χρώμα στη γραμμή 142 στο αρχείο fatent.c

d. .ent_get = fat16_ent_get
   - printk() με μπλε χρώμα στη γραμμή 186 στο αρχείο fatent.c

e. .ent_put = fat16_ent_put
   - printk() με μπλε χρώμα στη γραμμή 238 στο αρχείο fatent.c

f. .ent_next = fat16_ent_next
   - printk() με μπλε χρώμα στη γραμμή 300 στο αρχείο fatent.c

iii. **struct fatent_operations fat32_ops** τοποθέτηση printk():

a. .ent_blocknr= fat_ent_blocknr
   - printk() με μπλε χρώμα στη γραμμή 48 στο αρχείο fatent.c

b. .ent_set_ptr= fat32_ent_set_ptr
   - printk() με μπλε χρώμα στη γραμμή 92 στο αρχείο fatent.c

c. .ent_bread= fat_ent_bread
   - printk() με μπλε χρώμα στη γραμμή 142 στο αρχείο fatent.c

d. .ent_get= fat32_ent_get
   - printk() με μπλε χρώμα στη γραμμή 199 στο αρχείο fatent.c

e. .ent_put= fat32_ent_put

- printk() με μπλε χρώμα στη γραμμή 251 στο αρχείο fatent.c

f. .ent_next= fat32_ent_next

- printk() με μπλε χρώμα στη γραμμή 316 στο αρχείο fatent.c

4. **Αρχείο (File):** Οι λειτουργίες αρχείου, για το FAT driver του Linux, ορίζονται στην δομή **struct file_operations fat_file_operations** του αρχείου **fs/fat/file.c**.

a. **.llseek = generic_file_llseek:** Αυτή είναι η λειτουργία που διαχειρίζεται την αλλαγή της τρέχουσας θέσης ανάγνωσης ή εγγραφής (γνωστής και ως "θέσης αρχείου") μέσα σε ένα αρχείο.
   - printk() με ανοιχτό μωβ χρώμα στη γραμμή 148 στο αρχείο read_write.c

b. **.read_iter = generic_file_read_iter:** Αυτή η λειτουργία διαβάζει δεδομένα από ένα αρχείο.
   - printk() με ανοιχτό μωβ χρώμα στη γραμμή 2775 στο αρχείο filemap.c

c. **.write_iter = generic_file_write_iter:** Αυτή η λειτουργία κάνει εγγραφή δεδομένων σε ένα αρχείο.
   - printk() με ανοιχτό μωβ χρώμα στη γραμμή 3827 στο αρχείο filemap.c

d. **.mmap = generic_file_mmap:** Αυτή η λειτουργία χειρίζεται την λειτουργία μνήμης σε αρχείο, που επιτρέπει σε ένα πρόγραμμα να απευθυνθεί στο περιεχόμενο ενός αρχείου σαν να ήταν μέρος της διεύθυνσης χώρου της μνήμης του.
   - printk() με ανοιχτό μωβ χρώμα στη γραμμή 3456 στο αρχείο filemap.c

e. **.unlocked_ioctl = fat_generic_ioctl:** Αυτή η λειτουργία διαχειρίζεται τα ειδικά IOCTL αιτήματα για το αρχείο.
   - printk() με ανοιχτό μωβ χρώμα στη γραμμή 160 στο αρχείο file.c

f. **.release = fat_file_release:** Κλείνει ένα αρχείο και ελευθερώνει ό,τι είχε δεσμευτεί γι' αυτό
   - printk() με ανοιχτό μωβ χρώμα στη γραμμή 183 στο αρχείο file.c

g. **.fsync = fat_file_fsync:** Αυτή η λειτουργία φροντίζει για τον συγχρονισμό του αρχείου, ώστε όλες οι αλλαγές να έχουν γραφτεί στον δίσκο

- printk() με ανοιχτό μωβ χρώμα στη γραμμή 199 στο αρχείο file.c

h. **.fallocate = fat_fallocate:** Αυτή η λειτουργία καθορίζει από πριν το μέγεθος ενός αρχείου, δεσμεύοντας τον απαραίτητο χώρο στον δίσκο.
   - printk() με ανοιχτό μωβ χρώμα στη γραμμή 278 στο αρχείο file.c

i. **.splice_read = generic_file_splice_read:** Αυτή η λειτουργία διαχειρίζεται την ανάγνωση δεδομένων από ένα αρχείο και τη μεταφορά τους απευθείας σε άλλο αρχείο, χωρίς να απαιτείται αντιγραφή τους στον χώρο χρήστη.
   - printk() με ανοιχτό μωβ χρώμα στη γραμμή 304 στο αρχείο splice.c

5. **Inode:** Μερικές λειτουργίες inode, για το FAT driver του Linux, ορίζονται στην δομή **struct inode_operations fat_file_inode_operations** του αρχείου **fs/fat/file.c**

   a. **.setattr = fat_setattr:** Αυτή η λειτουργία χρησιμοποιείται για να ρυθμίσει τις ιδιότητες ενός inode, όπως δικαιώματα πρόσβασης, ιδιοκτήτη, ομάδα, μέγεθος και ημερομηνίες.
      - printk() με γκρι χρώμα στη γραμμή 506 στο αρχείο file.c

   b. **.getattr = fat_getattr:** Αυτή η λειτουργία χρησιμοποιείται για να διαβάσει τις ιδιότητες ενός inode, δηλαδή όσα έχουν οριστεί μέσω της setattr**.**
      - printk() με γκρι χρώμα στη γραμμή 420 στο αρχείο file.c

6. **Directories:** Οι λειτουργίες των καταλόγων, για το FAT driver του Linux, ορίζονται στην δομή **struct inode_operations msdos_dir_inode_operations** του αρχείου **fs/fat/namei_msdos.c** για το σύστημα αρχείων FAT και **struct inode_operations vfat_dir_inode_operations** του αρχείου **fs/fat/namei_vfat.c** για το σύστημα αρχείων VFAT. Εμείς επειδή ασχολούμαστε με τον vfat driver, θα βάλουμε printk μόνο εκεί.

   a. **.create = vfat_create:** Αυτή η λειτουργία είναι υπεύθυνη για τη δημιουργία ενός νέου αρχείου εντός ενός καταλόγου.
      - printk() με κίτρινο χρώμα στη γραμμή 770 στο αρχείο namei_vfat.c

b. **.lookup = vfat_lookup:** Αυτή η λειτουργία χρησιμοποιείται για τον εντοπισμό ενός αρχείου ή υποκαταλόγου μέσα σε έναν κατάλογο, με βάση το όνομά του.
   - printk() με κίτρινο χρώμα στη γραμμή 708 στο αρχείο namei_vfat.c

c. **.unlink = vfat_unlink:** Αυτή η λειτουργία αφαιρεί έναν σύνδεσμο σε ένα αρχείο από έναν κατάλογο, διαγράφοντας το αρχείο εάν δεν υπάρχουν άλλοι σύνδεσμοι προς αυτό.
   - printk() με κίτρινο χρώμα στη γραμμή 839 στο αρχείο namei_vfat.c

d. **.mkdir = vfat_mkdir:** Αυτή η λειτουργία δημιουργεί ένα νέο κατάλογο μέσα σε έναν άλλο κατάλογο.
   - printk() με κίτρινο χρώμα στη γραμμή 870 στο αρχείο namei_vfat.c

e. **.rmdir = vfat_rmdir:** Αυτή η λειτουργία αφαιρεί έναν κατάλογο από έναν άλλο κατάλογο, με την προϋπόθεση ότι ο κατάλογος είναι κενός.
   - printk() με κίτρινο χρώμα στη γραμμή 804 στο αρχείο namei_vfat.c

f. **.rename = vfat_rename:** Αυτή η λειτουργία μετονομάζει ένα αρχείο ή κατάλογο σε έναν κατάλογο.
   - printk() με κίτρινο χρώμα στη γραμμή 958 στο αρχείο namei_vfat.c

g. **.setattr = fat_setattr:** Όπως προηγουμένως, αυτή η λειτουργία ορίζει τις ιδιότητες ενός inode
   - printk() με κίτρινο χρώμα στη γραμμή 506 στο αρχείο file.c

h. **.getattr = fat_getattr:** Όπως προηγουμένως, αυτή η λειτουργία λαμβάνει τις ιδιότητες ενός inode.
   - printk() με κίτρινο χρώμα στη γραμμή 420 στο αρχείο file.c

