Универзитет у Београду
Електротехнички факултет

MirageFS

- Дипломски рад -

Ментор:
проф. др Зоран Јовановић

Кандидат:
Никола Кнежевић

Београд, септембар 2006.

Оцу, мајци и брату,
за сву пружену подршку...

Никола

Садржај

1 Увод **7**
1.1 Уводне напомене . . . 7
1.2 Преглед садржаја . . . 7

2 УМЛ **9**
2.1 Дизајн и имплементација . . . 9
2.1.1 Системски позиви . . . 9
2.1.2 Кернел мод и кориснички мод . . . 9
2.1.3 Промена контекста . . . 10
2.1.4 Организација меморије . . . 11
2.2 Употреба и предности . . . 11
2.3 Резиме . . . 13

3 VFS **14**
3.1 Објекти и методи . . . 14
3.2 Регистровање и монтирање фајл система . . . 16
3.3 Супер-блок . . . 16
3.4 Фајлови и операције . . . 19
3.5 Имена . . . 22
3.6 И-чворови . . . 25

4 MirageFS **27**
4.1 Монтирање фајл система и опције . . . 28
4.2 Сигурност . . . 28
4.3 Имплементација . . . 29
4.3.1 Операција `unlink` . . . 29
4.3.2 Операција `readdir` . . . 31
4.3.3 Операција `file_write` . . . 35

5 Резултати **38**

6 Резиме **40**

A Изворни кôд MirageFS фајл система **42**
A.1 Датотека miragefs_kern.c 42
A.2 Датотека miragefs_user.c 85
A.3 Датотека miragefs.h 94

Индекс слика

1 Меморијски простор УМЛ-а . . . 12
2 Меморијски простор УМЛ-а . . . 12
2 Декларације функција за регистровање и де-регистровање фајл система . . . 16
3 Структура struct file_system_type . . . 16
4 Структура struct super_block . . . 17
5 Структура struct super_operations . . . 18
6 Структура struct file . . . 19
7 Структура struct file_operations . . . 20
9 Пример изгледа фајл система . . . 22
10 Стање dcache структуре за пример са слике 9 . . . 23
11 Структура struct dentry_operations . . . 23
11 Пример мапе за убрзање приступа dentry објектима . . . 24
12 Структура struct inode_operations . . . 25
14 Монтирање MirageFS фајл система . . . 28
14 Имплементација unlink операције . . . 29
15 Имплементација readdir операције . . . 31
16 Имплементација file_write операције . . . 35

Индекс табела

1 Упоредни резултати 38

Поглавље 1

Увод

1.1 Уводне напомене

У овом раду је представљен фајл систем за УМЛ (*User-Mode-Linux*, виртуални оперативни систем за Линукс), којим се омогућава коришћење истог фајл система као и домаћин, уз копирај-по-упису (*Copy-on-Write*) технику приступа. Рад је настао као последица боравка аутора на стручној пракси на *EPFL*-у (Швајцарска). Аутор је боравио у Лабораторији за умрежене системе (*Networked Systems Lab - NSL*), од маја до септембра 2006. године. MirageFS је настао као део већег пројекта.

MirageFS се посебно издваја од осталих фајл система по је то што омогућава виртуалној машини да користи исти фајл систем као и код домаћина. На овај начин се може постићи исто окружење у оквиру виртуалног оперативног система као и у домаћину, што је посебно погодно за тестирање софтвера.

Имплементација фајл система је распоређена у три фајла (дата у Додатку), који укупно имају преко 3000 линија кôда. Уз текст су дати и исечци из овог кода, да би се приказала одређена функционалност. Линије кôда које нису могле да стану у један ред су означене са $\Rightarrow$ на самом крају реда.

1.2 Преглед садржаја

У поглављу 2 описује се УМЛ, јер фајл систем користи особине овог окружења. Описан је принцип рада ове виртуалне машине, као и интерфејс ка домаћину.

Поглавље 3 садржи детаљан опис Линуксовог виртуалног фајл система (*VFS - Virtual FileSystem*), интерфејса који сваки фајл систем мора да подржи, да би се могао користити са Линуксовим кернелом. Сем интерфејса, описане су и потребне структуре података и дат је детаљан опис системске структуре dcache, која служи за убрзано налажење потребних података.

