

Υλοποίηση αρχείου καταγραφής στο σύστημα αρχείων FAT του Linux

Θεόδωρος Γκόλτσιος

AM 1991

Εργασία Εξαμήνου 2 Λειτουργικών Συστημάτων

Καθηγητής: Στέργιος Αναστασιάδης

Ιωάννινα, Μάιος, 2023



**ΤΜΗΜΑ ΜΗΧ. Η/Υ & ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ**

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
UNIVERSITY OF IOANNINA**

Περίληψη

Στο παρόν project, καλούμαστε να υλοποιήσουμε ένα αρχείο καταγραφής (journal) για το FAT σύστημα αρχείων, το οποίο κρατάει τις αλλαγές στις βασικές δομές του συστήματος

αρχείων , όταν καλούμε κάποιες εφαρμογές της lkl (πχ cprotofs),

Το σύστημα αρχείων αυτό βρίσκεται μέσα στον πυρήνα της lkl. Θα εξηγήσουμε αναλυτικότερα γιατί είναι χρήσιμη η lkl, θα περιγράψουμε την εφαρμογή cprotofs και τέλος θα αναλύσουμε ποια σημαντικά πεδία δομών αποφασίσαμε να καταγράψουμε στο αρχείο καταγραφής, εξηγώντας τις λειτουργίες τους.

Λέξεις Κλειδιά: journal , Linux Kernel Library , FAT file system

Πίνακας Περιεχομένων

Κεφάλαιο 1. Εισαγωγικά: Linux Kernel Library	1
1.1 Περιγραφή της LKL.....	Error! Bookmark not defined.
1.1.1 Τί είναι η LKL και γιατί είναι χρήσιμη	1
 Κεφάλαιο 2. Δημιουργία Journal για το FAT fs.....	7
2.1 Υλοποίηση του FAT fs και βασικές δομές	Error! Bookmark not defined.
2.2 Σε τί μας χρησιμεύει το journal	Error! Bookmark not defined.
2.3 Journal entries για τις βασικές δομές του FAT.....	Error! Bookmark not defined.

Κεφάλαιο 1. Εισαγωγικά: Linux

Kernel Library

1.1 Περιγραφή της LKL

1.1.1 Τί είναι η LKL και γιατί είναι χρήσιμη

Η Linux Kernel Library (LKL) είναι library του linux kernel (παρέχοντας και εφαρμογές και API μαζί) , με την οποία μπορούμε να τροποποιήσουμε κάποιες κλήσεις συστήματος και να μελετήσουμε-βελτιώσουμε διάφορους drivers(όπως το FAT στο project μας) από Το user space. Η LKL μας προσφέρει και τα παρακάτω:

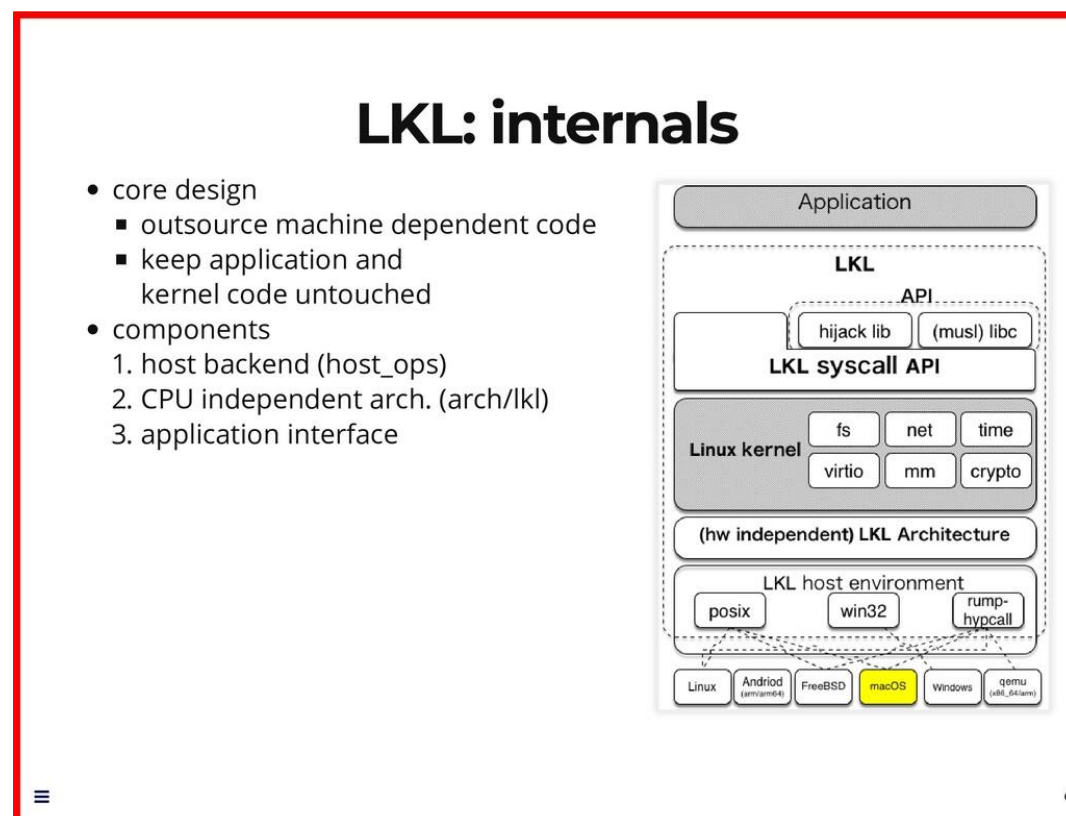
1. Αρκετά βελτιωμένη απόδοση: Η LKL παρέχει δυνατότητα στις εφαρμογές να κάνουν sys_call κατευθείαν από user-space, χωρίς όμως την συνθετότητα και κατανάλωση πόρων του συστήματος που είναι παρελκόμενα με τη χρήση του πυρήνα. Αυτό έχει ως συνέπεια τις βελτιωμένες επιδόσεις και να μειώνεται ο χρόνος απόκρισης των εφαρμογών.
2. Μεγάλη Προσαρμοστικότητα: Η LKL μπορείς να κάνει intercept τις κλήσεις συστήματος (μέσω της βιβλιοθήκης hijack.lib, που προσφέρεται από το API της) και την αλλαγή τους ανάλογα με ανάγκες της εκάστοτε εφαρμογής. Έτσι μπορούμε ουσιαστικά να αλλάξουμε τις κλήσεις συστήματος για να επιτύχουμε μέγιστη απόδοση.
3. Ανεξαρτησία από τον πυρήνα: Η LKL επιτρέπει στις εφαρμογές να λειτουργούν ανεξάρτητα από τις αλλαγές στον πυρήνα του λειτουργικού συστήματος. Αυτό σημαίνει ότι οι εφαρμογές μπορούν να παραμείνουν συμβατές ακόμα και όταν

υπάρχουν αλλαγές στον πυρήνα, χωρίς την ανάγκη για επαναδιαμόρφωση ή αναβάθμιση.

4. Ασφάλεια: Η LKL παρέχει μηχανισμούς ασφαλείας που επιτρέπουν τον έλεγχο και την προστασία των κλήσεων συστήματος από κακόβουλες επεμβάσεις ή ευπάθειες.

Συνολικά, η Linux Kernel Library είναι ένα πολύ χρήσιμο εργαλείο που επιτρέπει στις εφαρμογές να αξιοποιούν τις λειτουργίες του πυρήνα του λειτουργικού συστήματος με αποδοτικό και ευέλικτο τρόπο. Η χρήση της LKL μπορεί να οδηγήσει σε βελτιωμένες επιδόσεις, μείωση της πολυπλοκότητας και αύξηση της ασφάλειας για τις εφαρμογές που την υιοθετούν.

Μπορούμε παρακάτω να δούμε σε αυτήν την εικόνα, σχηματικά την LKL και τα API της, και πώς γίνεται η οργάνωση στο host λειτουργικό σύστημα.



Εικόνα 1. LKL internals

Θα δώσουμε παρακάτω ένα παράδειγμα, διαφορετικό του project μας, για να δείξουμε άλλη μία πρακτική εφαρμογή της LKL, για κάποια άλλη χρήση.

Τυπικό-σύγχρονο παράδειγμα διαφορετικής χρήσης LKL

Στο παρακάτω τυπικό παράδειγμα έχουμε ένα node.js stack, και χρησιμοποιούμε την LKL για να βελτιώσουμε την απόδοσή του.

Τόσο η βιομηχανία όσο και ο ακαδημαϊκός χώρος υιοθετούν την υπολογιστική περιβάλλοντος του Node.js* με εκθετικό ρυθμό. Αυτό συμβαίνει εν μέρει επειδή το Node.js είναι τόσο ελαφρύ όσο και πολύ αποδοτικό στη διαχείριση μαζικών αριθμών ταυτόχρονων συνδέσεων. Το Node.js είναι επίσης υψηλής κλιμακωσιμότητας, κάτι που το καθιστά απίστευτα χρήσιμο για δικτυακές εφαρμογές. Ωστόσο, ειδικά για εφαρμογές Node.js στην πλευρά του διακομιστή, η δικτυακή επικοινωνία μπορεί να αποτελέσει ένα κρίσιμο φραγμό.

Το Node.js χρησιμοποιεί ένα ασύγχρονο μοντέλο I/O για τη μετάφραση των μηνυμάτων TCP/IP από τον χώρο του πυρήνα στον χώρο του χρήστη. Για τη δικτυακή επικοινωνία, οι εφαρμογές Node.js λαμβάνουν αιτήσεις που αποστέλλονται στο Διαδίκτυο σε μορφή μηνυμάτων. Όσο πιο γρήγορα λαμβάνονται και επεξεργάζονται αυτά τα μηνύματα από το Διαδίκτυο, τόσο πιο γρήγορα μπορεί να ανταποκριθεί μια εφαρμογή εξυπηρέτησης. Να θυμάστε ότι στο Node.js, η επεξεργασία I/O χρησιμοποιεί την επεξεργασία του TCP/IP stack στον πυρήνα. Σύμφωνα με το Ίδρυμα Node.js, σχεδόν καμία λειτουργία στο Node.js δεν εκτελεί απευθείας I/O. Λόγω αυτού, μπορείτε συνήθως να ελαχιστοποιήσετε τα φραγμούς επικοινωνίας βελτιστοποιώντας την εκτέλεση του Node.js για αύξηση του ρυθμού I/O.

Έτσι ένας τρόπος που μπορούμε να βελτιστοποιήσουμε αυτό το συγκεκριμένο stack του Node.js, είναι να αυξήσουμε τον ρυθμό I/O, χρησιμοποιώντας την LKL.

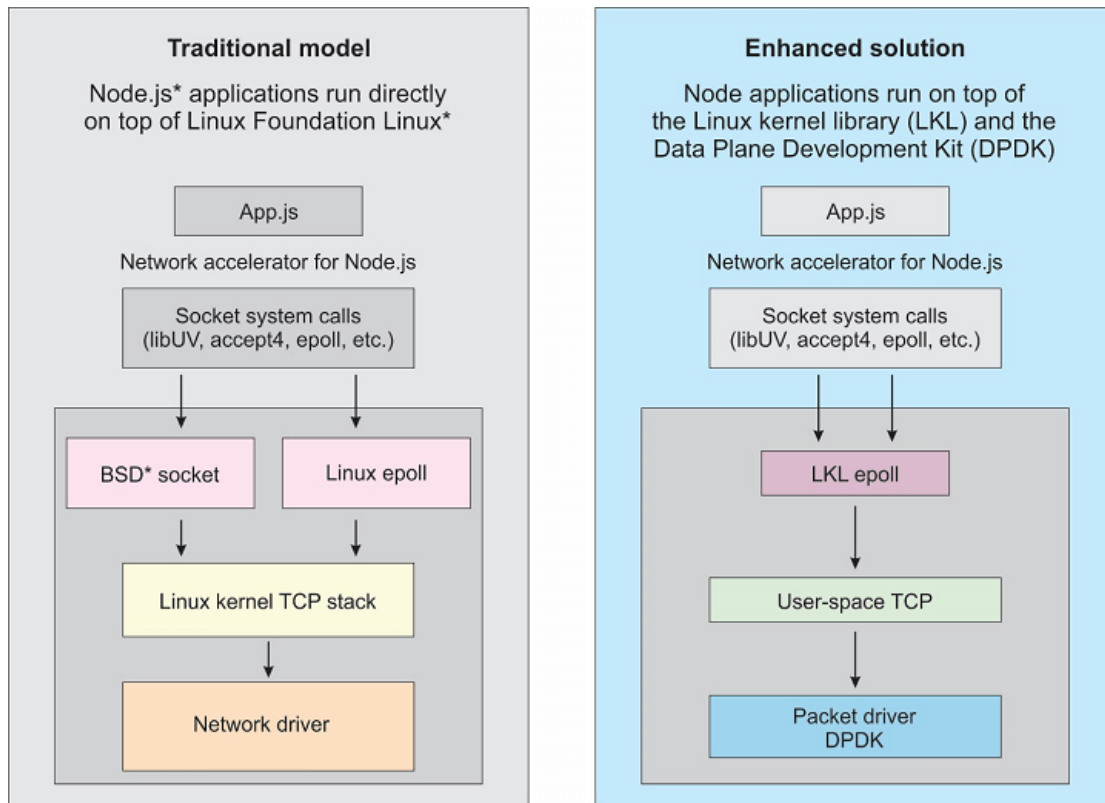
Η LKL είναι ένα fork του πυρήνα του Linux που μας επιτρέπει να εκτελέσετε λειτουργίες από τον πυρήνα στον χώρο του χρήστη. Η λύση του παραδείγματος(ακολουθεί εικόνα παρακάτω) χρησιμοποιεί ένα υποσύνολο της LKL, τη βιβλιοθήκη hijack. Αυτή η βιβλιοθήκη ανακατευθύνει τις κλήσεις συστήματος μιας εκτελούμενης εφαρμογής. Χρησιμοποιώντας αυτόν τον τύπο παρέμβασης, χρησιμοποιείτε μηχανισμούς LD_PRELOAD(μεταβλητή περιβάλλοντος στο Linux που επιτρέπει την προ-φόρτωση ενός καθορισμένου shared object πριν την εκτέλεση μιας εφαρμογής. Αυτό το shared object μπορεί να παρεμβάλλει και να τροποποιήσει συναρτήσεις που καλούνται από την εφαρμογή) για να ανακατευθύνετε τα μηνύματα προς τη βιβλιοθήκη hijack, η οποία

είναι στην πραγματικότητα ένα κοινόχρηστο αντικείμενο. Η βιβλιοθήκη hijack του LKL σας επιτρέπει να μεταφέρετε την επεξεργασία των μηνυμάτων TCP/IP από τον χώρο του πυρήνα στον χώρο του χρήστη. Ξέρουμε ότι κατά τη διάρκεια εκτέλεσης ενός διεργασίας, παράγονται κλήσεις συστήματος για τη δικτυακή επικοινωνία. Έτσι, χρησιμοποιώντας πχ μια εφαρμογή, που τρέχει πάνω από το Node.js, και το Node.js χρησιμοποιεί τη βιβλιοθήκη libUV (βιβλιοθήκη η οποία χρησιμοποιείται στο παρόν παράδειγμα για διαχείριση επικοινωνιών I/O δικτύου). Η βιβλιοθήκη libUV δημιουργεί τις κλήσεις συστήματος. Η βιβλιοθήκη LKL hijack παρεμβαίνει σε αυτές τις κλήσεις. Από αυτό το σημείο, η επεξεργασία TCP/IP περνά μέσω του χώρου χρήστη (user space).