Εκτέλεση και Ανάλυση Εντολών για το VFAT στο LKL

Αξιοποιώντας την παραπάνω υποδομή, πραγματοποιήσαμε πειράματα στα οποία αντιγράψαμε διαφορετικά είδη αρχείων (π.χ. μικρά αρχεία κειμένου, πηγαίος κώδικας C, μεγάλα αρχεία) και καταγράψαμε τις αλλαγές που πραγματοποιήθηκαν. Κάναμε χρήση των εντολών boot, cptofs και cpfromfs ώστε να κατανοήσουμε καλύτερα τη λειτουργία του FAT. Ακολουθεί σύντομη περιγραφή της καθεμίας:

1. **boot**: Εκκινεί το lkl με το σύστημα αρχείων που του αναθέτουμε εμείς, στην περιπτωσή μας ένα VFAT image στο /tmp/disk. Ουσιαστικά φορτώνει το περιβάλλον που θα τρέξουν οι cptofs και cpfromfs που βλέπουμε παρακάτω.
2. **cptofs**: Αντιγράφει αρχεία από το δικό μας σύστημα αρχείων προς το εικονικό σύστημα, δηλαδή το FAT στο lkl.
3. **cpfromfs**: Αντιγράφει αρχεία από το εικονικό file system στο δικό μας του Debian.

Για να διεξάγουμε τις δοκιμές σωστά, θα εκτελέσουμε τις δύο τελευταίες εντολές με διάφορα σενάρια. Συγκεκριμένα, επιλέξαμε ένα μικρό αρχείο κειμένου, ένα μικρό αρχείο .c το οποίο τυπώνει “Hello World”, το αρχείο lklfuse.c το οποίο αποτελείται από περίπου 600 γραμμές κώδικα, ένα πολύ μεγάλο binary αρχείο 3MB και ένα directory με δύο μικρά αρχεία.

Αρχικά εκτελέσαμε “**make –j8 clean**” και “**make -j8**” για τη μεταγλώττιση της βιβλιοθήκης. Ύστερα θα δημιουργήσουμε και θα μορφοποιήσουμε το εικονικό αρχείο δίσκου (**/tmp/disk**) με την εντολή “**./disk.sh -t vfat**” στον φάκελο **/lkl-source/tools/lkl/tests**. Η εντολή αυτή είναι σαν να λέει στο σύστημα “φόρτωσε και δούλεψε με το σύστημα αρχείου VFAT”, γι αυτό και όταν την τρέχουμε αρχίζουμε να βλέπουμε μερικά από τα printk() που βάλαμε. Πιο συγκεκριμένα βλέπουμε να εκτελείται η **fat_alloc_inode** η οποία δημιουργεί inode για να εκπροσωπήσει έναν φάκελο ή ένα αρχείο με το οποίο θέλει να δουλέψει. Βλέπουμε ακόμα τη συνάρτηση **generic_file_read_iter** όπου εκτελείται για να διαβάσει μερικά αρχεία, πιθανώς για κάποιον έλεγχο.

```
  [    0.013246] FILE_OPS - Executing generic_file_read_iter
  lkl_start_kernel("mem=128M loglevel=8") = 0
 ...
* 3 mount_dev
ok 3 mount_dev
 ---
 time_us: 2748
 log: |
  [    0.013941] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
  [    0.013941]
  [    0.013946] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
  [    0.013946]
  [    0.013947] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
  [    0.013947]
  lkl_mount_dev(disk_id, cla.partition, cla.fstype, 0, NULL, mnt_point, sizeof(mnt_point)) = 0
```

```
 time_us: 12
 log: |
  [    0.027728] FILE_OPS - Executing generic_file_write_iter
  lkl_sys_write(0, wrbuf, sizeof(wrbuf)) = 5
 ...
* 14 lseek_cur
ok 14 lseek_cur
 ---
 time_us: 5
 log: |
  [    0.027772] FILE_OPS - Executing fat_fallocate
  lkl_sys_lseek(0, 0, LKL_SEEK_CUR) = 5
 ...
* 15 lseek_end
ok 15 lseek_end
 ---
 time_us: 4
 log: |
  [    0.027809] FILE_OPS - Executing fat_fallocate
  lkl_sys_lseek(0, 0, LKL_SEEK_END) = 5
 ...
* 16 lseek_set
ok 16 lseek_set
 ---
 time_us: 4
 log: |
  [    0.027864] FILE_OPS - Executing fat_fallocate
  lkl_sys_lseek(0, 0, LKL_SEEK_SET) = 0
 ...
* 17 read
ok 17 read
 ---
 time_us: 6
 log: |
  [    0.027901] FILE_OPS - Executing generic_file_read_iter
  lkl_sys_read=5 buf=test
```

Έπειτα θα συνεχίσουμε με την εκτέλεση της εντολής **boot** γράφοντας ” **tests/boot -t vfat -d /tmp/vfatfile -p -P 0**” στο τερματικό. Έτσι αρχικοποιείται το lkl και γίνεται mount το FAT. Κατά τη διαδικασία αυτή βλέπουμε επίσης κλήσεις συναρτήσεων που καταγράψαμε με printk(): Εκτελούνται οι **generic_file_write_iter**, η **fat_allocate** και η **file_read_iter** διότι στο αρχείο boot.c υπάρχουν συναρτήσεις οι οποίες τεστάρουν λειτουργίες όπως την read την write και την lseek για να δείξουν ότι το file system λειτουργεί σωστά.

```
0.025568] FILE_OPS - Executing generic_file_read_iter
0.025974] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.025974]
0.026045] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.026045]
0.026073] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.026073]
0.029378] VFAT_DIR_OPS - Executing vfat_lookup
0.029443] VFAT_DIR_OPS - Executing vfat_create
0.029490] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.029490]
0.029767] FILE_OPS - Executing generic_file_write_iter
0.029820] MEMORY_OPERATIONS - Executing fat_write_begin
0.029860] FAT_ENTRIES_OPS - Executing fat_ent_blocknr
0.029902] FAT_ENTRIES_OPS - Executing fat_ent_bread
0.030018] FAT_ENTRIES_OPS - Executing fat16_ent_set_ptr
0.030071] FAT_ENTRIES_OPS - Executing fat16_ent_get
0.030108] FAT_ENTRIES_OPS - Executing fat16_ent_put
0.030150] MEMORY_OPERATIONS - Executing fat_write_end
0.030193] FILE_OPS - Executing generic_file_write_iter
0.030230] MEMORY_OPERATIONS - Executing fat_write_begin
0.030269] MEMORY_OPERATIONS - Executing fat_write_end
0.030308] FILE_OPS - Executing generic_file_write_iter
0.030344] MEMORY_OPERATIONS - Executing fat_write_begin
0.030381] FAT_ENTRIES_OPS - Executing fat_ent_blocknr
0.030418] FAT_ENTRIES_OPS - Executing fat_ent_bread
0.030454] FAT_ENTRIES_OPS - Executing fat16_ent_set_ptr
0.030490] FAT_ENTRIES_OPS - Executing fat16_ent_get
0.030526] FAT_ENTRIES_OPS - Executing fat16_ent_put
0.030562] FAT_ENTRIES_OPS - Executing fat_ent_blocknr
0.030598] FAT_ENTRIES_OPS - Executing fat_ent_bread
0.030633] FAT_ENTRIES_OPS - Executing fat16_ent_set_ptr
0.030669] FAT_ENTRIES_OPS - Executing fat16_ent_get
0.030705] FAT_ENTRIES_OPS - Executing fat_ent_blocknr
0.030740] FAT_ENTRIES_OPS - Executing fat_ent_bread
0.030776] FAT_ENTRIES_OPS - Executing fat16_ent_set_ptr
0.030811] FAT_ENTRIES_OPS - Executing fat16_ent_get
0.030847] FAT_ENTRIES_OPS - Executing fat16_ent_put
0.030883] FAT_ENTRIES_OPS - Executing fat_ent_blocknr
```