MirageFS је детаљно описан у поглављу 4. Описана је његова двослојна структура, дат је преглед основних идеја и имплементациони детаљи.

У поглављу 5 су дати резултати и описано је како се MirageFS односи према осталим фајл системима. Показано је да он не доноси велико успорење, иако

користи копирај-по-упису технику.

Поглавље 6 доноси резиме рада и предлоге за даљу разраду, као и могућу употребу.

Допринос рада дат је у имплементацији новог фајл система са занимљивим својствима и широким спектром употребе, као и у опису Линуксовог виртуалног фајл система.

Поглавље 2

УМЛ

УМЛ (*UML, User Mode Linux*) је виртуални оперативни систем за Линукс (Linux), настао модификацијом Линуксовог кернела, тако да се може извршавати као обична апликација у постојећем Линукс окружењу. Потребне су и измене у домаћиновом кернелу, али како измене нису велике, све веће дистрибуције испоручују и овако модификовано језгро.

Једна од честих грешака је спомињање УМЛ-а као виртуалне машине. УМЛ је виртуални оперативни систем јер виртуализује системске позиве, а не хардвер (као што то раде виртуалне машине). Може се рећи да је УМЛ порт Линукса на себе, јер посматра Линукс као платформу на коју се сам може пренети, слично као код Интел (*Intel*) и Алфа (*Alpha*) платформи. Ипак, УМЛ нуди изолацију ресурса, јер апликације унутар УМЛ-а немају приступ домаћиновим реурсима, осим ако нису експлицитно наведени.

2.1 Дизајн и имплементација

2.1.1 Системски позиви

Оперативни систем је скуп структура података и процедура за баратање тим структурама. Сваки оперативни систем нуди исти интерфејс – ове приступне тачке се називају системски позиви. Системски позив је механизам којим апликација захтева сервис од оперативног система – кернела. Преко системског позива се преносе параметри из апликације у кернел и обратно. Линукс нуди преко 180 системских позива, од којих је 30 маркирано као застарело и више се не користи.

2.1.2 Кернел мод и кориснички мод

У оквиру сваког оперативног система, постоје два (или више) мода, тј. нивоа привилегија. У случају Линукса, постоје два мода – кернел мод и кориснички мод. Кернел мод има веће привилегије и у њему се извршава кернел. Кориснички мод нема директни приступ хардверу, нити меморији која припада кернелу. У њему се извршавају корисничке апликације. Обично

све хардверске платформе обезбеђују инструкције и механизме којима се врши промена из једног мода у други, као и одржавање нивоа привилегија.

Као што је већ речено, УМЛ посмата Линукс као хардверску платформу, али како Линукс не нуди наведене механизме својим процесима, користи се `ptrace` механизам за праћење системских позива. У УМЛ-у постоји специјална нит чији је главни задатак да извршава `ptrace` над свим корисничким процесима у УМЛ-у. Сви системски позиви процеса који се извршавају у корисничком моду се пресрећу од стране пратеће нити. Праћење се не врши за процесе који се извршавају у кернел моду. Ово је разлика између корисничког и кернел мода у УМЛ-у.

Пратећа нит врши транзицију између корисничког и кернел мода, тако што прати извршавање корисничког процеса. Када праћени процес изврши системски позив или прими сигнал, пратећа нит пребацује процес у кернел мод (ако је потребно) и наставља рад без праћења системских позива. Транзиција је реализована увођењем новог начина праћења у `ptrace` механизму - `PTRACE_SYSEMU`. Овај режим праћења омогућава пратиоцу да промени параметре системског позива пре самог извршења, или да га не изврши уопште. Када УМЛ „ухвати“ неки системски позив, анулира га у корисничком процесу, а сам УМЛ кернел извршава исти системски позив ка домаћиновом кернелу.

2.1.3 Промена контекста

УМЛ се као и стандардни кернел извршава у другом адресном подручју него корисничке апликације. Ова промена је уведена тек 2002. године и до тада су и УМЛ кернел и кориснички процеси делили исти адресни простор. Промена између модова рада је вршена предајом сигнала, који су се показали прилично спорим. Свакој нити у оквиру УМЛ одговарао је по један процес у домаћиновом кернелу, да би се одржавале све потребне структуре – мапирања меморије, руковаоци (handlers) сигналима и сл. Овакав вид организације је захтевао по 4 промене контекста, једно испоручивање сигнала и један пријем, по сваком системском позиву, што је уносило значајно успорење. Такође, процес који преузима контекст мора да скенира комплетан адресни простор и успостави нова мапирања, јер постоји могућност да је у међувремену дошло до промене дозвола на неким страницама, да су неке странице послате на диск или да су станице одмапиране.