Έτσι χρησιμοποιώντας αυτή τη λύση, βελτιώνουμε σημαντικά την απόδοση του stack μας.

Επιπλέον υπάρχουν και άλλα πλεονεκτήματα στη χρήση της LKL. Πρώτον, ο κώδικας της LKL είναι πολύ καθαρός. Δεύτερον, ο κώδικας αυτός ακολουθεί στενά τις οδηγίες προγραμματισμού του πυρήνα Linux. Φυσικά, αυτά τα πλεονεκτήματα βοηθούν στη διατήρηση του κώδικα της εφαρμογής σας ως αναγνώσιμος και πιο εύκολος στη συντήρηση.

Επίσης πρέπει να αναφέρουμε ότι στη συγκεκριμένη λύση, μαζί με την LKL, χρησιμοποιείται το DPDK που είναι ένα σύνολο βιβλιοθηκών και drivers για γρήγορη επεξεργασία πακέτων. Συγκεκριμένα, το DPDK ασχολείται με την επεξεργασία πακέτων στον χώρο του χρήστη, χρησιμοποιώντας τη CPU. Η Χρήση της δεν είναι απαραίτητη, αλλά όταν είμαστε στο σημείο που έχουμε κάνει intercept τα system calls μέσω της της LKL, μπορούμε να χρησιμοποιήσουμε και τις συναρτήσεις της DPDK για να κάνουμε πιο γρήγορη την παραπάνω διαδικασία (επεξεργασία μηνυμάτων TCP/IP στο user space).



Εικόνα 2. Enhanced solution of Node.js stack using LKL

Κεφάλαιο 2. Δημιουργία Journal για το FAT fs

2.1 Υλοποίηση του FAT fs και βασικές δομές

Στο παρών project, μας δίνεται κώδικας της LKL, ο οποίος περιέχει κώδικα του πυρήνα, διάφορες εφαρμογές-tools(πχ cptofs) και API χρησιμοποιώντας το πρόθεμα lkl_ πριν από τις διάφορες κλήσεις(πχ lkl_sys_mount κλπ).

Μέσα στο μέρος του πυρήνα της LKL, υπάρχουν διάφορες υλοποιήσεις πολλών συστημάτων αρχείων. Οι υλοποιήσεις αυτές στην ουσία αποτελούν drivers συστημάτων αρχείων, δηλαδή μας χρησιμεύουν έτσι όταν συνδεθεί(γίνεται mount) μία συσκευή(πχ ένας εξωτερικός δίσκος) στο λειτουργικό μας, μέσω αυτών των drivers να μπορέσουμε να επικοινωνήσουμε με την συσκευή(ανταλλαγή αρχείων κλπ). Ο πυρήνας της LKL επίσης μας προσφέρει το VFS layer (virtual file system), το οποίο μέσω κάποιων λειτουργιών και δομών, κάνει abstraction σε όλες τις υλοποιήσεις-drivers των συστημάτων αρχείων του διευκολύνοντας έτσι την επικοινωνία μεταξύ των συσκευών.

Έτσι γενικά με αυτό το abstraction, διάφορες εφαρμογές μπορούν να χρησιμοποιούν τις ίδιες συναρτήσεις και δομές δεδομένων για να αναπαραστήσουν και να διαχειριστούν αρχεία ανεξάρτητα από το συγκεκριμένο σύστημα αρχείων που χρησιμοποιείται στο σύστημα. Αυτό επιτρέπει την ευελιξία και την ανταλλαξιμότητα των συστημάτων αρχείων, καθώς μπορεί να γίνει εύκολη αλλαγή του συστήματος αρχείων χωρίς να χρειάζεται να τροποποιηθούν οι εφαρμογές.

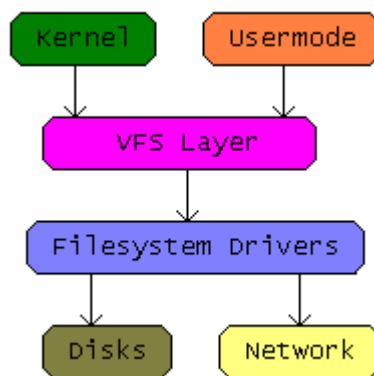
Στο παρών project, μας ενδιαφέρει η υλοποίηση-driver του FAT συστήματος αρχείων.

Θα σχολιάσουμε στη συνέχεια αναλυτικά για καλύτερη κατανόηση:

1.Τη δομή του συστήματος αρχείων FAT

2.Τη δομή του VFS του Linux

3.Τη δομή της υλοποίησης-driver του FAT στο Linux



Εικόνα 3. Linux File System Stack

Μερικές γενικές πληροφορίες για το FAT: Το FAT είναι ένα από τα παλαιότερα συστήματα αρχείων που χρησιμοποιούνται ακόμα και σήμερα. Το FAT χρησιμοποιήθηκε αρχικά στο λειτουργικό σύστημα MS-DOS της Microsoft και αργότερα στα Windows. Υπάρχουν τρεις κύριες εκδοχές του FAT:

1.FAT12: Χρησιμοποιήθηκε στην αρχική έκδοση του MS-DOS. Καθώς το όνομα υπονοεί, χρησιμοποιεί 12-bit καταχωρίσεις στον πίνακα αρχείων.

2.FAT16: Εισήχθη με το MS-DOS 3.0 το 1984. Το FAT16 επέτρεπε μεγαλύτερο μέγεθος δίσκου και μεγαλύτερα αρχεία σε σχέση με το FAT12.

3.FAT32: Εισήχθη με το Windows 95 OSR2 το 1996. Το FAT32 υποστηρίζει ακόμα μεγαλύτερα μεγέθη δίσκου και αρχείων σε σχέση με το FAT16, και περιλαμβάνει επίσης διάφορες βελτιώσεις στην απόδοση και την αξιοπιστία.

Σημειώστε ότι, παρόλο που το FAT είναι απλό και συμβατό με πολλά συστήματα, δεν υποστηρίζει ορισμένα σύγχρονα χαρακτηριστικά όπως τα μεγάλα αρχεία (>4GB για FAT32), τα δικαιώματα αρχείων και την κρυπτογράφηση.

Όμως, ένα από τα σημαντικά χαρακτηριστικά που λείπουν από το FAT συστήματα αρχείων είναι η λειτουργία του journaling. Θα σχολιάσουμε περισσότερα για αυτό παρακάτω.

1.Η δομή του συστήματος αρχείων FAT

Το FAT (File Allocation Table) είναι ένα αρκετά απλό σύστημα αρχείων, αλλά έχει μερικές **κύριες δομές** που είναι ζωτικής σημασίας για τη λειτουργία του:

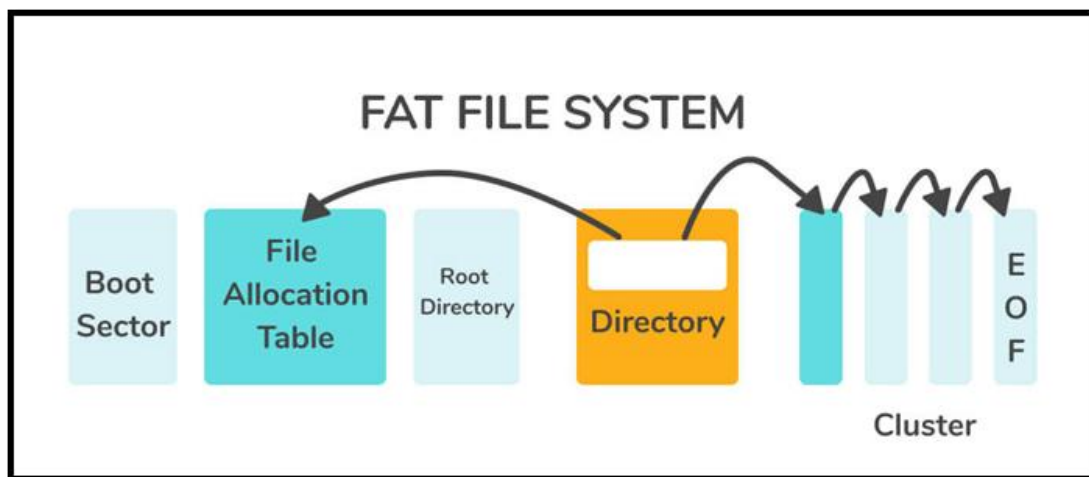
1.Boot Sector (ή Boot Area): Αυτή είναι η πρώτη δομή που αναγνωρίζει το σύστημα κατά την εκκίνηση. Περιέχει βασικές πληροφορίες σχετικά με το σύστημα αρχείων, όπως το μέγεθος των cluster, τον αριθμό των FAT και τη θέση του root directory.

2.File Allocation Tables (FATs): Αυτές είναι οι δομές που δίνουν το όνομα στο σύστημα αρχείων. Υπάρχουν τουλάχιστον δύο αντίγραφα του FAT σε κάθε δίσκο, για λόγους αξιοπιστίας. Κάθε FAT είναι ένας πίνακας που περιέχει την πληροφορία σχετικά με την κατάσταση όλων των clusters στον δίσκο. Root Directory: Αυτός είναι ο κατάλογος στον οποίο αναφέρονται όλα τα άλλα αρχεία και καταλόγοι. Περιέχει εγγραφές για κάθε αρχείο και κατάλογο στον δίσκο, με πληροφορίες όπως το όνομα, το μέγεθος, την ημερομηνία τελευταίας τροποποίησης και η αρχική θέση του στον FAT.

3. Root Directory Table: Αυτός είναι ο κατάλογος στον οποίο **αναφέρονται** όλα τα άλλα αρχεία και καταλόγοι. Περιέχει εγγραφές για κάθε αρχείο και κατάλογο στον δίσκο, με πληροφορίες όπως το όνομα, το μέγεθος, την ημερομηνία τελευταίας τροποποίησης και η αρχική θέση του στον FAT. Δείχνουν τα clusters στην data region που βρίσκονται τα files ή τα directories και μέσα στα File Allocation Tables δείχνονται σε ποια clusters βρίσκονται τα υπόλοιπα δεδομένα κάποιου αρχείου, ανάλογα πόσο μεγάλο είναι(αν είναι αρκετά μεγάλο, θα έχει αρκετές εγγραφές μέσα στο Fat Allocation Table).

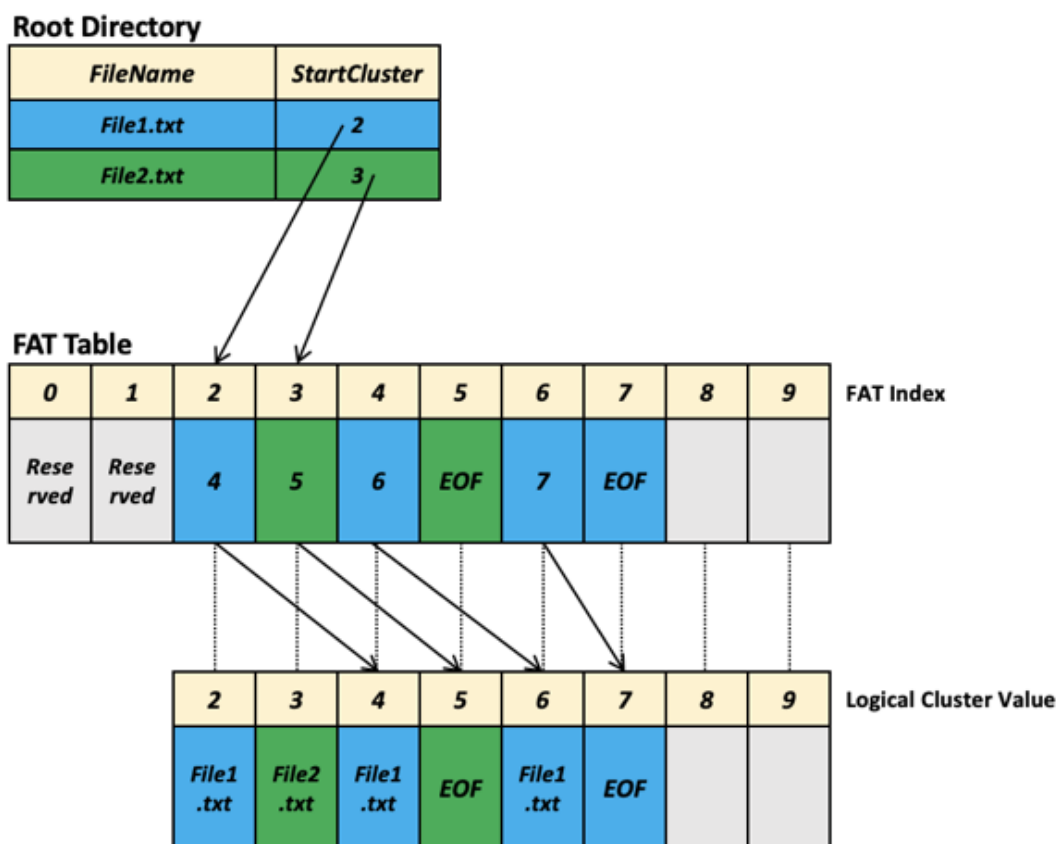
4.Data Region: Αυτή είναι η περιοχή του δίσκου που περιέχει τα πραγματικά δεδομένα των αρχείων και των καταλόγων. Είναι οργανωμένη σε clusters, τα οποία είναι τα βασικά μπλοκ του συστήματος αρχείων. Κάθε αρχείο ή κατάλογος είναι μια αλυσίδα από clusters, η οποία διασκορπισμένη σε όλο τον δίσκο. Η θέση του κάθε cluster στην αλυσίδα ελέγχεται από τις εγγραφές στον FAT.

Ακολουθούν εικόνες, για περαιτέρω κατανόηση:



Εικόνα 4. The FAT file system

Και παρακάτω ακολουθεί εικόνα που επισημαίνει τη συσχέτιση του **root directory** με το **data region** και τους πίνακες **Fat Allocation Tables**:



Εικόνα 5. FAT table, Root Directory and Data Region interaction

2.Η δομή του VFS του Linux

Το **VFS (Virtual File system)** είναι ένας μηχανισμός που κάνοντας abstraction επιτρέπει στο λειτουργικό σύστημα του Linux να διαβάσει διάφορα συστήματα αρχείων και να παρέχει ένα ενιαίο **interface** για τη διαχείριση αρχείων.