Θα ξεκινήσουμε τις επόμενες δοκιμές αρχικά χρησιμοποιώντας το αρχείο lklfuse.c. Στο τερματικό γράφουμε την εντολή “**./cptofs -p -i /tmp/disk -t vfat lklfuse.c /**”. Κατά την αντιγραφή του αρχείο που είναι περίπου 14KB παρατηρούμε αρχικά μια κλήση **generic_file_read_iter** και μετά δημιουργία inodes στα superblock operations. Ακολουθεί η εγγραφή των δεδομένων (**generic_file_write_iter**) και συνεχίζουμε με συναρτήσεις ενημέρωσης της FAT, όπως **fat_ent_blocknr** και **fat_ent_bread**. Οι λειτουργίες αυτές διαχειρίζονται την κατανομή των δεδομένων σε blocks και clusters. Παρατηρούμε 5 κλήσεις

```
0.030919] FAT_ENTRIES_OPS - Executing fat_ent_bread
0.030954] FAT_ENTRIES_OPS - Executing fat16_ent_set_ptr
0.030990] FAT_ENTRIES_OPS - Executing fat16_ent_get
0.031028] MEMORY_OPERATIONS - Executing fat_write_end
0.031067] FILE_OPS - Executing generic_file_write_iter
0.031104] MEMORY_OPERATIONS - Executing fat_write_begin
0.031142] MEMORY_OPERATIONS - Executing fat_write_end
0.031194] INODE_OPS - Executing now - fat_setattr
0.031194]
0.057931] FILE_OPS - Executing now - fat_file_release
0.057931]
1.037422] SUPERBLOCK_OPERATIONS - Executing fat_remount
1.037422]
1.037830] SUPERBLOCK_OPERATIONS - Executing fat_write_inode
1.037830]
1.037941] MEMORY_OPERATIONS - Executing fat_writepages
1.037994] SUPERBLOCK_OPERATIONS - Executing fat_write_inode
1.037994]
1.039344] reboot: Restarting system
```

των δύο αυτών συναρτήσεων, υποψιαζόμαστε ότι οι τέσσερεις θα είναι για τα δεδομένα και η πέμπτη για μεταδεδομένα. Ύστερα γίνεται η ενημέρωση των πληροφορίων του αρχείου με την fat_setattr και τέλος κλείνει το αρχείο με **fat_file_release**. Μετά από αυτό παρατηρούμε μερικές ακόμα ενέργειες που πιθανόν σχετίζονται με το καθάρισμα και τον συγχρονισμό του αρχείου καταγράφοντας τα υπόλοιπα δεδομένα που έμειναν στη μνήμη.

Τέλος θα δοκιμάσουμε την εντολή **cpfromfs** ώστε να αντιγράψουμε δεδομένα από το εικονικό σύστημα αρχείων στο τοπικό. Εκτελούμε την εντολή "**./cpfromfs -p -i /tmp/disk -t vfat /cptofs.c .**" και βλέπουμε τις παρακάτω κλήσεις στο τερματικό:

```
0.023579] FILE_OPS - Executing generic_file_read_iter
0.024104] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.024104]
0.024124] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.024124]
0.024128] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.024128]
0.027239] VFAT_DIR_OPS - Executing vfat_lookup
0.027260] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.027260]
0.027287] INODE_OPS - Executing now - fat_getattr
0.027287]
0.027612] FILE_OPS - Executing generic_file_read_iter
0.027632] MEMORY_OPERATIONS - Executing fat_readahead
0.027678] FAT_ENTRIES_OPS - Executing fat_ent_blocknr
0.027695] FAT_ENTRIES_OPS - Executing fat_ent_bread
0.027779] FAT_ENTRIES_OPS - Executing fat16_ent_set_ptr
0.027797] FAT_ENTRIES_OPS - Executing fat16_ent_get
0.027873] FILE_OPS - Executing generic_file_read_iter
0.027899] FILE_OPS - Executing generic_file_read_iter
0.027907] FILE_OPS - Executing generic_file_read_iter
0.027928] FILE_OPS - Executing generic_file_read_iter
0.028055] SUPERBLOCK_OPERATIONS - Executing fat_remount
0.028055]
0.061949] FILE_OPS - Executing now - fat_file_release
0.061949]
0.062390] reboot: Restarting system
```

Για αρχή παρατηρούμε την **generic_file_read_iter** να διαβάζει το αρχείο από το mounted σύστημα. Ύστερα με κλήσεις **fat_alloc_inode** δημιουργούνται νέα inodes για το αρχείο. Παράλληλα εκτελείται και η αναζήτηση με την **vfat_lookup**. Η **fat_getattr** διαβάζει τα μεταδεδομένα προτού ξεκινήσει η κανονική ανάγνωση με **generic_file_read_iter**. Αξίζει επίσης να σχολιάσουμε ότι έχουμε λιγότερες κλήσεις από FAT_ENTRIES_OPS καθώς σε αντίθεση με την cptofs δεν χρειάζεται τροποποίηση του FAT εφόσον υπάρχουν ήδη στο δίσκο, το μόνο που χρειάζεται είναι η μέθοδος να τα εντοπίσει μια φορά και να τα διαβάσει. Στην cptofs η διαδικασία χρειάζεται να κάνει επαναλαμβανόμενες προσπελάσεις του FAT για να βρει ελεύθερα clusters και να κάνει

ενημερώσεις. Συνεχίζοντας, βλέπουμε μερικά ακόμη reads, ένα remount, πιθανώς συσχετιζόμενο με το reboot, κλείσιμο του αρχείου με **fat_file_release** και επανεκκίνηση.

Μετά την εκτέλεση των cptofs και cpfromfs έγινε χρήση της εντολής "**mount -t vfat -o loop /tmp/disk /mnt**" (ώς root) για να γίνει mount ο εικονικός δίσκος tmp/disk στο /mnt. Αυτό μας δίνει πρόσβαση στο περιεχόμενο του FAT ώστε να μπορούμε να επιβεβαιώσουμε ότι το αρχείο lklfuse.c όντως αντιγράφηκε σωστά. Πηγαίνοντας στον φάκελο /mnt βλέπουμε πράγματι το αρχείο και το μέγεθος του να έχει αντιγραφεί.

```
myy601@myy601lab2:~/lkl-source/tools/lkl$ su root
Password:
root@myy601lab2:/home/myy601/lkl-source/tools/lkl# mount -t vfat -o loop /tmp/disk /mnt
mount: /mnt: /tmp/disk is already mounted.
root@myy601lab2:/home/myy601/lkl-source/tools/lkl# cd /mnt
root@myy601lab2:/mnt# ls -l
total 16
-rwxr-xr-x 1 root root 14192 Nov  7  2023 lklfuse.c
```