Стога је 2002. године уведен SKAS (Separate Kernel Address Space, засебни адресни простор за кернел). У овој организацији система, укинуто је мапирање од нити у УМЛ-у ка процесима у домаћиновом кернелу и кернел је измештен у засебни адресни простор. Промена контекста је сведена на промену адресног простора, што је значајно бржа операција. Да би ово функционисало, уведене су и промене у домаћиновом кернелу – најзначајнији је `/proc/mm` интерфејс.

Ова промена је изведена тако што је омогућено да кориснички процеси могу да баратају адресним простором. Наиме, у кернелу постоји структура `mm_struct` у којој се чувају мапирања. Ова структура није увезана са процесом, тако да је извожење ове структуре у простор корисничких процеса једноставан задатак. Тренутна реализација интерфејса је изведена преко дескриптора фајлова (file descriptors). Операције над фајловима се мапирају на операције над адресним просторима. Тако, отварањем `/proc/self/mm` се добија податак (ручка) на адресни простор тренутног процеса. Промена адресног простора је изведена увођењем новог системског позива (у домаћиновом кернелу) – `switch_to`, који прихвата нови дескриптор и у складу с њим мења адресни простор. Уклањање адресног простора је једноставна операција затварања дескриптора.

Промена контекста (унутар УМЛ-а) је сада једноставна и брза операција, јер нема промене контекста са домаћиновим кернелом – потребно је изменити само адресни простор и поново успоставити стање регистара.

2.1.4 Организација меморије

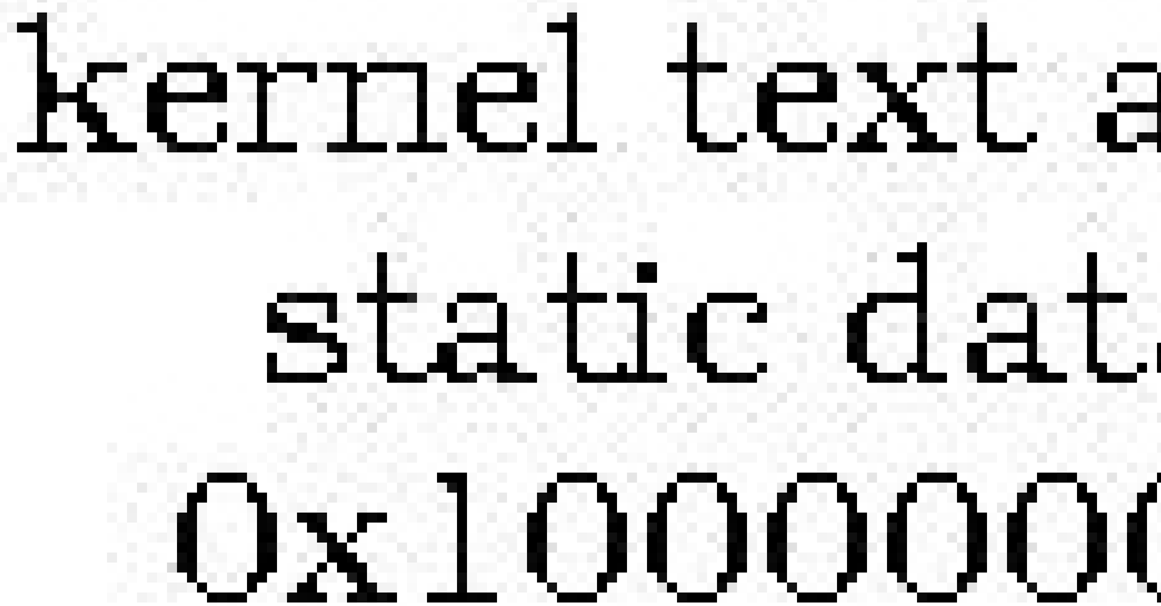
Наведене промене су довеле до тога да УМЛ доста личи на оригинални кернел, јер сада УМЛ може да користи комплетан простор за апликације (са гледишта домаћина) од 3GB. На слици 1 је приказан изглед меморије за УМЛ. Процеси заузимају исте опсеге адреса као што би заузимали и на домаћину. Стек процеса је померен за 0,5GB наниже, да би се ослободио простор за УМЛ кернел.