ΠΡΟΣΟΧΗ: Το VFS είναι μηχανισμός συγκεκριμένα του Linux(και σε διάφορα άλλα Unix-like), όπως έχουμε και για την εργασία μας. Παρόμοιοι μηχανισμοί, αλλά όχι ίδιοι, είναι υλοποιημένοι σε διαφορετικά λειτουργικά συστήματα, όπως πχ το **IFS (Installable File System)**, για το Windows os.

Οι βασικές δομές δεδομένων που χρησιμοποιούνται στο VFS περιλαμβάνουν:

1. **Superblock:** Αυτή η δομή περιέχει πληροφορίες για ένα σύστημα αρχείων. Κάθε σύστημα αρχείων έχει το δικό του superblock, που δημιουργείται κατά τη διάρκεια της διαμόρφωσης(format) του συστήματος αρχείων.
2. **Inode (Index Node):** Αυτή η δομή περιέχει πληροφορίες για ένα συγκεκριμένο αρχείο ή κατάλογο. Περιλαμβάνει πληροφορίες όπως τα δικαιώματα πρόσβασης, οι ιδιοκτησίες, οι χρόνοι τροποποίησης και η τοποθεσία των δεδομένων στο δίσκο.
3. **Dentry (Directory Entry):** Αυτή η δομή συνδέει τα ονόματα των αρχείων με τα αντίστοιχα inodes. Είναι ουσιαστικά μια δομή που αντιπροσωπεύει ένα στοιχείο σε έναν κατάλογο. Αυτό το στοιχείο μπορεί να είναι ένα αρχείο, ένας υποκατάλογος, ή ένας συμβολικός σύνδεσμος. Κάθε Dentry συνδέεται με έναν κατάλογο και έναν αριθμό inode, και αποτελεί τη γέφυρα μεταξύ του ονόματος του στοιχείου (το οποίο είναι αυτό που βλέπει ο χρήστης) και της εσωτερικής αναπαράστασης του στοιχείου από το σύστημα αρχείων (την inode).
4. **File:** Αυτή η δομή αντιπροσωπεύει ένα ανοιχτό αρχείο και περιέχει πληροφορίες όπως η τρέχουσα θέση της "κεφαλής" ανάγνωσης/εγγραφής στο αρχείο, τα flags ανοίγματος και οι δείκτες προς τη σχετική inode και dentry.
5. **Address Space Object:** Η δομή αυτή αναπαριστά τον χώρο διευθύνσεων ενός αρχείου στη μνήμη. Κάθε αρχείο στο σύστημα έχει ένα συσχετισμένο address space object, το οποίο χρησιμοποιείται για τη διαχείριση των επιχειρήσεων I/O της μνήμης, συμπεριλαμβανομένης της προσωρινής αποθήκευσης (caching) και της χαρτογράφησης της μνήμης. Αυτό επιτρέπει στο λειτουργικό σύστημα να

διαχειρίζεται τον τρόπο με τον οποίο τα δεδομένα του αρχείου φορτώνονται στη μνήμη και γράφονται πίσω στον δίσκο.

6. **Filesystem type:** Αυτή η δομή παρέχει μια σειρά από λειτουργίες που είναι συγκεκριμένες για κάθε τύπο συστήματος αρχείων και μπορούν να χρησιμοποιηθούν για την εκτέλεση λειτουργιών όπως η δημιουργία ή η απόκτηση superblocks. Για το project μας ασχολούμαστε με ένα μόνο σύστημα αρχείων, το FAT.

3. δομή της υλοποίησης-driver του FAT στο Linux

Στον πυρήνα Linux, ο driver για το FAT filesystem (υπάρχουν διάφορες παραλλαγές, όπως vfat, msdos, κλπ.) χρησιμοποιεί ορισμένες βασικές δομές και λειτουργίες («Η υλοποίηση του συστήματος αρχείων FAT στο Linux προσδιορίζει διάφορες λειτουργίες για την διαχείριση του superblock, των inodes, των αρχείων και των καταλόγων») για την ερμηνεία και τη διαχείριση του FAT filesystem. Είναι χρήσιμο σε αυτή τη φάση να συμπεριλάβουμε και τις αντιστοιχίες δομές του VFS. Οι βασικές δομές με τις λειτουργίες τους περιλαμβάνουν:

1. **Superblock:** Αντιπροσωπεύει την υψηλότερη δομή του filesystem και περιέχει πληροφορίες που είναι κρίσιμης σημασίας για τον χειρισμό του filesystem.
 - 1.2 **struct super_block:** Αυτή είναι η δομή του **superblock** στο **VFS** του **Linux**. Περιλαμβάνει πληροφορίες για το **filesystem**, όπως οι λειτουργίες που υποστηρίζει, οι ανοιχτές λίστες αρχείων κλπ.
 - 1.3 **struct msdos_sb_info:** Αυτή είναι μια ειδική δομή που χρησιμοποιείται στο FAT driver για να αποθηκεύσει πρόσθετες πληροφορίες σχετικά με το FAT filesystem, όπως το μέγεθος του cluster, η τοποθεσία του FAT στον δίσκο και άλλες παράμετροι που είναι ειδικές για το FAT.
 - 1.4 Οι **λειτουργίες** του **superblock** ορίζονται στην δομή **struct super_operations fat_sops** του αρχείου **fs/fat/inode.c**.

2. **Inode:** Αντιπροσωπεύει ένα αρχείο στο filesystem και περιέχει μεταδεδομένα για το αρχείο, όπως το μέγεθος, τα δικαιώματα πρόσβασης, τους χρόνους τροποποίησης και πρόσβασης κ.α.

2.1 **struct inode:** Αυτή είναι η γενική δομή inode στο VFS του Linux.

2.2 **struct msdos_inode_info:** Αυτή είναι μια ειδική δομή που χρησιμοποιείται στο FAT driver για να αποθηκεύσει πρόσθετες πληροφορίες για ένα αρχείο σε ένα FAT filesystem, όπως η τοποθεσία του αρχείου στον δίσκο (η πρώτη θέση cluster) και η μορφή του ονόματος του αρχείου.

2.3 Μερικές **λειτουργίες inode** ορίζονται στην δομή **struct inode_operations fat_file_inode_operations** του αρχείου **fs/fat/file.c**.

3. **Directory Entry (ή Dentry):** Ένα dentry αποτελεί την εσωτερική αναπαράσταση ενός αρχείου ή ενός καταλόγου μέσα στον πυρήνα. Ένα **dentry** είναι συνδεδεμένο με ένα inode, που περιέχει τα μεταδεδομένα του αρχείου και πληροφορίες για τη θέση των δεδομένων του αρχείου στον φυσικό δίσκο.

3.1 **struct dentry:** Αυτή είναι η γενική δομή dentry στο VFS του Linux.

3.2 **struct fat_slot_info:** Αυτό το struct στον FAT driver του Linux περιλαμβάνει πληροφορίες σχετικά με τη θέση ενός **directory entry** στον δίσκο, συμπεριλαμβανομένου του **inode** του γονικού καταλόγου, της θέσης του **directory entry** στον δίσκο, και της θέσης του **directory entry** μέσα στο γονικό κατάλογο.

3.3 Οι **λειτουργίες** των καταλόγων ορίζονται στην δομή **struct inode_operations msdos_dir_inode_operations** του αρχείου **fs/fat/namei_msdos.c** για το σύστημα αρχείων FAT και **fs/fat/namei_vfat.c** για το σύστημα αρχείων VFAT

4. **File Operations:** Αυτή η δομή περιέχει δείκτες σε λειτουργίες που εκτελούνται σε αρχεία, όπως η ανάγνωση, η εγγραφή και η ανοιχτή.

4.2 **struct file** είναι μια δομή του VFS (Virtual File System) στον πυρήνα του Linux. Αυτή η δομή περιέχει πληροφορίες σχετικά με ένα ανοιχτό αρχείο και χρησιμοποιείται όταν ένα πρόγραμμα ανοίγει ένα αρχείο για ανάγνωση ή εγγραφή.

4.3 Οι **λειτουργίες** αρχείου ορίζονται στην δομή **struct file_operations** του αρχείου **fs/fat/file.c**. (**struct file_operations**: Αυτή είναι η γενική δομή για τις λειτουργίες αρχείων στο VFS του Linux.

4.4 Οι δομές των αρχείων στον **vfat driver** **δεν είναι διαφορετικές**. Ο **vfat driver** εξακολουθεί να χρησιμοποιεί τη δομή **struct file** του **Linux VFS** για να αντιπροσωπεύει ανοικτά αρχεία. Ωστόσο, προσθέτει επιπλέον πληροφορίες που είναι συγκεκριμένες για το FAT filesystem, όπως τα operations που αναφέραμε παραπάνω και inodes κλπ.

5. **Δομές Αποθήκευσης στο Δίσκο:** αναφέρονται σε δομές δεδομένων που χρησιμοποιούνται από τον **driver του FAT filesystem** στο **Linux** για την αναπαράσταση των δομών δεδομένων του **FAT** που αποθηκεύονται φυσικά στο δίσκο.

5.2 **struct __fat_dirent**: Αυτή η δομή χρησιμοποιείται για την αναπαράσταση των καταχωρήσεων καταλόγου σε ένα FAT filesystem. Περιέχει πληροφορίες όπως το όνομα του αρχείου, τα attributes, η ημερομηνία δημιουργίας και τροποποίησης, καθώς και η τοποθεσία του αρχείου στον δίσκο.

5.3 **struct fat_boot_sector**: Αυτή η δομή αναπαριστά τον boot sector του FAT filesystem, ο οποίος περιέχει τις βασικές πληροφορίες για το filesystem, όπως το μέγεθος των clusters, το πλήθος των FATs, και την τοποθεσία του root directory.

5.4 **struct msdos_dir_slot**: Αυτή η δομή χρησιμοποιείται για την αναπαράσταση των επεκτάσεων ονόματος αρχείου σε ένα VFAT filesystem. Επιτρέπει την αποθήκευση ονομάτων αρχείων που είναι μεγαλύτερα από τα περιορισμένα 8.3 χαρακτήρες του παραδοσιακού FAT.

5.5 **struct msdos_dir_entry**: Αυτή η δομή αναπαριστά μια καταχώρηση καταλόγου στο FAT filesystem. Περιλαμβάνει πεδία για την αποθήκευση του ονόματος του αρχείου, των attributes, του χρόνου δημιουργίας και τροποποίησης, και του cluster στον οποίο ξεκινά το αρχείο στον δίσκο.

5.6 **struct fat_boot_fsinfo**: Αυτή η δομή περιλαμβάνει πληροφορίες που είναι σχετικές με το filesystem της FAT, όπως το συνολικό αριθμό των clusters και τον αριθμό του επόμενου διαθέσιμου cluster. Αυτές οι πληροφορίες είναι χρήσιμες για την αποδοτική διαχείριση του χώρου στον δίσκο.

Στη συνέχεια θα αναλύσουμε τις εφαρμογές `cptofs/cpfromfs`, κοιτώντας τον πηγαίο κώδικα και θα τοποθετήσουμε διάφορες `printf`, στα operations του fat driver, τις οποίες αναφέραμε πιο πάνω ποιες είναι. Έτσι εκτελώντας την εφαρμογή (πχ `cptofs`), βλέπουμε έτσι στο terminal ποιες συναρτήσεις καλούνται και ποιες βασικές δομές του FAT εμπλέκονται.

cptofs/cpfromfs

Αρχικά μεταβαίνουμε στον αρχείο **cptofs.c**, από τα **tools** που μας παρέχει η **lkl**. Κοιτάμε την `main` συνάρτηση και επισημαίνουμε τα παρακάτω κύρια σημεία:

1. **lkl_disk_add(&disk)**: - Προσθέτει τον δίσκο στο σύστημα LKL. Αυτή η συνάρτηση επιστρέφει έναν αναγνωριστικό αριθμό δίσκου (`disk_id`) που χρησιμοποιείται στις επόμενες κλήσεις συναρτήσεων.
2. **lkl_start_kernel(&lkl_host_ops, "mem=100M")**: - Εκκινεί τον πυρήνα LKL καλώντας **lkl_host_ops** και με 100MB μνήμης. Τα **lkl_host_ops**, είναι operations που αναφέρονται στις λειτουργίες host που χρησιμοποιεί ο πυρήνας της LKL. Πιο συγκεκριμένα, αυτό το στοιχείο περιέχει δείκτες σε συναρτήσεις που παρέχουν τη λειτουργικότητα που απαιτείται από τον πυρήνα LKL για να αλληλεπιδράσει με τον host.

3. **lkl_mount_dev(disk_id, cla.part, cla.fsimg_type, cptoofs ? 0 : LKL_MS_RDONLY, NULL, mpoint, sizeof(mpoint))**: Το **lkl_mount_dev()** είναι όντως η συνάρτηση που πραγματοποιεί τη διαδικασία της **προσάρτησης (mount)** του δίσκου στον πυρήνα LKL. Ας δούμε λεπτομερώς την κάθε παράμετρο:
- a. **disk_id**: Αυτός είναι ο αναγνωριστικός αριθμός δίσκου που επιστράφηκε από την **lkl_disk_add()**. Αυτός χρησιμοποιείται για να αναφερθεί στον δίσκο που θα προσαρτηθεί (δηλαδή **mount**).
 - b. **cla.part**: Αυτή η παράμετρος καθορίζει τον αριθμό της διαμέρισης του δίσκου που θα προσαρτηθεί. Σε πολλά συστήματα, ένας δίσκος μπορεί να διαμεριστεί σε πολλές ξεχωριστές περιοχές ή διαμερίσεις, **κάθε μία από τις οποίες μπορεί να περιέχει ένα διαφορετικό σύστημα αρχείων**.
 - c. **cla.fsimg_type**: Αυτή η παράμετρος προσδιορίζει τον **τύπο του συστήματος** αρχείων που περιέχεται στην εικόνα του δίσκου.
 - d. **cptoofs ? 0 : LKL_MS_RDONLY**: Αυτή η παράμετρος καθορίζει τις επιλογές προσάρτησης. Αν **cptoofs** είναι true (1), τότε το σύστημα αρχείων θα προσαρτηθεί με επιλογές εγγραφής (0). Αν το **cptoofs** είναι false (0), τότε το σύστημα αρχείων θα προσαρτηθεί ως μόνο για ανάγνωση (**LKL_MS_RDONLY**), δηλαδή δεν θα επιτρέπονται εγγραφές στο σύστημα αρχείων.
 - e. **NULL**: Αυτό είναι το data argument που προβλέπει ο πυρήνας Linux για την συνάρτηση **mount**. Σε αυτήν την περίπτωση, δεν χρησιμοποιείται και γι' αυτό είναι **NULL**.
 - f. **mpoint**: Αυτή είναι η τοποθεσία στον πυρήνα LKL όπου το σύστημα αρχείων θα προσαρτηθεί. Συνήθως, αυτό θα είναι ένας κατάλογος που δημιουργείται ειδικά για αυτήν την προσάρτηση.
 - g. **sizeof(mpoint)**: Αυτό δείχνει το μέγεθος της μνήμης που έχει διατεθεί για το **mpoint**.
4. **copy_one(cla.paths[i], mpoint, cla.paths[cla.npaths - 1])**: Αυτή η συνάρτηση είναι υπεύθυνη για την πραγματική αντιγραφή των αρχείων μεταξύ του συστήματος αρχείων του host και της εικόνας του δίσκου.
5. **lkl_umount_dev(disk_id, cla.part, 0, 1000)**: Αυτή η συνάρτηση αποσυνδέει την εικόνα του δίσκου από τον πυρήνα LKL. Οι παράμετροι περιλαμβάνουν τον αναγνωριστικό αριθμό του δίσκου, τον αριθμό της τμηματοποίησης, ένα flag (0 σημαίνει αποσύνδεση) και μια προθεσμία (1000 ms).