Συνεχίζοντας με τις δοκιμές, δοκιμάσαμε να εκτελέσουμε την cptofs για ένα πολύ μικρό text αρχείο (31 bytes). Παρόλο που το αρχείο είναι πολύ πιο μικρό από το παραπάνω παρατηρούμε ότι εκτελούνται ακόμα **FAT_ENTRIES_OPS**, αν και πολύ πιο λίγες, εφόσον ακόμη εξακολουθεί να χρειάζεται τουλάχιστον 1 cluster. Επιπλέον όπως αναμένεται βλέπουμε μια μόνο κλήση των **fat_write_begin / fat_write_end** σε αντίθεση με τις τέσσερις του παραπάνω παραδείγματος λόγω του μεγέθους.

```
0.013734] FILE_OPS - Executing generic_file_write_iter
0.013751] MEMORY_OPERATIONS - Executing fat_write_begin
0.013756] FAT_ENTRIES_OPS - Executing fat_ent_blocknr
0.013775] FAT_ENTRIES_OPS - Executing fat_ent_bread
0.013799] FAT_ENTRIES_OPS - Executing fat16_ent_set_ptr
0.013814] FAT_ENTRIES_OPS - Executing fat16_ent_get
0.013818] FAT_ENTRIES_OPS - Executing fat16_ent_next
0.013820] FAT_ENTRIES_OPS - Executing fat16_ent_get
0.013822] FAT_ENTRIES_OPS - Executing fat16_ent_next
0.013833] FAT_ENTRIES_OPS - Executing fat16_ent_get
0.013839] FAT_ENTRIES_OPS - Executing fat16_ent_put
0.013862] MEMORY_OPERATIONS - Executing fat_write_end
0.013896] INODE_OPS - Executing now - fat_setattr
0.013896]
0.043695] FILE_OPS - Executing now - fat_file_release
```

Ενδιαφέρον έχει αυτό που παρατηρήσαμε όταν εκτελέσαμε cpfromfs για το ίδιο αρχείο. Συγκεκριμένα δεν υπήρχαν καθόλου **FAT_ENTRIES_OPS**. Παρατηρούμε όμως μια κλήση της **fat_readahead**, που σημαίνει πως η ανάγνωση των δεδομένων από το αρχείο έγινε μέσω της cache λόγω του μικρού μεγέθους του και δεν χρειάστηκαν συναρτήσεις όπως **fat_ent_bread**. Παράλληλα σιγουρεύουμε ότι η αντιγραφή του αρχείου έγινε σωστά ελέγχοντας τον κατάλογο /mnt όπως πριν.

```
0.009967] FILE_OPS - Executing generic_file_read_iter
0.010337] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.010337]
0.010357] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.010357]
0.010386] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.010386]
0.010785] VFAT_DIR_OPS - Executing vfat_lookup
0.010804] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.010804]
0.010811] INODE_OPS - Executing now - fat_getattr
0.010811]
0.010875] FILE_OPS - Executing generic_file_read_iter
0.010964] MEMORY_OPERATIONS - Executing fat_readahead
0.011006] FILE_OPS - Executing generic_file_read_iter
0.011136] SUPERBLOCK_OPERATIONS - Executing fat_remount
0.011136]
0.025784] FILE_OPS - Executing now - fat_file_release
0.025784]
0.026085] reboot: Restarting system
```

```
myy601@myy601lab2:~/lkl-source/tools/lkl$ su root
Password:
root@myy601lab2:/home/myy601/lkl-source/tools/lkl# umount /mnt
root@myy601lab2:/home/myy601/lkl-source/tools/lkl# mount -t vfat -o loop /tmp/disk /mnt
root@myy601lab2:/home/myy601/lkl-source/tools/lkl# cd /mnt
root@myy601lab2:/mnt# ls -l
total 24
-rwxr-xr-x 1 root root 14192 Nov  7  2023 lklfuse.c
-rwxr-xr-x 1 root root    31 May 18 00:24 test1.txt
```

Θα εκτελέσουμε στη συνέχεια την αντιγραφή του μεγάλου binary αρχείου (3MB) με τις δύο εντολές. Με την κλήση της cptofs βλέπουμε ότι το output είναι πολύ μεγάλο για να διαβαστεί όλο από το τερματικό, συνεπώς το βάλαμε σε ένα αρχείο με όνομα output.txt. Αυτό που παρατηρούμε είναι αυξημένη εκτέλεση των ίδιων συναρτήσεων, όπως **write_begin**, **write_end**, **writepages** κλπ. Δεν παρατηρήσαμε κλήση κάποιας συνάρτησης που δεν είχε εμφανιστεί πριν με την χρήση του αρχείου lklfuse.c. Με βάση τα printk's που χρησιμοποιούμε αυτό δεν μας κάνει μεγάλη εντύπωση καθώς δεν υπάρχει κάποια συνάρτηση που περιμέναμε να κληθεί (δεδομένου ότι χρησιμοποιούμε πολύ μεγάλο αρχείο) που δεν έχουμε δει ήδη. Ίδιο φαινόμενο παρατηρείται και με την **cpfromfs**.

Τέλος θα επιχειρήσουμε να αντιγράψουμε έναν κατάλογο με περιεχόμενα. Εκτελώντας cptofs για έναν κατάλογο με περιεχόμενα δύο μικρά αρχεία παρατηρούμε ότι εκτός από τις κλήσεις που βλέπαμε μέχρι στιγμής, εμφανίζονται πλέον οι λειτουργίες **VFAT_DIR_OPS**, και πιο συγκεκριμένα η **vfat_mkdir** για τη δημιουργία καταλόγου, η **vfat_lookup** η οποία ψάχνει ένα αρχείο σε έναν φάκελο με βάση το όνομα του και η **vfat_create** η οποία το δημιουργεί.

```
0.015846] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.015846]
0.016244] VFAT_DIR_OPS - Executing vfat_lookup
0.016268] VFAT_DIR_OPS - Executing vfat_mkdir
0.016272] FAT_ENTRIES_OPS - Executing fat_ent_blocknr
0.016278] FAT_ENTRIES_OPS - Executing fat_ent_bread
```

```
0.052412] VFAT_DIR_OPS - Executing vfat_lookup
0.052466] VFAT_DIR_OPS - Executing vfat_create
0.052509] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.052509]
0.052630] FILE_OPS - Executing generic_file_write_iter
0.052683] MEMORY_OPERATIONS - Executing fat_write_begin
0.052726] FAT_ENTRIES_OPS - Executing fat_ent_blocknr
0.052764] FAT_ENTRIES_OPS - Executing fat_ent_bread
0.052802] FAT_ENTRIES_OPS - Executing fat16_ent_set_ptr
0.052810] FAT_ENTRIES_OPS - Executing fat16_ent_get
0.052832] FAT_ENTRIES_OPS - Executing fat16_ent_put
0.052867] MEMORY_OPERATIONS - Executing fat_write_end
0.052972] INODE_OPS - Executing now - fat_setattr
0.052972]
0.053000] VFAT_DIR_OPS - Executing vfat_lookup
0.053033] VFAT_DIR_OPS - Executing vfat_create
```

Λειτουργίες **vfat_lookup** εμφανίζονται παράλληλα και στην **cpfromfs**. Παρατηρούμε μόνο lookup και όχι create και mkdir διότι οι κλήσεις αυτές δεν πρέπει να εμφανίζονται στο FAT το οποίο στην cpfromfs απλώς διαβάζει. Οι εγγραφές γίνονται στο fs του debian και δεν είναι η δουλειά του FAT να τις εμφανίζει εφόσον δεν γίνονται σε αυτό.

```
0.017040] VFAT_DIR_OPS - Executing vfat_lookup
0.017071] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.017071]
0.017087] FAT_ENTRIES_OPS - Executing fat_ent_blocknr
0.017094] FAT_ENTRIES_OPS - Executing fat_ent_bread
0.017187] FAT_ENTRIES_OPS - Executing fat16_ent_set_ptr
0.017212] FAT_ENTRIES_OPS - Executing fat16_ent_get
0.017322] INODE_OPS - Executing now - fat_getattr
0.017322]
0.017376] VFAT_DIR_OPS - Executing vfat_lookup
0.017388] SUPERBLOCK_OPERATIONS - Executing fat_alloc_inode
0.017388]
0.017402] INODE_OPS - Executing now - fat_getattr
0.017402]
0.017478] FILE_OPS - Executing generic_file_read_iter
0.017557] MEMORY_OPERATIONS - Executing fat_readahead
0.017595] FILE_OPS - Executing generic_file_read_iter
0.017748] VFAT_DIR_OPS - Executing vfat_lookup
```

Μέσα από την εκτέλεση των παραδειγμάτων και την παρατήρηση των printk(), εντοπίσαμε ποιες συναρτήσεις καλούνται κατά την εγγραφή (cptofs), την ανάγνωση (cpfromfs) στο FAT

καθώς και κατά την εκκίνηση του συστήματος (boot). Αυτό μας βοηθά να κατανοήσουμε πού ακριβώς παρεμβαίνει το σύστημα αρχείων, ώστε να δημιουργήσουμε το journal καταγράφοντας ουσιαστικές ενέργειες σε κατάλληλα σημεία.

Καταγραφή Λειτουργιών VFAT σε Journal Αρχείο

Το αρχείο καταγραφής (Journal) έχει ως σκοπό να αποθηκεύει πληροφορίες ώστε σε περίπτωση που το σύστημα καταρρεύσει ή βρεθεί σε μη λειτουργική κατάσταση, να μπορούμε να ανακτήσουμε αυτές τις πληροφορίες και να επαναφέρουμε το σύστημα στην προηγούμενη σταθερή του κατάσταση. Όπως συζητήθηκε στις τελευταίες διαλέξεις, το σύστημα αρχείων FAT δεν διαθέτει εγγενή μηχανισμό journaling.

Στόχος μας είναι να ανοίξουμε ένα αρχείο στο οποίο θα καταγράφονται οι αλλαγές ή οι αρχικοποιήσεις μεταβλητών που σχετίζονται με πεδία βασικών δομών του συστήματος αρχείων όπως το **superblock**, τα **inodes**, οι εγγραφές καταλόγων (**directory entries**) και τα **αρχεία**. Η καταγραφή θα γίνεται κάθε φορά που ο κώδικας περνάει από ένα σημείο που μπορεί να επιφέρει τέτοιες αλλαγές.

Τρέχοντας τις εντολές cptofs και cpfromfs προηγουμένως, παρατηρούσαμε ότι γενικά τρέχανε πρώτα οι λειτουργίες του superblock, γι΄ αυτό το λόγο σκεφτήκαμε πως πρέπει να ψάξουμε εκεί για να βρούμε μια συνάρτηση που γίνεται όσο πιο κοντά γίνεται στην προσάρτηση του VFAT έτσι ώστε να ξεκινήσουμε εκεί το journal. Καταλήξαμε στη συνάρτηση **fat_fill_super()** (στο αρχείο **inode.c**) καθώς είναι αυτή που αρχικοποίει το superblock. Επιπλέον τα παρακάτω σχόλια του προγραμματιστή στη μέθοδο μας λένε ότι ο κώδικας στο σημείο αυτό είναι πριν γίνει προσάρτηση του file system:

```
1700        * GFP_KERNEL is ok here, because while we do hold the
1701        * superblock lock, memory pressure can't call back into
1702    →   * the filesystem, since we're only just about to mount
1703    →   * it and have no inodes etc active!
1704        */
```

Αρχικά θα πρέπει να γίνουν οι κατάλληλες δηλώσεις στο αρχείο **fat.h**. Ξεκινώντας θα βάλουμε τα πρότυπα των συναρτήσεων που θα δημιουργήσουμε για το άνοιγμα, το κλείσιμο, και την εγγραφή στο journal. Θα μπορούσαμε να χρησιμοποιήσουμε include όμως στο μάθημα μας προτάθηκε να μην χρησιμοποιήσουμε αυτή τη λύση. Αναζητήσαμε τα πρότυπα των **open()**, **close()**, **write()** και **fsync()** στην παρακάτω ιστοσελίδα και τα τοποθετήσαμε στο fat.h:

https://www.man7.org/linux/man-pages/man2

```
481   /* Added Code                          */
482   /* journal-related syscall prototypes */
483   int open(const char *pathname, int flags, ... /* mode_t mode */);
484   int close(int fd);
485   ssize_t write(int fd, const void *buf, size_t count);
486   int fsync(int fd);
```

Επιπλέον προσθέσαμε τον **file descriptor** σαν **extern** μεταβλητή έτσι ώστε να έχουν όλα τα αρχεία που κάνουν include το fat.h πρόσβαση σε αυτόν.

```
488   extern int file_descriptor;
```

Πίσω στο αρχείο **inode.c** και συγκεκριμένα στη μέθοδο **fat_fill_super** αρχικοποιούμε μερικά πεδία που θα χρησιμοποιήσει η συνάρτηση **open()**. Συγκεκριμένα γίνεται αρχικοποίηση του ονόματος του αρχείου το οποίο θα καταγράφεται στο journal, τα flags για το άνοιγμα του αρχείου (εγγραφή στο τέλος του αρχείου, άνοιγμα για ανάγνωση και εγγραφή και δημιουργία του αρχείου αν δεν υπάρχει). Δίνουμε επιπλέον και τα δικαιώματα του αρχείου (ανάγνωση και εγγραφή προς όλους τους χρήστες):

```
1693      /* Added code */
1694      char *path_for_journal = "journal.txt"; // Create the file in root directory of fs
1695      int flags = O_APPEND | O_RDWR | O_CREAT; // Flags
1696      mode_t mode = 0666; // Permissions
```

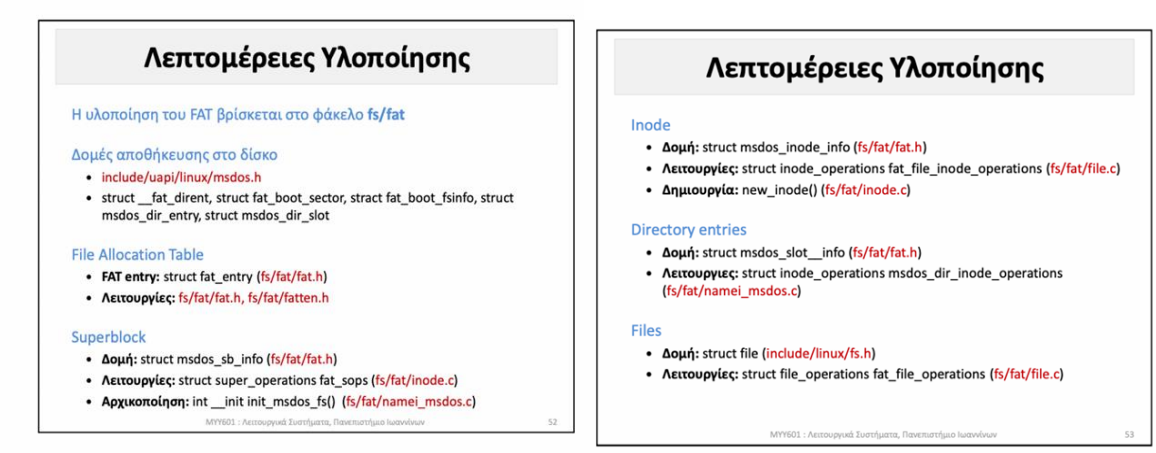
Λίγες γραμμές πιο κάτω γίνεται το άνοιγμα του journal με χρήση της **open()** και των παραμέτρων που ορίσαμε παραπάνω.

```
1710      /* Added code */
1711      /* Opening journal file here*/
1712      file_descriptor = open(path_for_journal, flags, mode);
```

Επιπλέον χρειάζεται να γίνει αρχικοποίηση του **file descriptor** και σε ένα από τα αρχεία στα οποία θα γίνουν καταγραφές. Π.χ. στην αρχή του **inode.c** τον ορίζουμε στην αρχή μαζί με τα πεδία:

```
40    /* Added code */
41    /* Global file descriptor for journaling */
42    int file_descriptor = -1;
```

Αφού δημιουργήσαμε το αρχείο journal, επόμενο βήμα είναι να εντοπίσουμε ποια **πεδία** από τις βασικές δομές του FAT driver είναι σημαντικά ώστε να καταγράφονται, με στόχο την παρακολούθηση ή το debugging της λειτουργίας του.