6. `lkl_sys_halt()`; - Αυτή η συνάρτηση σταματά τον πυρήνα LKL.

Ωραία, αφού περιγράψαμε τα βασικά points των λειτουργιών `cptofs/cpfromfs`, προχωράμε να τοποθετήσουμε κλήσεις `printk` στις λειτουργίες των βασικών δομών της υλοποίησης του FAT (driver), στον πυρήνα της LKL. Θα περιγράψουμε εν συντόμως την κάθε λειτουργία η οποία πραγματοποιείται στις συναρτήσεις, οι οποίες αρχικοποιούνται ως πεδία του struct. Θα αναφέρουμε σε ποια γραμμή κάθε αρχείου τοποθετούμε τις `printk`, οι οποίες τοποθετούνται μετά τις δηλώσεις μεταβλητών, σε κάθε συνάρτηση που μας χρειάζεται. Έχουμε δώσει και διαφορετικό χρώμα για `printk` που αφορούν διαφορετικές δομές, για καλύτερη ανάγνωση στο terminal.

Τοποθέτηση `printk` στις λειτουργίες βασικών δομών του FAT driver

- I. **Superblock:** οι λειτουργίες της βασικής δομής superblock, για το FAT driver του Linux, βρίσκονται ορισμένες στη δομή **`struct super_operations fat_sops`**, στο αρχείο **`inode.c`** :

Θα σχολιάσουμε μία μία τις λειτουργίες αυτού του struct και θα τονίσουμε σε ποιο σημείο βάλαμε `printk`, για την εκάστοτε λειτουργία:

1. **`.alloc_inode = fat_alloc_inode`:** Αυτή η λειτουργία διαχειρίζεται τη δημιουργία νέων inodes στο FAT fs.
 - a. **Τοποθέτηση `printk`:** Στη γραμμή 807 του αρχείου `inode.c`
2. **`.destroy_inode = fat_destroy_inode`:** Αυτή η λειτουργία είναι υπεύθυνη για την διαγραφή ενός inode από το FAT fs.
 - a. **Τοποθέτηση `printk`:** Στη γραμμή 827 του αρχείου `inode.c`
3. **`.write_inode = fat_write_inode`:** Αυτή η λειτουργία γράφει την κατάσταση ενός inode στον δίσκο.
 - a. **Τοποθέτηση `printk`:** Στη γραμμή 979 του αρχείου `inode.c`
4. **`.evict_inode = fat_evict_inode`:** Αυτή η λειτουργία αναλαμβάνει την τη διαδικασία καθαρισμού του inode από την cache του συστήματος αρχείων.
 - a. **Τοποθέτηση `printk`:** Στη γραμμή 712 του αρχείου `inode.c`

5. **.put_super = fat_put_super:** Αυτή η λειτουργία αναλαμβάνει την καταστροφή ενός superblock, συνήθως όταν το σύστημα αρχείων κάνει umount.
 - a. **Τοποθέτηση printk:** Στη γραμμή 790 του αρχείου inode.c
6. **.statfs = fat_statfs:** Αυτή η λειτουργία παρέχει στατιστικές πληροφορίες για το σύστημα αρχείων FAT, όπως τον συνολικό αριθμό των blocks και των inodes.
 - a. **Τοποθέτηση printk:** Στη γραμμή 895 του αρχείου inode.c
7. **.remount_fs = fat_remount:** Αυτή η λειτουργία χειρίζεται την επανασύνδεση του συστήματος αρχείων, που συνήθως χρησιμοποιείται για την αλλαγή των επιλογών mount σε ένα τρέχων mount.
 - a. **Τοποθέτηση printk:** Στη γραμμή 873 του αρχείου inode.c
8. **.show_options = fat_show_options:** Αυτή η λειτουργία δείχνει τις τρέχουσες επιλογές του συστήματος αρχείων. Συνήθως, αυτό περιλαμβάνει τις επιλογές που ορίστηκαν κατά τη διάρκεια της διαδικασίας του mount και είναι χρήσιμο για την αποσφαλμάτωση και τη διαχείριση του συστήματος αρχείων.
 - a. **Τοποθέτηση printk:** Στη γραμμή 1021 του αρχείου inode.c

II. **Μνήμη(χώρος διευθύνσεων):** οι λειτουργίες της μνήμης (χώρου διευθύνσεων) για το FAT driver του Linux, βρίσκονται ορισμένες στη δομή **struct address_space_operations fat_aops**, στο αρχείο **inode.c** :

1. **.readpage = fat_readpage:** Αυτή η συνάρτηση διαβάζει μία σελίδα από το αρχείο, τοποθετώντας τα δεδομένα στη μνήμη.
 - a. **Τοποθέτηση printk:** Στη γραμμή 208-209 του αρχείου inode.c
2. **.readpages = fat_readpages:** Αντίστοιχα με την προηγούμενη, αυτή η συνάρτηση διαβάζει **πολλές** σελίδες από το αρχείο σε μια λειτουργία.
 - a. **Τοποθέτηση printk:** Στη γραμμή 218-219 του αρχείου inode.c
3. **.writepage = fat_writepage:** Αυτή η συνάρτηση γράφει μία σελίδα στο αρχείο από τη μνήμη.
 - a. **Τοποθέτηση printk:** Στη γραμμή 191-192 του αρχείου inode.c

4. **.writepages = fat_writepages:** Αντίστοιχα με την `fat_writepage`, αυτή η συνάρτηση γράφει **πολλές** σελίδες στο αρχείο σε μια λειτουργία.
 - a. **Τοποθέτηση printk:** Στη γραμμή 200-201 του αρχείου `inode.c`
5. **.write_begin = fat_write_begin:** Αυτή η συνάρτηση **αρχίζει** μια διεργασία εγγραφής στη μνήμη.
 - a. **Τοποθέτηση printk:** Στη γραμμή 242-243 του αρχείου `inode.c`
6. **.write_end = fat_write_end:** Αυτή η συνάρτηση **τερματίζει** μια διεργασία εγγραφής στη μνήμη.
 - a. **Τοποθέτηση printk:** Στη γραμμή 261-262 του αρχείου `inode.c`
7. **.direct_IO = fat_direct_IO:** Αυτή η συνάρτηση είναι υπεύθυνη για την **εκτέλεση direct I/O** ενεργειών, παρακάμπτοντας την cache του συστήματος.
 - a. **Τοποθέτηση printk:** Στη γραμμή 283-284 του αρχείου `inode.c`
8. **.bmap = _fat_bmap:** Αυτή η συνάρτηση μετατρέπει ένα λογικό block number (που χρησιμοποιείται από το λειτουργικό και τα προγράμματα) σε ένα φυσικό block number (που γράφεται στο δίσκο), γεγονός που είναι πολύ σημαντικό για την βελτιστοποίηση της ανάγνωσης και της εγγραφής στην μνήμη.
 - a. **Τοποθέτηση printk:** Στη γραμμή 341-342 του αρχείου `inode.c`

III. **Εγγραφές FAT:** Οι λειτουργίες των εγγραφών FAT, για το FAT driver του Linux, βρίσκονται ορισμένες στη δομή **struct fatent_operations fat12/16/32_ops** του αρχείου **fs/fat/fatent.c** (θα σχολιάσουμε τις λειτουργικότητες του struct `fat12_ops` μόνο, διότι διότι και τα άλλα structs είτε έχουν ίδια πεδία, είτε αρχικοποιούν παρόμοιες συναρτήσεις, θα γράψουμε όμως που είναι η `printk` για όλα τα πεδία όλων των struct) :

1. **struct fatent_operations fat12_ops** πεδία και λειτουργίες:
 - a. **.ent_blocknr = fat12_ent_blocknr:** Αυτή η λειτουργία επιστρέφει τον αριθμό του μπλοκ όπου βρίσκεται μια συγκεκριμένη εγγραφή FAT.
 - i. **Τοποθέτηση printk:** Στη γραμμή 27-28 του αρχείου `fatent.c`

- b. **.ent_set_ptr = fat12_ent_set_ptr:** Η λειτουργία αυτή ρυθμίζει έναν δείκτη FAT στη σωστή θέση μέσα στην εγγραφή.
 - i. **Τοποθέτηση printk:** Στη γραμμή 54-55 του αρχείου fatent.c
- c. **.ent_bread = fat12_ent_bread:** είναι μια λειτουργία που κάνει ανάγνωση ένα μπλοκ από τον δίσκο στον οποίο ανήκει μια συγκεκριμένη εγγραφή FAT.
 - i. **Τοποθέτηση printk:** Στη γραμμή 104 του αρχείου fatent.c
- d. **.ent_get = fat12_ent_get:** Η λειτουργία αυτή παίρνει την τρέχουσα τιμή της εγγραφής FAT στην οποία δείχνει ο δείκτης.
 - i. **Τοποθέτηση printk:** Στη γραμμή του αρχείου fatent.c
- e. **.ent_put = fat12_ent_put:** Αυτή η λειτουργία ρυθμίζει την τιμή μιας εγγραφής FAT.
 - i. **Τοποθέτηση printk:** Στη γραμμή 220 του αρχείου fatent.c
- f. **.ent_next = fat12_ent_next:** Αυτή η λειτουργία προχωρά τον δείκτη στην επόμενη εγγραφή FAT.
 - i. **Τοποθέτηση printk:** Στη γραμμή 277 του αρχείου fatent.c

2. **struct fatent_operations fat16_ops** τοποθέτηση printk:

- a. **.ent_blocknr=fat_ent_blocknr**, Τοποθέτηση printk: Στη γραμμή 42 του αρχείου fatent.c
- b. **.ent_set_ptr=fat16_ent_set_ptr**, Τοποθέτηση printk: Στη γραμμή 76 του αρχείου fatent.c
- c. **.ent_bread= fat_ent_bread**, Τοποθέτηση printk: Στη γραμμή 146 του αρχείου fatent.c
- d. **.ent_get = fat16_ent_get**, Τοποθέτηση printk: Στη γραμμή 194 του αρχείου fatent.c
- e. **.ent_put = fat16_ent_put**, Τοποθέτηση printk: Στη γραμμή 243 του αρχείου fatent.c
- f. **.ent_next = fat16_ent_next**, Τοποθέτηση printk: Στη γραμμή 326 του αρχείου fatent.c

3. **struct fatent_operations fat32_ops** τοποθέτηση printk:

- a. **.ent_blocknr= fat_ent_blocknr**, Τοποθέτηση printk: Στη γραμμή 42-43 του αρχείου fatent.c
- b. **.ent_set_ptr= fat32_ent_set_ptr**, Τοποθέτηση printk: Στη γραμμή 92 του αρχείου fatent.c
- c. **.ent_bread= fat_ent_bread**, Τοποθέτηση printk: Στη γραμμή 146 του αρχείου fatent.c
- d. **.ent_get= fat32_ent_get**, Τοποθέτηση printk: Στη γραμμή 180-181 του αρχείου fatent.c
- e. **.ent_put= fat32_ent_put**, Τοποθέτηση printk: Στη γραμμή 207 του αρχείου fatent.c
- f. **.ent_next= fat32_ent_next**, Τοποθέτηση printk: Στη γραμμή 365 του αρχείου fatent.c

IV. **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**.

1. **.llseek = generic_file_llseek:** Αυτή είναι η λειτουργία που χειρίζεται την μετακίνηση της θέσης ανάγνωσης/εγγραφής (τη λεγόμενη "θέση αρχείου") μέσα σε ένα αρχείο.

Τοποθέτηση printk: Στη γραμμή του αρχείου 147-148 `read-write.c`

2. **.read_iter = generic_file_read_iter:** Αυτή η λειτουργία χειρίζεται την ανάγνωση δεδομένων από ένα αρχείο.

Τοποθέτηση printk: Στη γραμμή 2030-2031 του αρχείου `filemap.c`

3. **.write_iter = generic_file_write_iter:** Αυτή η λειτουργία χειρίζεται την εγγραφή δεδομένων σε ένα αρχείο.

Τοποθέτηση printk: Στη γραμμή 2097-2098 του αρχείου `filemap.c`

4. **.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. Υπάρχουν 2 ορισμοί για αυτό.

Τοποθέτηση printk: Στη γραμμή 2445-2446 του αρχείου `filemap.c`

Τοποθέτηση printk: Στη γραμμή 2469-2470 του αρχείου `filemap.c`

5. **.release = fat_file_release:** Αυτή η λειτουργία κλείνει ένα αρχείο και απελευθερώνει όλους τους πόρους που έχουν δεσμευτεί από αυτό.

Τοποθέτηση printk: Στη γραμμή 164-165 του αρχείου `file.c`

6. **.unlocked_ioctl = fat_generic_ioctl:** Αυτή η λειτουργία χειρίζεται τα ειδικά αιτήματα ελέγχου του αρχείου (IOCTLs).

Τοποθέτηση printk: Στη γραμμή 133-134 του αρχείου file.c

7. **.fsync = fat_file_fsync:** Αυτή η λειτουργία χειρίζεται την ενέργεια του συγχρονισμού του αρχείου, που διασφαλίζει ότι όλες οι εκκρεμείς αλλαγές στο αρχείο έχουν εγγραφεί στον δίσκο.

Τοποθέτηση printk: Στη γραμμή 181-182 του αρχείου file.c

8. **.splice_read = generic_file_splice_read:** Αυτή η λειτουργία χειρίζεται την ανάγνωση δεδομένων από το αρχείο και την μεταφορά τους σε άλλο αρχείο χωρίς να χρειάζεται να τα κάνει copy στον χώρο του χρήστη.

Τοποθέτηση printk: Στη γραμμή 308-309 του αρχείου splice.c

9. **.fallocate = fat_fallocate:** Αυτή η λειτουργία χρησιμοποιείται για να προκαθορίσει το μέγεθος ενός αρχείου, καθορίζοντας τον χώρο στον δίσκο που θα καταλάβει. Αυτό μπορεί να βελτιώσει την απόδοση για μεγάλα αρχεία, εξασφαλίζοντας ότι ο δίσκος δεν θα χρειαστεί να ψάξει για χώρο κατά τη διάρκεια της εγγραφής.

Τοποθέτηση printk: Στη γραμμή 261-262 του αρχείου file.c

Σημείωση: Το CONFIG_COMPAT είναι μια επιλογή της διαμόρφωσης του πυρήνα που επιτρέπει την υποστήριξη παλαιότερων binary interfaces (διεπαφές που χρησιμοποιούνται για να επιτρέψουν την επικοινωνία και την αλληλεπίδραση μεταξύ των διάφορων μερών ενός λειτουργικού συστήματος ή μεταξύ διαφορετικών λογισμικών). Όταν είναι ενεργοποιημένο, το **.compat_ioctl = fat_generic_compat_ioctl** είναι επίσης διαθέσιμο. Αυτή η λειτουργία χειρίζεται τα αιτήματα ελέγχου του αρχείου (IOCTLs) για παλαιότερα binaries, και **τοποθετούμε για αυτό printk** στη γραμμή του αρχείου

- V. **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**

1. **.setattr = fat_setattr:** Αυτή η λειτουργία χρησιμοποιείται για να ορίσει τις ιδιότητες ενός inode. Οι ιδιότητες αυτές μπορεί να περιλαμβάνουν

διάφορες πληροφορίες, όπως τα δικαιώματα πρόσβασης, τον ιδιοκτήτη, τον ομάδα, το μέγεθος και τις ημερομηνίες.

Τοποθέτηση printk: Στη γραμμή 507 του αρχείου file.c

2. **.getattr = fat_getattr:** Αυτή η λειτουργία χρησιμοποιείται για να αποκτήσει τις ιδιότητες ενός inode. Ουσιαστικά, επιστρέφει τις πληροφορίες που ορίζονται από τη λειτουργία setattr.

Τοποθέτηση printk: Στη γραμμή 424 του αρχείου file.c

VI. **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.

1. **.create = vfat_create:** Αυτή η λειτουργία δημιουργεί ένα νέο αρχείο μέσα σε έναν κατάλογο.

Τοποθέτηση printk: Στη γραμμή 792-793 του αρχείου namei_vfat.c

2. **.lookup = vfat_lookup:** Αυτή η λειτουργία χρησιμοποιείται για να εντοπίσει ένα αρχείο ή κατάλογο μέσα σε έναν κατάλογο βάσει του ονόματός του.

Τοποθέτηση printk: Στη γραμμή 729-730 του αρχείου namei_vfat.c

3. **.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.

Τοποθέτηση printk: Στη γραμμή 860-861 του αρχείου namei_vfat.c

4. **.mkdir = vfat_mkdir:** Αυτή η λειτουργία δημιουργεί ένα νέο κατάλογο μέσα σε έναν κατάλογο.

Τοποθέτηση printk: Στη γραμμή 792-793 του αρχείου namei_vfat.c

5. **.rmdir = vfat_rmdir:** Αυτή η λειτουργία αφαιρεί έναν κατάλογο από έναν κατάλογο, με την προϋπόθεση ότι ο κατάλογος είναι κενός.

Τοποθέτηση printk: Στη γραμμή 890-891 του αρχείου namei_vfat.c

6. **.rename = vfat_rename:** Αυτή η λειτουργία μετονομάζει ένα αρχείο ή κατάλογο σε έναν κατάλογο.

Τοποθέτηση printk: Στη γραμμή 826-827 του αρχείου namei_vfat.c

7. **.setattr = fat_setattr:** Όπως προηγουμένως, αυτή η λειτουργία ορίζει τις ιδιότητες ενός inode.

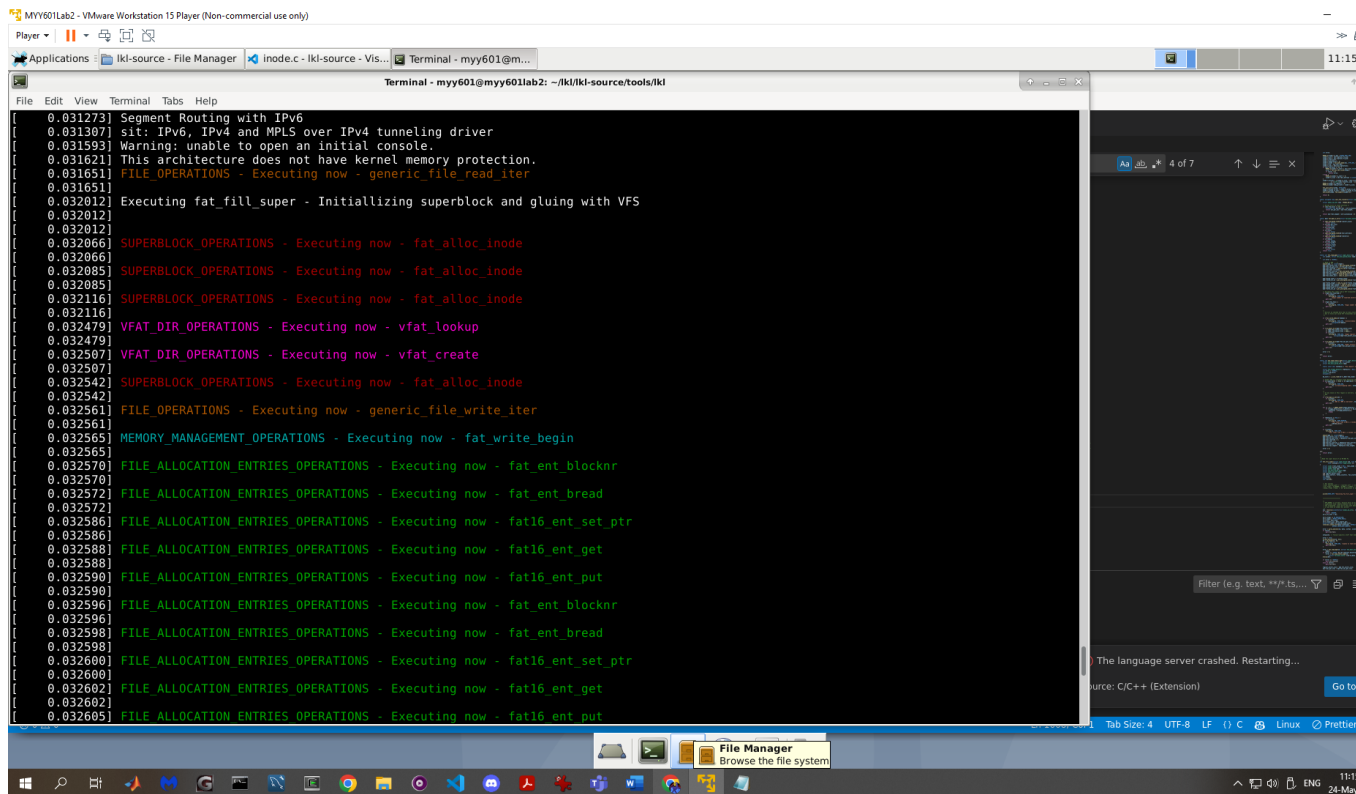
Τοποθέτηση printk: Στη γραμμή 493-494 του αρχείου file.c

8. **.getattr = fat_getattr:** Όπως προηγουμένως, αυτή η λειτουργία λαμβάνει τις ιδιότητες ενός inode.

Τοποθέτηση printk: Στη γραμμή 410-411 του αρχείου file.c

Μηνύματα στο terminal – cptofs εκτέλεση

Έτσι εκτελώντας **./cptofs -i /tmp/vfatfile -p -t vfat lklfuse.c /** , ενώ βρισκόμαστε στον φάκελο **/tools/lkl**, αφού έχουμε κάνει **make**, παίρνουμε το παρακάτω αποτέλεσμα (δίνουμε το αρχικό, έχουμε τοποθετήσει σε αυτό το σημείο **printk** και στην **fat_fill_super** του αρχείου **inode.c** , η οποία κάνει αρχικοποίηση το **superblock**):



```
0.031273] Segment Routing with IPv6
0.031307] sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
0.031593] Warning: unable to open an initial console.
0.031621] This architecture does not have kernel memory protection.
0.031651] FILE_OPERATIONS - Executing now - generic_file_read_iter
0.031651] Executing fat_fill_super - Initializing superblock and gluing with VFS
0.032012]
0.032012]
0.032066] SUPERBLOCK_OPERATIONS - Executing now - fat_alloc_inode
0.032066]
0.032085] SUPERBLOCK_OPERATIONS - Executing now - fat_alloc_inode
0.032085]
0.032116] SUPERBLOCK_OPERATIONS - Executing now - fat_alloc_inode
0.032116]
0.032479] VFAT_DIR_OPERATIONS - Executing now - vfat_lookup
0.032479]
0.032507] VFAT_DIR_OPERATIONS - Executing now - vfat_create
0.032507]
0.032542] SUPERBLOCK_OPERATIONS - Executing now - fat_alloc_inode
0.032542]
0.032561] FILE_OPERATIONS - Executing now - generic_file_write_iter
0.032561]
0.032565] MEMORY_MANAGEMENT_OPERATIONS - Executing now - fat_write_begin
0.032565]
0.032570] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat_ent_blocknr
0.032570]
0.032572] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat_ent_bread
0.032572]
0.032586] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_set_ptr
0.032586]
0.032588] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
0.032588]
0.032590] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_put
0.032590]
0.032596] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat_ent_blocknr
0.032596]
0.032596] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat_ent_bread
0.032596]
0.032600] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_set_ptr
0.032600]
0.032602] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
0.032602]
0.032605] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_put
```

Παρατηρούμε ότι εκτελούνται γενικά πρώτα λειτουργίες superblock.

Τώρα συγκριτικά, δοκιμάζοντας πάλι `cptofs` , με ένα πολύ μικρότερο αρχείο `teos_file_experiment.c` , το οποίο το έχουμε τοποθετήσει στο `/tools/lkl/` , με εντολή

`./cptofs -i /tmp/vfatfile -p -t vfat teos_file_experiment.c /`

Έχουμε:

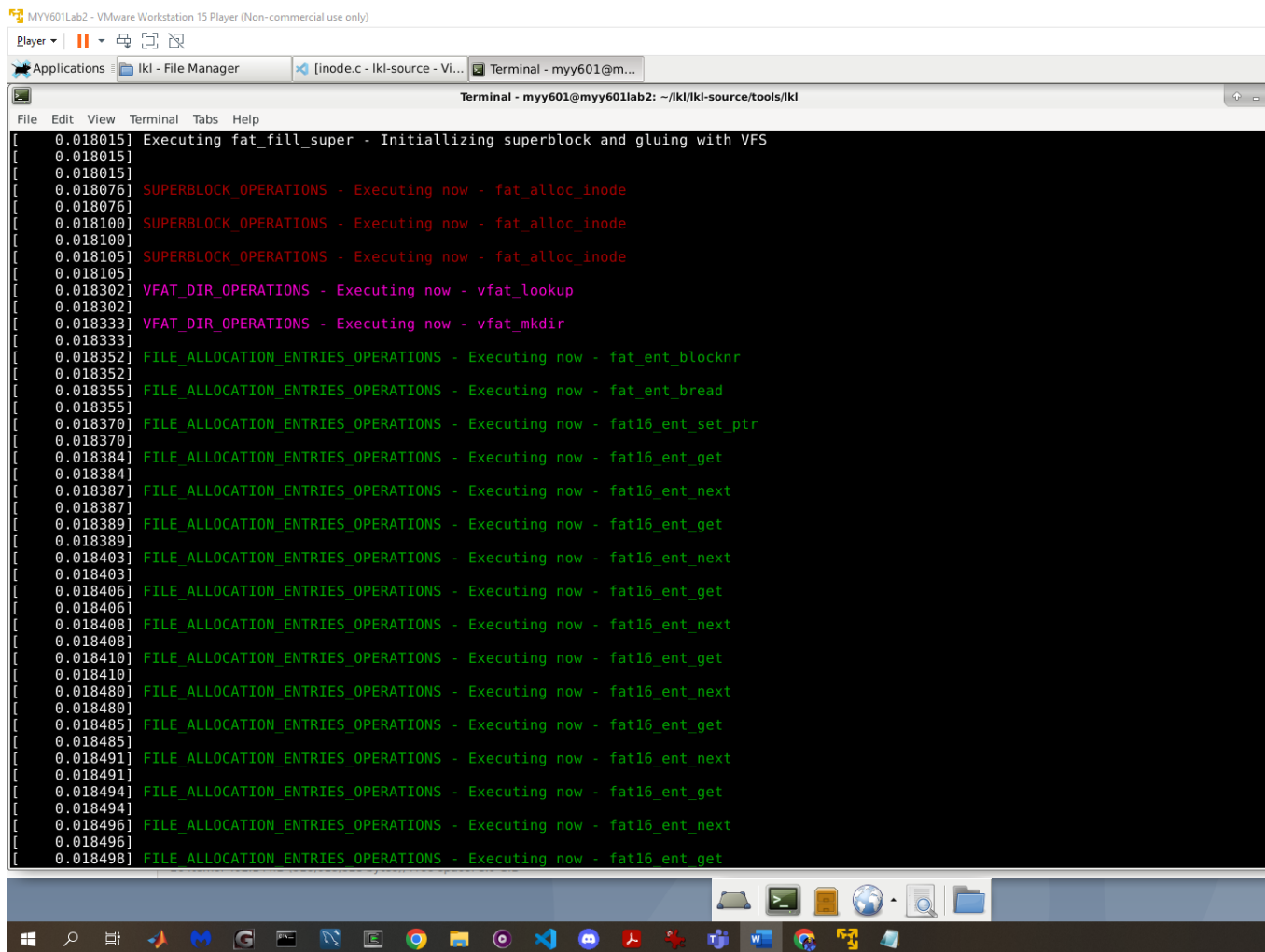
```
0.018974]
0.018974]
0.019056] SUPERBLOCK_OPERATIONS - Executing now - fat_alloc_inode
0.019056]
0.019082] SUPERBLOCK_OPERATIONS - Executing now - fat_alloc_inode
0.019082]
0.019086] SUPERBLOCK_OPERATIONS - Executing now - fat_alloc_inode
0.019086]
0.019341] VFAT_DIR_OPERATIONS - Executing now - vfat_lookup
0.019341]
0.019364] SUPERBLOCK_OPERATIONS - Executing now - fat_alloc_inode
0.019364]
0.019412] INODE_OPERATIONS - Executing now - fat_setattr
0.019412]
0.019431] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat_ent_blocknr
0.019431]
0.019434] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat_ent_bread
0.019434]
0.019450] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_set_ptr
0.019450]
0.019456] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
0.019456]
0.019458] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_put
0.019458]
0.019470] FILE_OPERATIONS - Executing now - generic_file_write_iter
0.019470]
0.019487] MEMORY_MANAGEMENT_OPERATIONS - Executing now - fat_write_begin
0.019487]
0.019492] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat_ent_blocknr
0.019492]
0.019507] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat_ent_bread
0.019507]
0.019509] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_set_ptr
0.019509]
0.019511] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
0.019511]
0.019516] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_next
0.019516]
0.019521] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
0.019521]
0.019523] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_next
0.019523]
0.019523] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
0.019525]
0.019525]
```

Συγκριτικά παρατηρούμε λιγότερες λειτουργίες για τους FAT πίνακες, λόγω του μεγέθους του αρχείου.