Κινηθήκαμε με βάση τις φροντιστηριακές διαφάνειες 53 και 54 για την περαιτέρω μελέτη των δομών.

Superblock

Αρχικά ψάχνουμε για την αρχικοποίηση του struct **msdos_sb_info** με "**bin/search ms_dos_sb_info**" στον κατάλογο **fs/fat** (διότι πλέον δουλεύουμε στο FAT). Η αναζήτηση επιστρέφει ότι το struct αρχικοποιείται ως struct msdos_sb_info ***sbi** = … Συνεπώς σε κάθε αρχείο του καταλόγου ψάξαμε τα σημεία που βρίσκεται "**sbi->**" το οποίο δείχνει αναφορά σε κάποιο πεδίο του καταλόγου. Για παράδειγμα βρήκαμε πάρα πολλές αναφορές στα πεδία του struct στη συνάρτηση **fat_fill_super()** του αρχείου inode.c η οποία συμπεράναμε ότι είναι υπεύθυνη για την αρχικοποίηση τους.

Inode

Ακολουθώντας παρόμοια διαδικασία ψάξαμε για αρχικοποίηση του struct **msdos_inode_info** με **bin/search** και βρήκαμε δύο αρχικοποιήσεις, μια με όνομα ***i** και μια με ***ei**. Όπως κάναμε πάνω ψάξαμε στα αρχεία για αναφορά στα πεδία τους αναζητώντας "**i->**" και "**ei->**". Η αναζήτηση έγινε μέσω VS Code καθώς υποστηρίζει την αναζήτηση σε πολλά αρχεία ταυτόχρονα κάνοντας τη διαδικασία πιο γρήγορη.

Directory Entries

Το struct **msdos_slot__info** δεν βρέθηκε ούτε στο αρχείο fat.h ούτε στον κατάλογο fs/fat. Στο αρχείο της διαφάνειας υπάρχει ένα struct με όνομα **fat_slot_info** και επιλέξαμε αυτό για να παρακολουθήσουμε την αλλαγή των πεδίων του. Ξανά κάνοντας **bin/search** βρήκαμε την

αρχικοποίηση του με όνομα ***sinfo**. Αντίστοιχα έγινε αναζήτηση για τις αναφορές στα πεδία τους.

Files

Η αναζήτηση του struct **file** στο **fs/fat** μάς έφερε πολλά αποτελέσματα καθώς η αναζήτηση μας έβρισκε και παρόμοια ονόματα όπως file_system_type κλπ που δεν σχετίζονταν με αυτό που θέλαμε. Για να προχωρήσουμε ψάξαμε στα αρχεία του καταλόγου για να βρούμε "struct file *" ώστε να βρούμε τη δήλωση που μας ενδιαφέρει. Τελικά βρήκαμε έτσι την αρχικοποίηση που γίνεται με όνομα **file** (struct file *file = ...).

Χρησιμοποιήσαμε τον **Gdb** για να βρούμε τα πεδία και το αν γίνονται αλλαγές σε αυτά στον κώδικα. Θα δείξουμε ένα παράδειγμα της διαδικασίας που ακολουθήσαμε με τη δομή του **superblock**. Αρχικά να αναφέρουμε ότι το struct που συμπεριλαμβάνει όλα τα πεδία του είναι το **msdos_sb_info** που βρίσκεται στο αρχείο **fat.h**. Η συνάρτηση στην οποία αρχικοποιείται το struct είναι η **fat_fill_super()** στο αρχείο **inode.c** (ορίζονται δύο πεδία και στην init_once() στο ίδιο αρχείο). Είναι η συνάρτηση στην οποία θα βάλουμε **breakpoints** για να εξετάσουμε τα πεδία. Στο τερματικό ξεκινάμε τον Gdb και ορίζουμε το breakpoint. Ύστερα κάνουμε **run** την cptofs για το test_directory.

```
myy601@myy601lab2:~/lkl-source/tools/lkl$ gdb ./cptofs
(gdb) break fat_fill_super
Breakpoint 1 at 0x161be0: file fs/fat/inode.c, line 1660.
(gdb) run -p -i /tmp/disk -t vfat test_directory /
```

Φτάνοντας στο breakpoint θα κάνουμε **next** ώστε να φτάσουμε στη γραμμή που γίνεται διαθέσιμο το struct, θα κάνουμε print για να πάρουμε τη διευθυνσή του και θα ορίσουμε ένα **watch-point** για μια μεταβλητή π.χ. **free_clusters**.

```
Thread 1 "cptofs" hit Breakpoint 1, fat_fill_super (sb=0x7fffe9fe2800, data=0x7fffe9fe9000, silent=0, isvfat=isvfat@entry=1,
    setup=setup@entry=0x5555556b7810 <setup>) at fs/fat/inode.c:1660
1660            struct fat_bios_param_block bpb;
(gdb) next
[Thread 0x7ffff75d26c0 (LWP 164351) exited]
[New Thread 0x7fffd97fa6c0 (LWP 164378)]
[Thread 0x7fffd97fa6c0 (LWP 164378) exited]
1682            sbi = kzalloc(sizeof(struct msdos_sb_info), GFP_KERNEL);
(gdb) next
553                     return kmalloc_trace(
(gdb) next
1682            sbi = kzalloc(sizeof(struct msdos_sb_info), GFP_KERNEL);
(gdb) next
1683            if (!sbi)
(gdb) print sbi
$1 = (struct msdos_sb_info *) 0x7fffe9dcc000
(gdb) watch ((struct msdos_sb_info *)0x7fffe9dcc000)->free_clusters
Hardware watchpoint 2: ((struct msdos_sb_info *)0x7fffe9dcc000)->free_clusters
(gdb) _
```

Με **continue** θα συνεχιστεί η εκτέλεση μέχρι να βρεθεί σημείο που αλλάζει η μεταβλητή. Όταν γίνει η αλλαγή ο Gdb μας δείχνει την παλιά και την καινούρια τιμή της.

```
(gdb) continue
Continuing.
[New Thread 0x7fffd97fa6c0 (LWP 164379)]
[Thread 0x7fffd97fa6c0 (LWP 164379) exited]

Thread 1 "cptofs" hit Hardware watchpoint 2: ((struct msdos_sb_info *)0x7fffe9dcc000)->free_clusters

Old value = 0
New value = 4294967295
fat_fill_super (sb=0x7fffe9fe2800, data=<optimized out>, silent=0, isvfat=<optimized out>, setup=<optimized out>) at fs/fat/inode.c:1767
1767            sb->s_maxbytes = 0xffffffff;
```

Για να εξετάσουμε για αλλαγές κάναμε παρόμοια διαδικασία πολλές φορές και για διαφορετικές δομές.

Καταγραφή πεδίων για το journal

Σημείωση : Μερικές αλλαγές μπορεί να αφορούν κάποια πεδία μιας δομής π.χ. ei->nr_caches (δομή inode), όμως η ονομασία στη συγκεκριμένη περίπτωση μας παραπέμπει ότι το πεδίο αφορά αλλαγές στη μνήμη παρόλο που ανήκει στο struct του inode. Η εκτύπωση στο journal περιλαμβάνει και άλλα παρόμοια σενάρια.