Στη συνέχεια εκτελώντας:

```
./cptofs -i /tmp/vfatfile -p -t vfat teos_empty_directory_experiment /
```

Όπου με το οποίο κάνουμε μεταφορά άδειου φακέλου που περιέχει μέσα αρχείο(έχουμε και γεμάτο φάκελο στα tools), και παρατηρούμε τώρα ότι πραγματοποιούνται και οι λειτουργίες των directories όπως **vfat_mkdir** , για δημιουργία νέου φακέλου:



```
[ 0.018015] Executing fat_fill_super - Initiallizing superblock and gluing with VFS
[ 0.018015]
[ 0.018015]
[ 0.018076] SUPERBLOCK_OPERATIONS - Executing now - fat_alloc_inode
[ 0.018076]
[ 0.018100] SUPERBLOCK_OPERATIONS - Executing now - fat_alloc_inode
[ 0.018100]
[ 0.018105] SUPERBLOCK_OPERATIONS - Executing now - fat_alloc_inode
[ 0.018105]
[ 0.018302] VFAT_DIR_OPERATIONS - Executing now - vfat_lookup
[ 0.018302]
[ 0.018333] VFAT_DIR_OPERATIONS - Executing now - vfat_mkdir
[ 0.018333]
[ 0.018352] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat_ent_blocknr
[ 0.018352]
[ 0.018355] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat_ent_bread
[ 0.018355]
[ 0.018370] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_set_ptr
[ 0.018370]
[ 0.018384] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
[ 0.018384]
[ 0.018387] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_next
[ 0.018387]
[ 0.018389] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
[ 0.018389]
[ 0.018403] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_next
[ 0.018403]
[ 0.018406] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
[ 0.018406]
[ 0.018408] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_next
[ 0.018408]
[ 0.018410] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
[ 0.018410]
[ 0.018480] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_next
[ 0.018480]
[ 0.018485] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
[ 0.018485]
[ 0.018491] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_next
[ 0.018491]
[ 0.018494] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
[ 0.018494]
[ 0.018496] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_next
[ 0.018496]
[ 0.018498] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
```

ΠΡΟΣΟΧΗ αυτά τα screenshots δεν περιλαμβάνουν τα τελευταία βήματα. Θα ακολουθήσουν και άλλα στο τέλος.

Επίσης τελειώνοντας εδώ, σχολιάζουμε περαιτέρω την:

./cptofs -i /tmp/vfatfile -p -t vfat lklfuse.c / :

-i /tmp/vfatfile: Αυτό καθορίζει τον εικονικό δίσκο όπου θα αντιγραφεί το αρχείο. Σε αυτήν την περίπτωση, ο εικονικός δίσκος είναι το /tmp/vfatfile.

-p: Αυτή η επιλογή είναι για την εμφάνιση μηνυμάτων του kernel.

-t vfat: Αυτό καθορίζει τον τύπο του filesystem που χρησιμοποιείται στον εικονικό δίσκο. Σε αυτήν την περίπτωση, είναι το vfat.

lklfuse.c: Αυτό είναι το αρχείο που θα αντιγραφεί από το host σύστημα στον εικονικό δίσκο.

/: Αυτό καθορίζει τη θέση στην οποία θα αντιγραφεί το αρχείο στον εικονικό δίσκο. Σε αυτήν την περίπτωση, είναι το root directory του filesystem.

2.2 Σε τί μας χρησιμεύει το journal

Journaling είναι μια μέθοδος που ενισχύει την αξιοπιστία ενός συστήματος αρχείων. Χρησιμοποιείται για να αποφύγει την εκτενή διαδικασία ελέγχου της συνοχής του δίσκου (fsck) μετά από μια απροσδόκητη διακοπή, κάτι που μπορεί να είναι χρονοβόρο και να οδηγήσει σε απώλεια δεδομένων. Αυτό επιτυγχάνεται μέσω του journaling, όπου οι ενδεχόμενες αλλαγές που πρέπει να γίνουν στο σύστημα αρχείων καταγράφονται πρώτα σε ένα "journal" πριν εφαρμοστούν. Έτσι, σε περίπτωση διακοπής της διαδικασίας, το σύστημα αρχείων μπορεί να ανατρέξει στο journal για να κατανοήσει τι ήταν πρόθεση να γίνει και να επαναφέρει το **σύστημα σε συνεπή κατάσταση με λιγότερο χρόνο και απώλεια δεδομένων**.

Από την άλλη πλευρά, θα έπρεπε να αναφέρουμε ότι το journaling αυξάνει τον αριθμό των εγγραφών στο δίσκο, καθώς κάθε αλλαγή πρέπει να καταγραφεί δύο φορές - μία στο journal και μία στον δίσκο. Παρά το επιπρόσθετο φόρτο εργασίας, η αυξημένη αξιοπιστία και η δυνατότητα άμεσης ανάκτησης συνήθως αποζημιώνουν την πρόσθετη επεξεργασία.

Έλλειψη journaling για το FAT σύστημα αρχείων

Το FAT (File Allocation Table), που είναι ένα από τα παλαιότερα συστήματα αρχείων, δεν παρέχει υποστήριξη για journaling. Αυτό σημαίνει ότι αν υπάρξει διακοπή τροφοδοσίας ή σύστημα που κάνει crash κατά τη διάρκεια μιας λειτουργίας εγγραφής, μπορεί να προκληθεί διαφθορά των δεδομένων. Αυτό είναι ένας από τους λόγους που το FAT χρησιμοποιείται συχνά σε μέσα αποθήκευσης όπως οι κάρτες SD, όπου το περιορισμένο μέγεθος και η απλότητα είναι σημαντικότερα από την ανάγκη για υψηλή αξιοπιστία ή απόδοση.

Στη συνέχεια θα αναλύσουμε σε ποιο σημείο της υλοποίησης του FAT στην LKL θα δημιουργήσουμε το journal, και ποια πεδία των βασικών δομών κρίνουμε θα αποθηκεύσουμε σε αυτό. Χρησιμοποιούμε επίσης διάφορες επιπλέον printk για περαιτέρω επαλήθευση.

2.3 Journal entries για τις βασικές δομές του FAT

Αρχική δημιουργία journal

Αναφέραμε και προηγουμένως, εκτελώντας την εφαρμογή `cptofs`, οι λειτουργίες του `superblock` είναι αυτές που εκτελούνται αρχικά, που είναι και λογικό. Έτσι σκεφτόμαστε ότι θα ήταν φρόνιμο να κάνουμε τη δημιουργία του `journal` μέσα σε αυτές τις συναρτήσεις που εκτελούν τις λειτουργίες του `superblock`. Ειδικά θα χρησιμοποιήσουμε την **`fat_fill_super`**.

Η `fat_fill_super` (βρίσκεται στο αρχείο `inode.c`) είναι η συνάρτηση που καλείται από τον πυρήνα του Linux για να δημιουργήσει και να αρχικοποιήσει ένα νέο `superblock`, όταν προσαρτάται ένα FAT filesystem (Παρόμοια συνάρτηση για αρχικοποίηση του `inode`, είναι η `fat_fill_inode`). Το `superblock` είναι ουσιαστικά η "ρίζα" του συστήματος αρχείων, περιέχοντας πληροφορίες για τη διάρθρωση και την κατάσταση του συστήματος αρχείων. Ανοίγοντας το αρχείο του `journal` στη `fat_fill_super`, εξασφαλίζεται ότι το `journal` θα είναι διαθέσιμο για χρήση αμέσως μετά την προσάρτηση του συστήματος αρχείων. Επιπλέον, είναι καλή πρακτική να διαχειριζόμαστε όλους τους απαιτούμενους πόρους του συστήματος αρχείων κατά την αρχικοποίηση του `superblock`, για να είναι πιο ευανάγνωστη και οργανωμένη η διαδικασία μας.

Αρχικά στο αρχείο `fat.h`, κάνουμε `include` τις βιβλιοθήκες (για χρήση και από άλλα αρχεία):

```
// Added Code
```

```
// For sys_write usage, generally for system calls
```

```
#include <linux/syscalls.h>
```

```
#include <linux/fcntl.h>
```

```
#include <linux/err.h> Για χρήση της sys_open και διαφόρων error μηνυμάτων, στις γραμμές 10-14.
```

Έτσι στη συνέχεια, μεταβαίνοντας στη συνάρτηση `fat_fill_super`, προσθέτουμε αυτές τις γραμμές για χρήση του `journal`, στις γραμμές 1662-1665 (έχουμε και την `printk` πιο κάτω, που μας ενημερώνει ότι είμαστε σε αυτήν τη συνάρτηση):

```
// For journal
char *journal_path = "journal.txt"; // Creating journal file in the root directory of the fs
int flags = O_APPEND | O_RDWR | O_CREAT; // Flags , write at the end (cause we
journal) , read-write and create if it doesn't exist
mode_t mode = 0666; // All permissions to all
```

Ας εξετάσουμε κάθε γραμμή πιο αναλυτικά:

char *journal_path = "journal.txt": Αυτή η γραμμή ορίζει το μονοπάτι του αρχείου καταγραφής (journal) που θα δημιουργηθεί. Το όνομα του αρχείου είναι "journal.txt" και θα δημιουργηθεί στον τρέχοντα κατάλογο (καθώς δεν έχει προηγουμένως καθοριστεί διαφορετικά).

int flags = O_APPEND | O_RDWR | O_CREAT: Οι σημαίες που ορίζονται εδώ καθορίζουν τον τρόπο λειτουργίας του αρχείου καταγραφής. O_APPEND σημαίνει ότι όλες οι εγγραφές θα προστίθενται στο τέλος του αρχείου, O_RDWR επιτρέπει την ανάγνωση και την εγγραφή στο αρχείο, και O_CREAT δημιουργεί το αρχείο αν δεν υπάρχει ήδη.

mode_t mode = 0666: Αυτή η γραμμή καθορίζει τα δικαιώματα του αρχείου, στην περίπτωση που το αρχείο δημιουργείται. Τα modes 0666 επιτρέπουν την **ανάγνωση και την εγγραφή στο αρχείο για όλους τους χρήστες**.

Χρειαζόμαστε και έναν file descriptor, για χρήση της **sys_open**. Είναι λογικό να τον ορίσουμε στο **fat.h** header file, που γίνονται ορισμοί και το βλέπουν όλα τα αρχεία. Εφόσον από ότι είπαμε στη φάση αυτή είμαστε στην αρχικοποίηση του superbblock, τοποθετούμε τον file descriptor στο struct **ms_dos_info** , στη γραμμή 105 : **int file_desc;**

Γυρνώντας τώρα στη συνάρτηση **fat_fille_super**, αφού έχουμε ορίσει ήδη τις παραμέτρους για το journal, κάνουμε μία διερεύνηση για το που θα ήταν καλό να το δημιουργήσουμε. Ένα καλό σημείο θα ήταν, μετά την κλήση της **fat_set_state**.

Η **fat_set_state** θέτει την κατάσταση του συστήματος αρχείων ως "dirty" ή "clean", και αυτό πρέπει να γίνει πριν αρχίσει το journaling, ώστε το journal να έχει σωστές πληροφορίες σχετικά με την κατάσταση του συστήματος αρχείων. Με αυτόν τον τρόπο, εάν κάποια αλλαγή δεν εφαρμόζεται σωστά κατά τη διάρκεια του journaling, το σύστημα

αρχείων θα είναι σε θέση να αναγνωρίσει ότι υπάρχει κάποιο πρόβλημα με την κατάσταση των δεδομένων.

Έτσι δημιουργούμε το journal στη γραμμή 1934 του αρχείου `inode.c`, με την εντολή:

`sbi->file_desc = sys_open(journal_path, flags, mode);` (έχει οριστεί προηγουμένως στη `fat_fill_super` το `struct msdos_sb_info *sbi;`).

Έτσι έχουμε δημιουργήσει το αρχείο journal. Τώρα θα διερευνήσουμε ποια πεδία των βασικών δομών του fat-driver του Linux, αξίζει να καταγράψουμε σε αυτό.

Superblock

Η βασική δομή του superblock για την υλοποίηση του FAT στο Linux, υλοποιείται στη δομή **`struct msdos_sb_info`** (όπως είπαμε και προηγουμένως) στο αρχείο **`fat.h`**.

Ο τρόπος που προσεγγίζουμε το journaling είναι, κοιτάμε όλα τα πεδία των βασικών δομών `struct`, προσπαθώντας να κατανοήσουμε ποια πληροφορία περιέχει το καθένα, και αν αξίζει να τα καταγράψουμε στο journal. Αν συμβαίνει το δεύτερο, ψάχνουμε `references` των πεδίων αυτών σε συναρτήσεις που υλοποιούνται (αλλάζουν), και μέσα σε αυτές τις συναρτήσεις τα καταγράφουμε στο journal (με `sys_write`) και επίσης τυπώνουμε με `printf` ότι καταγράφουμε. Για τη χρήση `sys_write`, δημιουργούμε `buffer` σε κάθε συνάρτηση, αρχικοποιώντας τον δυναμικά.

Αναφέρουμε σύντομα όλα τα πεδία και τις λειτουργίες τους, του **`msdos_sb_info`** παρακάτω, και επισημαίνουμε με **κόκκινο χρώμα** ποια θα κρατήσουμε για το journal:

1. **`unsigned short sec_per_clus;`** - Αριθμός των sectors ανά cluster. Δεν χρειάζεται να καταγραφεί στο journal, καθώς είναι μια σταθερή τιμή για κάθε σύστημα αρχείων και δεν αλλάζει κατά την διάρκεια της λειτουργίας.
2. **`unsigned short cluster_bits;`** - Το \log_2 του μεγέθους του cluster. Όπως και παραπάνω, αυτό δεν χρειάζεται να καταγραφεί στο journal.
3. **`unsigned int cluster_size;`** - Μέγεθος του cluster. Αυτό είναι άλλος ένας δείκτης του μεγέθους των clusters και, καθώς δεν αλλάζει κατά τη διάρκεια της λειτουργίας, δεν χρειάζεται να καταγραφεί στο journal.
4. **`unsigned char fats, fat_bits;`** - Αριθμός των FATs και των bits του FAT (12,16,32). Αυτές οι τιμές πιθανώς δεν χρειάζεται να καταγραφούν στο journal, καθώς είναι σταθερές για κάθε σύστημα αρχείων.

5. **unsigned short fat_start;** - Αρχική θέση της FAT. Αυτό είναι ένα σταθερό χαρακτηριστικό του συστήματος αρχείων και δεν χρειάζεται να καταγραφεί στο journal.
6. **unsigned long fat_length;** - Μήκος της FAT σε τομείς. Είναι ένα άλλο σταθερό χαρακτηριστικό του συστήματος αρχείων και δεν χρειάζεται να καταγραφεί στο journal.
7. **unsigned long dir_start;** - Αρχικός τομέας του root directory. Σταθερή τιμή, δεν χρειάζεται καταγραφή στο journal.
8. **unsigned short dir_entries;** - Αριθμός των καταχωρήσεων στο root directory. Αυτή η τιμή είναι σταθερή για συστήματα FAT12 και FAT16, αλλά σε FAT32 το root directory μπορεί να επεκταθεί, οπότε ίσως χρειαστεί καταγραφή στο journal. Εμείς θα το αφήσουμε.
9. **unsigned long data_start;** - Πρώτος τομέας δεδομένων. Είναι σταθερή τιμή, δεν χρειάζεται καταγραφή στο journal.
10. **unsigned long max_cluster;** - Μέγιστος αριθμός clusters. Αυτό είναι ένα σταθερό χαρακτηριστικό του συστήματος αρχείων και δεν χρειάζεται να καταγραφεί στο journal.
11. **unsigned long root_cluster;** - Πρώτο cluster του root directory. Αυτό είναι σταθερό για ένα σύστημα αρχείων και δεν χρειάζεται να καταγραφεί.
12. **unsigned long fsinfo_sector;** - Αριθμός τομέα των πληροφοριών του FAT32. Αυτό είναι σταθερό για ένα σύστημα αρχείων και δεν χρειάζεται να καταγραφεί στο journal.
13. **struct mutex fat_lock;** - **Mutex** που χρησιμοποιείται για την προστασία της FAT. Δεν χρειάζεται να καταγραφεί στο journal.
14. **struct mutex nfs_build_inode_lock;** - **Mutex** που χρησιμοποιείται για την προστασία της δημιουργίας inodes στο NFS (Network File System). Δεν χρειάζεται να καταγραφεί στο journal.
15. **struct mutex s_lock;** - **Mutex** για γενικούς σκοπούς. Δεν χρειάζεται να καταγραφεί στο journal.
16. **unsigned int prev_free;** - Προηγούμενος εκχωρημένος αριθμός cluster. Αυτό θα μπορούσε να καταγραφεί στο journal, καθώς η τιμή του μπορεί να αλλάξει κατά τη διάρκεια της λειτουργίας.
17. **unsigned int free_clusters;** - Αριθμός των διαθέσιμων clusters. Αυτό είναι μια τιμή που μπορεί να αλλάξει και θα ήταν σκόπιμο να καταγραφεί στο journal.

18. **unsigned int free_clus_valid;** - Δείχνει εάν η free_clusters είναι έγκυρη. Αυτό θα μπορούσε να καταγραφεί στο journal, καθώς η τιμή του μπορεί να αλλάξει κατά τη διάρκεια της λειτουργίας. Εμείς θα το αφήσουμε.
19. **struct fat_mount_options options;** - Επιλογές που χρησιμοποιήθηκαν κατά την προσάρτηση του συστήματος αρχείων. Αυτές οι επιλογές είναι σταθερές μετά την προσάρτηση και δεν χρειάζεται να καταγραφούν στο journal.
20. **struct nls_table *nls_disk;** - Ο πίνακας που χρησιμοποιείται για την κωδικοποίηση των ονομάτων αρχείων στο δίσκο. Δεν χρειάζεται να καταγραφεί στο journal.
21. **struct nls_table *nls_io;** - Ο πίνακας που χρησιμοποιείται για την κωδικοποίηση των ονομάτων αρχείων κατά την είσοδο/έξοδο. Δεν χρειάζεται να καταγραφεί στο journal.
22. **const void *dir_ops;** - Πεδίο για την αποθήκευση των λειτουργιών του directory που χρησιμοποιούνται σε αυτό το σύστημα αρχείων. Αυτές οι λειτουργίες είναι σταθερές για ένα δεδομένο σύστημα αρχείων και δεν χρειάζεται να καταγραφούν στο journal.
23. **int dir_per_block;** - Αριθμός καταχωρήσεων directory ανά block. Αυτό είναι μια σταθερή τιμή για ένα σύστημα αρχείων και δεν χρειάζεται να καταγραφεί στο journal.
24. **int dir_per_block_bits;** - Το λογάριθμο του αριθμού των καταχωρήσεων directory ανά block. Όπως το προηγούμενο, είναι σταθερό για ένα σύστημα αρχείων και δεν χρειάζεται να καταγραφεί στο journal.
25. **unsigned int vol_id;** - Ο αναγνωριστικός αριθμός τόμου του συστήματος αρχείων. Αυτό είναι σταθερό για ένα σύστημα αρχείων και δεν χρειάζεται να καταγραφεί στο journal.
26. **int fatent_shift;** - Χρησιμοποιείται στις λειτουργίες του FAT entry. Αυτή η τιμή είναι σταθερή για ένα σύστημα αρχείων και δεν χρειάζεται να καταγραφεί στο journal.
27. **const struct fatent_operations *fatent_ops;** - Πεδίο για την αποθήκευση των λειτουργιών της καταχώρησης FAT που χρησιμοποιούνται σε αυτό το σύστημα αρχείων. Αυτές οι λειτουργίες είναι σταθερές για ένα σύστημα αρχείων και δεν χρειάζεται να καταγραφούν στο journal.
28. **struct inode *fat_inode;** - Το inode του File Allocation Table (FAT). Αυτό μπορεί να αλλάξει κατά τη διάρκεια της λειτουργίας και μπορεί να χρειαστεί να

καταγραφεί στο journal. Είναι απαραίτητο , αλλά δεν το καταγράψαμε, γράψαμε όμως το επόμενο struct.

29. **struct inode *fsinfo_inode;** - Ο inode των πληροφοριών του συστήματος αρχείων. Όπως το fat_inode, αυτό μπορεί να αλλάξει και μπορεί να χρειαστεί να καταγραφεί στο journal. Θα το κρατήσουμε.
30. **struct ratelimit_state ratelimit;** - Χρησιμοποιείται για τον περιορισμό των μηνυμάτων που εκτυπώνονται στο kernel log. Δεν χρειάζεται να καταγραφεί στο journal.
31. **spinlock_t inode_hash_lock;** - Spinlock που χρησιμοποιείται για την προστασία του inode hash table. Δεν χρειάζεται να καταγραφεί στο journal.
32. **struct hlist_head inode_hashtable[FAT_HASH_SIZE];** - Hash table για την αναζήτηση inodes. Μπορεί να χρειαστεί να καταγραφεί στο journal, ανάλογα με το πώς το journaling εφαρμόζεται. Εμείς δεν θα το καταγράψουμε.
33. **spinlock_t dir_hash_lock;** - Αυτός ο spinlock χρησιμοποιείται για την προστασία του hash table του directory. Δεν χρειάζεται να καταγραφεί στο journal, καθώς είναι σχετικό με τον συγχρονισμό των πολλαπλών threads και όχι με την κατάσταση του συστήματος αρχείων.
34. **struct hlist_head dir_hashtable[FAT_HASH_SIZE];** - Hash table για την αναζήτηση καταλόγων. Αυτό μπορεί να χρειαστεί να καταγραφεί στο journal, ανάλογα με το πώς το journaling εφαρμόζεται στο σύστημα αρχείων. Εμείς θα το αφήσουμε.
35. **unsigned int dirty;** - Αυτό το πεδίο διατηρεί την κατάσταση του συστήματος αρχείων πριν από την προσάρτηση. Μπορεί να είναι χρήσιμο να καταγράφεται αυτό στο journal, ειδικά εάν το σύστημα αρχείων μπορεί να μεταβάλλει την κατάστασή του ενώ είναι προσαρτημένο. Εμείς θα το αφήσουμε.
36. **struct rcu_head rcu;** - Χρησιμοποιείται για την υποστήριξη του Read-Copy-Update (RCU) μηχανισμού στον kernel του Linux. Δεν χρειάζεται να καταγραφεί στο journal, καθώς είναι σχετικό με τον συγχρονισμό και όχι με την κατάσταση του συστήματος αρχείων.
37. **int file_desc;** - Ο file descriptor που προσθέσαμε, δεν θα καταγραφεί διότι με αυτόν ανοίγουμε και δημιουργούμε το journal.

Έτσι σχολιάζουμε παρακάτω τί κάνουμε για το κάθε πεδίο που μόλις αναφέραμε, που ενδιαφερόμαστε να το κρατήσουμε στο journal.

Αναφέρουμε γενικά τον κώδικα που θα χρησιμοποιούμε συνέχεια για καταγραφή στο journal, τύπωση οθόνη των πεδίων που μας ενδιαφέρουν με printk .

Αρχικά θα δηλώνουμε στην αρχή κάθε συνάρτησης, εκεί που γίνονται οι δηλώσεις buffer για sys_write και μήκος buffer:

```
// Buffer for sys_write
char buffer[1024];
int length;
```

Και μετά όπου χρειάζεται τον παρακάτω κώδικα για καταγραφή και εκτύπωση, μετά από αλλαγή κάθε πεδίου που μας ενδιαφέρει (είναι γενικό το μήνυμα παρακάτω, το προσαρμόζουμε για το εκάστοτε πεδίο):

```
// Added code
length = scnprintf(buffer, sizeof(buffer), "plhroforia gia vasikh domh, poio pedio mas
endiaferi, se poia synarthsh vriskomaste , onoma_pediou(me katallhlo format dipla):
%d\n", struct->pedio_struct);
if (length > 0) {
    sys_write(sbi->file_desc, buffer, length); // Write to journal
    printk(KERN_INFO "%s", buffer);    // Print to the screen
}
// Finished write and print for here
```

Θα αναλύσουμε αυτήν την λογική, διότι θα είναι η ίδια που θα χρησιμοποιήσουμε για τα άλλα πεδία:

scnprintf: Αυτή η συνάρτηση χρησιμοποιείται για τη δημιουργία μιας συμβολοσειράς, η οποία αποθηκεύει μηνύματα και πληροφορίες σχετικά με την κατάσταση της fat12_ent_bread συνάρτησης. Η scnprintf επιστρέφει τον αριθμό των χαρακτήρων που εγγράφησε στο buffer (χωρίς να συμπεριλαμβάνει το null terminator), που αποθηκεύεται στην length.

sys_write: Αν η scnprintf επέστρεψε έναν αριθμό μεγαλύτερο από το μηδέν (δηλαδή, αν κατάφερε να γράψει κάποιο μήνυμα στο buffer), τότε γίνεται μια κλήση στην sys_write, που γράφει το μήνυμα από το buffer στο αρχείο καταγραφής (journal) που περιγράφεται από το file_desc του sbi. Η sys_write θα γράψει length χαρακτήρες από το buffer στο αρχείο.

printk: Μετά την εγγραφή στο αρχείο καταγραφής, το ίδιο μήνυμα εμφανίζεται επίσης στο terminal (μήνυμα: "Eggrafh pediwn msdos_sb_info - pedio struct inode *fat_inode (emfwleumeno struct) - sth sunarthsh fat12_ent_bread - krataw pedio, nr_bhs: %d\n")

*Αξίζει επίσης να σημειωθεί ότι η **scnprintf**, είναι μια συνάρτηση που παρέχεται από τον Linux kernel, και είναι παρόμοια με την **snprintf** (παρέχεται από τη βιβλιοθήκη C), αλλά **επιστρέφει τον πραγματικό αριθμό χαρακτήρων που γράφτηκαν στον buffer**, ενώ η **snprintf** επιστρέφει τον αριθμό των χαρακτήρων που **θα έγραφε αν υπήρχε αρκετός χώρος στον buffer**.*

Συνεχίζουμε για τα αλλαγμένα πεδία του msdos_sb_info

1. unsigned int prev_free

Βρίσκουμε περιπτώσεις-συναρτήσεις που το πεδίο αυτό αλλάζει

Για τη συνάρτηση **fat_fill_super** του αρχείου **inode.c**

Εδώ παρατηρούμε υπάρχει ήδη δήλωση και αρχικοποίηση

```
struct msdos_sb_info *sbi = MSDOS_SB(sb);
```

Προσθέτουμε και εδώ λογική για buffer και length, στις δηλώσεις μεταβλητών και μετά τον κώδικα που έχουμε αναφέρει πιο πάνω για sys_write και printk στη γραμμή 1830,

όπου παρατηρούμε ότι αλλάζει το πεδίο αυτό, με προσαρμοσμένο μήνυμα printf και χρήση format %u, διότι το πεδίο είναι τύπου unsigned int.

Επίσης παρατηρούμε αλλαγές αυτού του πεδίου στις γραμμές 1944 και 1953 και βάζουμε και εκεί κώδικα για sys_write και printf.

2. unsigned int free_clusters;

Για τη συνάρτηση fat_fill_super του αρχείου inode.c

Ξανά, είμαστε στην ίδια συνάρτηση με πριν, έχουν οριστεί sbi, buffer και length και απλά τοποθετούμε τον κώδικα για sys_write και προσαρμοσμένη printf, στην αρχικοποίηση του πεδίου και εκεί που αλλάζει: στη γραμμή 1821, 1871 και 1953.

3. struct inode *fsinfo_inode;

Το πεδίο αυτό είναι ήδη struct τύπου inode. Αυτή η συγκεκριμένη δομή inode είναι component του VFS. Είναι ορισμένη στο αρχείο fs.h στη γραμμή 554. Κοιτώντας τα πεδία αυτού του struct, διακρίνουμε 2 τα οποία είναι ενδιαφέρον για καταγραφή:

i_size: Αυτό είναι το μέγεθος του αρχείου σε bytes. Καταγράφοντας αυτό, μπορούμε να παρακολουθήσουμε αλλαγές στο μέγεθος του αρχείου.

i_blocks: Αυτός είναι ο αριθμός των blocks που δεσμεύει το αρχείο. Καταγράφοντας αυτό, μπορούμε να παρακολουθήσουμε πότε αλλάζει ο αριθμός των blocks που χρησιμοποιεί ένα αρχείο.

Πρέπει να σιγουρέψουμε ότι τα πεδία αυτά που πρέπει να καταγράψουμε είναι τύπου MSDOS FSINFO, διότι με αυτό ασχολούμαστε να καταγράψουμε τώρα.

Για το πεδίο i_size της συνάρτησης fat_calc_dir_size του αρχείου inode.c

Βλέπουμε ότι υπάρχει ήδη η αρχικοποίηση

```
struct msdos_sb_info *sbi = MSDOS_SB(inode->i_sb);
```

Βάζουμε αρχικά τον buffer και το length στις δηλώσεις μεταβλητών, τον κώδικα καταγραφής και printf (με format %lld), σε σημείο που αλλάζει το i_size, δηλαδή στις γραμμές 509 και 523.