1. Superblock:

i. fatent.c: Συναρτήσεις:

- int fat_alloc_clusters(): Γραμμές 605-611, 618-624, 648-655
- int fat_free_clusters(): Γραμμές 729-735
- int fat_count_free_clusters(): Γραμμές 890-896

ii. inode.c: Συναρτήσεις:

- int fat_fill_super(): Γραμμές 1816-1825, 1833-1839, 1861-1867, 1875-1882, 1899-1905, 1939-1945, 1958-1964, 1996-2002, 2009-2015, 2021-2027, 2034-2040, 2065-2071, 2084-2090,

2. Memory:

i. inode.c: Συναρτήσεις:

- static void init_once(): Γραμμές 831-838

- int fat_fill_super(): Γραμμές 1974-1980

ii. cache.c: Συναρτήσεις:

- static void __fat_cache_inval_inode(): Γραμμές 206-212, 220-226, 232-238

3. Εγγραφές FAT:

i. fatent.c: Συναρτήσεις:

- void fat_ent_access_init(): Γραμμές 388-395

4. File:

i. dir.c: Συναρτήσεις:

- static int fat_ioctl_readdir(): Γραμμές 806-812

5. Inode:

i. inode.c: Συναρτήσεις:
- int fat_fill_super(): Γραμμές 2107-2113, 2123-2129

6. Directory:

i. inode.c: Συναρτήσεις:

- int fat_fill_super(): Γραμμές 1915-1924

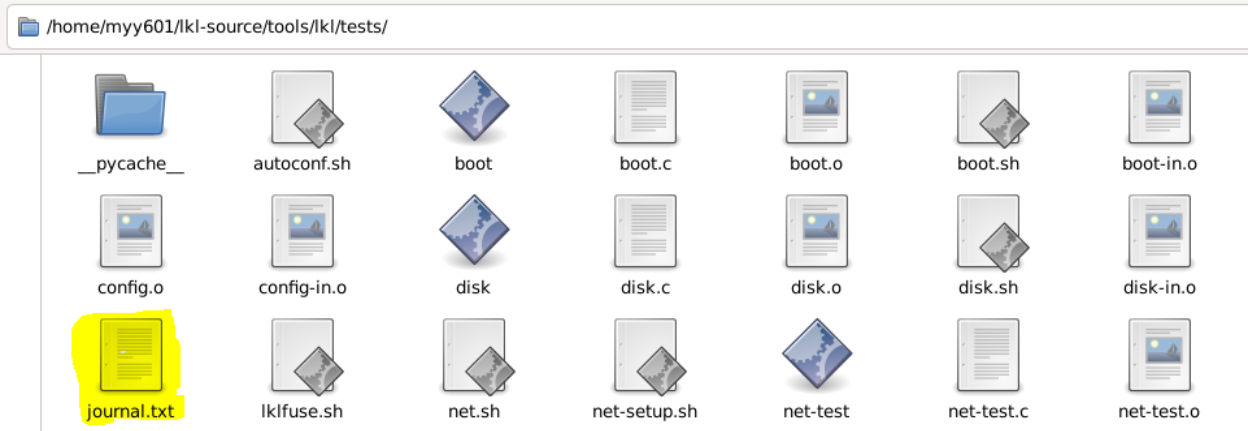
ii. dir.c: Συναρτήσεις:

- int fat_search_long(): Γραμμές 543-553
- int fat_scan(): Γραμμές 995-1002, 1012-1020
- int fat_scan_logstart(): Γραμμές 1043-1050, 1060-1068
- int fat_remove_entries(): Γραμμές 1130-1138
- int fat_add_entries(): Γραμμές 1390-1396, 1507-1516

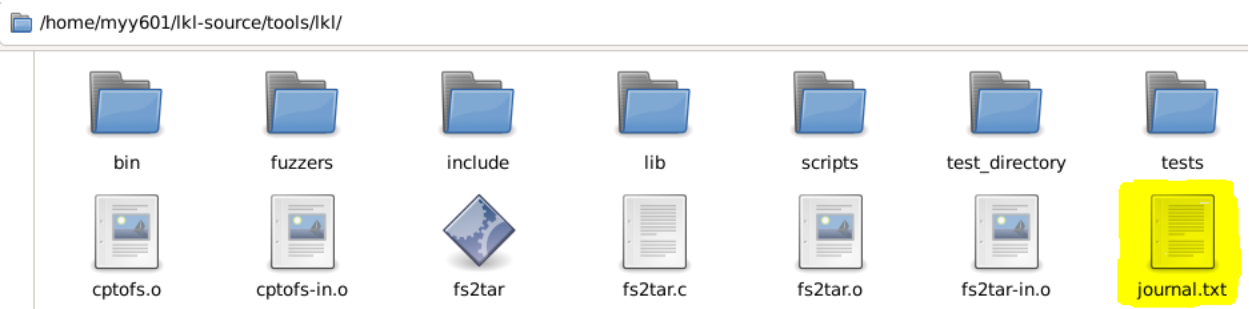
Τέλος αποφασίσαμε να βάλουμε τη συνάρτηση **close()** στη συνάρτηση **fat_put_super()** του αρχείου **inode.c** καθώς είναι υπεύθυνη για να απελευθερώσει πόρους του superblock και καλείται στο τέλος, συνήθως όταν γίνεται unmount του file system.

```
780     close(file_descriptor);
```

Με τις αλλαγές αυτές κάνουμε compile πρώτα με make –j8 clean και ύστερα με make –j8, εκτελούμε “./disk.sh -t vfat” στον φάκελο /lkl-source/tools/lkl/tests. Παρατηρούμε ότι με την εκτέλεση της εντολής δημιουργείται ένα αρχείο **journal.txt** στον φάκελο που είμαστε.



Πηγαίνοντας στο προηγούμενο directory με cd .. θα εκτελέσουμε την εντολή **boot** γράφοντας ” tests/boot -t vfat -d /tmp/vfatfile -p -P 0” στο terminal. Ύστερα εκτελούμε ”./cptofs -p -i /tmp/disk -t vfat lklfuse.c /” και παρατηρούμε τη δημιουργία ενός ακόμα αρχείου με όνομα journal.txt:



Θα τα ανοίξουμε δίπλα-δίπλα για να αναλύσουμε τα περιεχόμενα τους:

Αρχικά παρατηρούμε ότι το αρχείο από την εντολή **cptofs** έχει περισσότερες γραμμές καθώς έγιναν πιο πολλές αλλαγές και σε περισσότερα πεδία.

journal.txt

```
 69 nr_caches = 0
 70 cache_valid_id = 1
 71 init_once(), Memory changes:
 72 nr_caches = 0
 73 cache_valid_id = 1
 74 init_once(), Memory changes:
 75 nr_caches = 0
 76 cache_valid_id = 1
 77 init_once(), Memory changes:
 78 nr_caches = 0
 79 cache_valid_id = 1
 80 init_once(), Memory changes:
 81 nr_caches = 0
 82 cache_valid_id = 1
 83 init_once(), Memory changes:
 84 nr_caches = 0
 85 cache_valid_id = 1
 86 init_once(), Memory changes:
 87 nr_caches = 0
 88 cache_valid_id = 1
 89 init_once(), Memory changes:
 90 nr_caches = 0
 91 cache_valid_id = 1
 92 init_once(), Memory changes:
 93 nr_caches = 0
 94 cache_valid_id = 1
 95 init_once(), Memory changes:
 96 nr_caches = 0
 97 cache_valid_id = 1
 98 fat_fill_super(), Inode changes:
 99 fat_inode (count) = 1
100 fat_fill_super(), Inode changes:
101 fsinfo_inode = 2
102
```