Για το πεδίο i_size της συνάρτησης fat_read_root του αρχείου inode.c

Παρατηρούμε ότι ήδη υπάρχει η αρχικοποίηση

```
struct msdos_sb_info *sbi = MSDOS_SB(inode->i_sb);
```

Βάζουμε αρχικά τον buffer και το length στις δηλώσεις μεταβλητών, τον κώδικα καταγραφής και printk (με format **%lld**) , σε σημείο που αλλάζει το i_size, δηλαδή στις γραμμές 1497 και 1507.

Για το πεδίο **i blocks** της συνάρτησης **fat fill inode** του αρχείου **inode.c**

Παρατηρούμε ότι ήδη υπάρχει η αρχικοποίηση

```
struct msdos_sb_info *sbi = MSDOS_SB(inode->i_sb);
```

Βάζουμε αρχικά τον buffer και το length στις δηλώσεις μεταβλητών, τον κώδικα καταγραφής και printk (με format **%lld**, ίδιο με πριν) , σε σημείο που αλλάζει το i_blocks, δηλαδή στη γραμμή 618.

Για το πεδίο **i blocks** της συνάρτησης **fat read root** του αρχείου **inode.c**

Παρατηρούμε ότι ήδη υπάρχει η αρχικοποίηση

```
struct msdos_sb_info *sbi = MSDOS_SB(inode->i_sb);
```

Υπάρχει ήδη από πριν ο buffer και το length στις δηλώσεις μεταβλητών, βάζουμε επιπλέον τον κώδικα καταγραφής και printk (με format **%lld**, ίδιο με πριν) , σε σημείο που αλλάζει το i_blocks, δηλαδή στη γραμμή 1507.

Τελειώσαμε με δομές για εγγραφή για **superblock**.

File Allocation Table

Η δομή του File Allocation Table , για τον fat-driver, είναι η struct fat_entry και βρίσκεται στη γραμμή 320 του fat.h

Ενδιαφέροντα πεδία που θα μπορούσαμε να τοποθετήσουμε στο journal είναι:

entry: Αυτό το πεδίο αναφέρεται στη θέση της καταχώρησης μέσα στον πίνακα ανάθεσης αρχείων (FAT). Θα μπορούσε να είναι χρήσιμο για την παρακολούθηση των αλλαγών στις εγγραφές FAT.

u.ent12_p , u.ent16_p , u.ent32_p: Αυτά τα πεδία περιέχουν δείκτες στις καταχωρήσεις FAT. Εάν υπάρχει μια αλλαγή στην τιμή αυτών των πεδίων, θα μπορούσε να υποδηλώσει

μια αλλαγή στην κατάσταση ενός block (για παράδειγμα, ένα block που προηγουμένως δεν ήταν διαθέσιμο τώρα έχει ανατεθεί σε ένα αρχείο). Θα δούμε για το **u.ent16_p**.

nr_bhs: Αυτό το πεδίο καθορίζει τον αριθμό των buffer heads που χρησιμοποιούνται για την ανάγνωση των καταχωρήσεων FAT. Εάν υπάρχει αλλαγή σε αυτό, μπορεί να σημαίνει ότι τα δεδομένα FAT αλλάζουν.

Θα κρατήσουμε μερικά από αυτά σε κάποιες συναρτήσεις.

Για το πεδίο **entry** της συνάρτησης **fat12 ent next** του αρχείου **fatent.c** ΠΡΟΣΟΧΗ

Γραμμή 270 προσθέτω struct super_block *sb = fatent->fat_inode->i_sb;

struct msdos_sb_info *sbi = MSDOS_SB(sb);

είναι τρόπος να λάβουμε το super block είναι μέσω του inode αντικειμένου που έχουμε στην κύρια δομή fat_entry, για χρήση journal. Διαδικασίες print στη γραμμή 281.

Για το πεδίο **entry** της συνάρτησης **fat16 ent next** του αρχείου **fatent.c** ΠΡΟΣΟΧΗ

Τα ίδια με πριν και print στη γραμμή 330.

Για το πεδίο **entry** της συνάρτησης **fat32 ent next** του αρχείου **fatent.c** ΠΡΟΣΟΧΗ

Τα ίδια με πριν και print στη γραμμή 369.

Για το πεδίο **u.ent16 p** της συνάρτησης **fat16 ent set ptr** του αρχείου **fatent.c** ΠΡΟΣΟ

Τα ίδια με πριν και print στη γραμμή 339.

Για το πεδίο **nr_bhs** της συνάρτησης **fat ent bread** του αρχείου **fatent.c**

Τα ίδια με πριν και print στη γραμμή 158.

Inode

Στη δομή struct msdos_inode_info στο fat.h

Κοιτώντας τα πεδία, αυτά που θα μπορούσαμε να κρατήσουμε για το journal είναι:

i_start: Αυτό το πεδίο αναφέρεται στον πρώτο cluster του αρχείου ή στο 0 εάν δεν υπάρχει αρχείο. Οι αλλαγές σε αυτό το πεδίο μπορεί να υποδεικνύουν νέα αρχεία ή αλλαγές στα υπάρχοντα αρχεία.

i_logstart: Αυτό το πεδίο αναφέρεται στο λογικό πρώτο cluster. Αυτό μπορεί να είναι χρήσιμο για την παρακολούθηση των αλλαγών στην εσωτερική δομή του αρχείου.

i_pos: Αυτό το πεδίο αναφέρεται στη θέση της καταχώρησης του αρχείου στο δίσκο ή 0 εάν δεν υπάρχει καταχώρηση. Αυτό μπορεί να είναι χρήσιμο για την παρακολούθηση των αλλαγών στην τοποθεσία των αρχείων στο δίσκο.

Για πεδία i_start και i_logstart στη συνάρτηση fat_free του file.c

Τα γράφουμε μαζί. Επίσης προσθέτουμε struct msdos_sb_info *sbi = MSDOS_SB(sb);, έτσι ώστε να έχουμε πρόσβαση στο object για χρήση journal.

Print στη γραμμή 327

Για πεδίο i_pos στη συνάρτηση fat_fill_inode του inode.c

Print στη γραμμή 568.

Directory entries

Δομή fat_slot_info του fat driver, στο fat.h .

Παρακάτω είναι τα πεδία που πιθανόν μπορούμε να καταγράψουμε:

slot_off: Η αρχή της καταχώρισης του καταλόγου ή της θυρίδας (slot) στον buffer. Αυτό θα μπορούσε να είναι χρήσιμο για την ανίχνευση προβλημάτων με τη διαχείριση των slots των καταχωρήσεων καταλόγου.

struct msdos_dir_entry: Από αυτό το struct μπορούμε να καταγράψουμε το **name:** που είναι το όνομα του αρχείου ή του καταλόγου. Αυτό θα μπορούσε να βοηθήσει στην αναγνώριση των επιμέρους στοιχείων που καταγράφονται στο journal.

Για το πεδίο name του msdos_dir_entry στη συνάρτηση fat_scan του dir.c

Προσθέτω `struct msdos_sb_info *sbi = MSDOS_SB(dir->i_sb);` , για το λόγο που αναφέραμε πριν και `print` στη γραμμή 972.

Για το πεδίο **slot off** της συνάρτησης `fat_add_entries` του `dir.c`

`Print` στη γραμμή 1410.

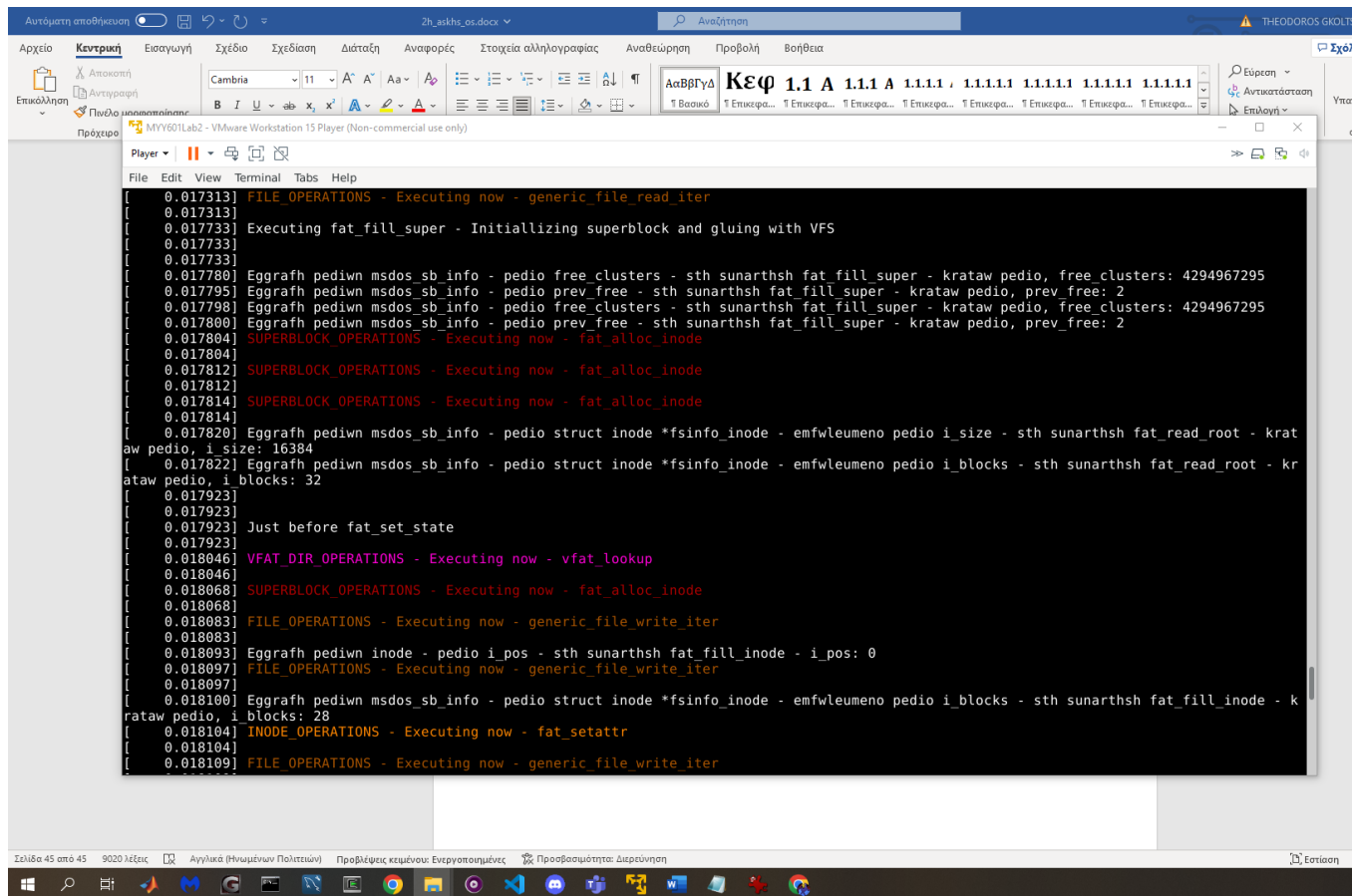
Έτσι κάνοντας `cptofs` βλέπουμε πολλά ενδιαφέροντα μηνύματα και τί τιμές έχουμε για το κάθε πεδίο που μας ενδιαφέρει.

Έγιναν πολλές προσπάθειες για προσπέλαση του `journal` μέσω

`mount -t vfat -o loop /tmp/vfatfile /vfat` και των υπόλοιπων βημάτων , αλλά δεν εμφανίζεται. Συμπεραίνουμε χρειάζεται να κάνουμε πιο περίπλοκες διαδικασίες για να φανεί.

Ακολουθούν screenshots, με τις τελευταίες λειτουργίες:

`./cptofs -i /tmp/vfatfile -p -t vfat lklfuse.c /`



./cptofs -i /tmp/vfatfile -p -t vfat teos_used_directory_experiment /

Αυτόματη αποθήκευση Zh_askhs_os.docx Αναζήτηση

Αρχείο **Κεντρική** Εισαγωγή Σχέδιο Σχεδίαση Διάταξη Αναφορές Στοιχεία αλληλογραφίας Αναθεώρηση Προβολή Βοήθεια

Επικόλληση Αποκοπή Αντιγραφή Πενόμοιο Μονοπούνισμα Πράγματο

Player

File Edit View Terminal Tabs Help

```
aw pedio, i size: 16384
[ 0.016014] Eggrafh pediwn msdos_sb_info - pedio struct inode *fsinfo_inode - emfwleumeno pedio i_blocks - sth sunarthsh fat_read_ro
ataw pedio, i blocks: 32
[ 0.016114]
[ 0.016114]
[ 0.016114] Just before fat_set_state
[ 0.016114]
[ 0.016227] VFAT_DIR_OPERATIONS - Executing now - vfat_lookup
[ 0.016227]
[ 0.016256] VFAT_DIR_OPERATIONS - Executing now - vfat_mkdir
[ 0.016256]
[ 0.016274] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat_ent_blocknr
[ 0.016274]
[ 0.016279] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat_ent_bread
[ 0.016279]
[ 0.016289] FILE_OPERATIONS - Executing now - generic_file_write_iter
[ 0.016289]
[ 0.016314] Eggrafh pediwn fat_entry - pedio nr_bhs - sth sunarthsh fat_ent_bread - krataw pedio, nr_bhs: 1
[ 0.016317] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_set_ptr
[ 0.016317]
[ 0.016319] FILE_OPERATIONS - Executing now - generic_file_write_iter
[ 0.016319]
[ 0.016322] Eggrafh pediwn fat_entry - pedio u.ent16_p - sth sunarthsh fat16_ent_set - entry: 1103104006
[ 0.016324] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
[ 0.016324]
[ 0.016326] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_next
[ 0.016326]
[ 0.016328] FILE_OPERATIONS - Executing now - generic_file_write_iter
[ 0.016328]
[ 0.016333] Eggrafh pediwn fat_entry - pedio entry - sth sunarthsh fat16_ent_next - entry: 4
[ 0.016335] FILE_OPERATIONS - Executing now - generic_file_write_iter
[ 0.016335]
[ 0.016337] Eggrafh pediwn fat_entry - u.ent16_p - sth sunarthsh fat16_ent_next - entry: 4
[ 0.016339] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_get
[ 0.016339]
[ 0.016340] FILE_ALLOCATION_ENTRIES_OPERATIONS - Executing now - fat16_ent_next
[ 0.016340]
[ 0.016340] εμφανίζεται. Συμπεραίνουμε χρειάζεται να κάνουμε πιο περίπλοκες οιαστώσεις για να
φραγεί.
```

Σελίδα 46 από 46 8 από 9028 λέξεις Αγγλικά (Ηνωμένων Πολιτειών) Προβλεψίες κειμένου: Ενεργοποιημένες Προσαρμοσμένη: Διερεύνηση