journal.txt

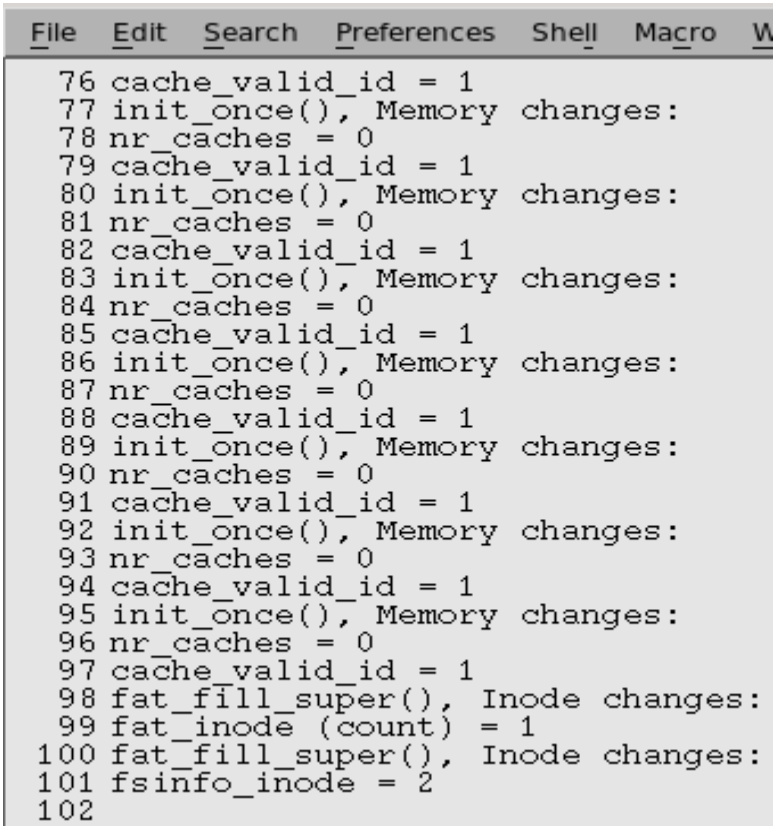
```
 87 nr_caches = 0
 88 cache_valid_id = 1
 89 init_once(), Memory changes:
 90 nr_caches = 0
 91 cache_valid_id = 1
 92 init_once(), Memory changes:
 93 nr_caches = 0
 94 cache_valid_id = 1
 95 init_once(), Memory changes:
 96 nr_caches = 0
 97 cache_valid_id = 1
 98 fat_fill_super(), Inode changes:
 99 fat_inode (count) = 1
100 fat_fill_super(), Inode changes:
101 fsinfo_inode = 2
102 fat_scan(), Directory changes:
103 slot_off = 0
104 bh = 0000000000000000
105 fat_add_entries, Directory changes:
106 nr_slots = 2
107 fat_add_entries, Directory changes:
108 slot_off = 0
109 de = LKLFUSE C
110 bh = 27
111 i_pos = 5377
112 fat_alloc_clusters(), Superblock changes:
113 prev_free = 3
114 fat_alloc_clusters(), Superblock changes:
115 free_clusters = -1
116 fat_alloc_clusters(), Superblock changes:
117 prev_free = 4
118 fat_alloc_clusters(), Superblock changes:
119 free_clusters = -1
120
```

Στην αρχή και των δύο γίνεται αρχικοποίηση του **Superblock**, του FAT, των μεταβλητών **free_clusters**, **vol_id** κλπ. Όπως παρατηρούμε από τις παραπάνω εικόνες, το journal που δημιουργήθηκε με την cptofs έχει καταγράψει πληροφορίες όπως αλλαγές στη συνάρτηση **fat_alloc_clusters()** και συγκεκριμένα αλλαγή των τιμών **prev_free** και **free_clusters** επειδή γίνεται allocation πραγματικών clusters για την αποθήκευση του αρχείου, καθώς και αλλαγές σε πεδία της **fat_search_long()** όπως **nr_slots**, **de**, **bh**, **i_pos**. Αυτό δείχνει ότι έγινε αναζήτηση για το όνομα lklfuse.c και δημιουργήθηκαν 2 slots γι΄ αυτό. Ακόμη η αλλαγή στο πεδίο **fat_inode (count)** μάς δείχνει ότι έγινε προσθήκη ενός καινούριου αρχείου.

Στη συνέχεια εκτελέσαμε **cptofs** για την αντιγραφή του καταλόγου **test_directory** ο οποίος περιέχει δύο μικρά αρχεία. Το journal που δημιουργείται σε αυτήν την περίπτωση είναι ακόμα μεγαλύτερο και αναλύουμε μερικές διαφορές του (συγκριτικά με αυτό που δημιουργήθηκε με την αντιγραφή μόνο ενός αρχείου) παρακάτω:

```
File  Edit  Search  Preferences  Shell  Macro  Windows
101 fsinfo_inode = 2
102 fat_alloc_clusters(), Superblock changes:
103 prev_free = 5
104 fat_alloc_clusters(), Superblock changes:
105 free_clusters = -1
106 fat_scan(), Directory changes:
107 slot_off = 0
108 bh = 0000000000000000
109 fat_add_entries, Directory changes:
110 nr_slots = 3
111 fat_add_entries, Directory changes:
112 slot_off = 64
113 de = TEST_D~1    <dle>
114 bh = 27
115 i_pos = 5380
116 fat_scan(), Directory changes:
117 slot_off = 0
118 bh = 0000000000000000
119 fat_add_entries, Directory changes:
120 nr_slots = 2
121 fat_add_entries, Directory changes:
122 slot_off = 64
123 de = TEST2    C
124 bh = 19
125 i_pos = 6659
126 fat_alloc_clusters(), Superblock changes:
127 prev_free = 6
128 fat_alloc_clusters(), Superblock changes:
129 free_clusters = -1
130 fat_scan(), Directory changes:
131 slot_off = 0
132 bh = 0000000000000000
133 fat_add_entries, Directory changes:
134 nr_slots = 2
135 fat_add_entries, Directory changes:
136 slot_off = 128
137 de = TEST1    TXT
138 bh = 19
139 i_pos = 6661
140 fat_alloc_clusters(), Superblock changes:
141 prev_free = 7
142 fat_alloc_clusters(), Superblock changes:
143 free_clusters = -1
144
```

Σε αυτό το αρχείο όπως αναμένεται έχουμε τρεις ομάδες εισαγωγών (**fat_add_entries**), μια για το directory και άλλες δύο για τα αρχεία που περιέχει. Παρατηρούμε επιπλέον κλήσεις **fat_alloc_clusters** εφόσον έχουμε περισσότερες εγγραφές. Βλέπουμε ακόμα το πεδίο **de** να παίρνει τιμές από τα ονόματα των αρχείων (TEST1 C, TEST2 TXT) και του directory (TEST_D-1), καθώς το πεδίο **bh** παίρνει σε δύο διαφορετικές εγγραφές την ίδια τιμή (19) υποδεικνύοντας ότι το ίδιο block χρησιμοποιείται για δύο αρχεία (ανήκουν στο ίδιο directory block).



Για να γίνουν όλες οι δοκιμές των λειτουργιών εκτελέσαμε και την εντολή **cpfromfs** με το **test_directory**. Το αρχείο journal που παίρνουμε έχει λιγότερες γραμμές και σε αντίθεση με το journal για την cpfromfs δεν γίνεται καμία κλήση της **fat_alloc_clusters()** και της **fat_add_entries()** καθώς δεν δημιουργούνται νέα entries, εκτελείται μόνο ανάγνωση. Μεταβλητές όπως **nr_slots**, **bh**, **de** δεν αλλάζουν με αποτέλεσμα λιγότερες γραμμές στο journal. Δεν υπάρχουν directory changes παρά μόνο για αρχικοποίηση στην αρχή.

Συμπεράσματα από την άσκηση:

Η άσκηση μάς έδωσε μια ξεκάθαρη εικόνα του πόσο απαιτητική είναι η δουλειά σε χαμηλό επίπεδο, τόσο σε όγκο κώδικα όσο και σε πολυπλοκότητα. Ήταν μια χρήσιμη εμπειρία που μας έδειξε τι σημαίνει να μπλέκεις με τον πυρήνα και πόση προσοχή χρειάζεται όταν δουλεύεις τόσο κοντά στο σύστημα.