Виртуелна меморија је у УМЛ-у реализована коришћењем привремених фајлова на домаћину. Фајл је операцијом `mmap` мапиран у део „физичке“ меморије у адресном простору УМЛ-а. На слици 2 је приказана организација меморије. Када домаћинов кернел мапира физичке странице у виртуалне, УМЛ мапира одговарајући део фајла на одговарајућу локацију.

2.2 Употреба и предности

УМЛ доноси много предности, а најважније су:

- Лакше дебаговање кернела и развој.
- Дебаговање процера у када кернел ради нешто „необјашњиво“ – пролазак кроз део кернела ће брже открити проблем.
- Изолација - код за који се сумња да је лош, или да потиче од непроверених извора се може покретати без угрожавања осталих сервиса. Такође, могуће је лакше проверити нове верзије кернела, дистрибуција и сервиса.



Слика 1: Меморијски простор УМЛ-а

- Едукација - администација, показне вежбе из дизајна система и груписање сервиса је много лакше изводити на виртуалним машинама.
- Хостовање - могућност да више корисника дели једну физичку машину, а да свако покреће свој сервис независно од осталих.

2.3 Резиме

Као што је показано, УМЛ наликује стварном кернелу, тако да апликације не могу да открију под каквим се системом извршавају. Но, како је УМЛ кернел само апликација под правим кернелом, он може да позива и функције

Слика 2: Меморијски простор УМЛ-а

из стандардне библиотеке, што отвара нове могућности. Више о овоме ће бити речено у делу о имплементираном фајл систему.

Поглавље 3

VFS

VFS је фундаментални подсистем Линуксовог кернела, задужен за фајл системе. Скраћеница потиче од *Virtual File-System*, али се може срести и назив *Virtual Filesystem Switch*. Последњи термин најбоље осликава предност подсистема *VFS* – висок ниво апстракције у кернелу, којим се омогућава постојање различитих фајлистема истовремено, као и њихово коришћење. Као и сваки подсистем за фајл системе и *VFS* нуди корисничким програмима интерфејс ка фајл систему.

Основа овог подсистема су објекти (структуре), операције над објектима и генеричке методе. У имплементацији кôд личи на објектно-оријентисани и операције су представљене коришћењем показивача на функције. *VFS* функционише тако што се за сваку операцију иницирану од стране корисничког програма (преко системских позива) извршава нека од генеричких метода са одређеним објектом као параметром. Тек када је потребно, извршава се операција над објектом. Свака од операција је специфична за одређену имплементацију фајл система и придружује се објекту у време креирања. Овај начин рада подсећа на рад комутатора (*Switch*), што оправдава име подсистема.

3.1 Објекти и методи

Основни објекти у овом подсистему су фајлови, и-чворови (*inodes*), фајл системи* и имена и-чворова. У имплементацији за сваки објекат постоје две структуре – прва чува податке, а друга операције.

Фајлови представљају објекте из којих се подаци могу читати или уписивати. Сваки фајл је представљен структуром `struct file` а операције структуром `struct file_operations`, која чува показиваче на методе које се могу применити над фајлом. Фајл је најважнија структура коју користи кориснички програм.

*у овом раду се реч фајл систем користи да означи и класу фајл система (као што је *ext2*) и инстанцу (фајл систем монтиран на `/proc`)

И-чворови су основни објекти (јединице) фајл система. И-чвор може бити регуларни фајл, директоријум, симболички линк и сл. *VFS* не ради никакво разлучивање између различитих врста објеката, већ оставља свакој имплементацији да пружи одговарајуће методе за рад. Сваком и-чвору је придружен идентификациони број.

Иако фајлови и и-чворови личе, постоје битне разлике. У фајл систему могу постојати објекти који нису фајлови, али су и-чворови (симболички линкови). Такође, фајлови чувају стање (као што је нпр. позиција), док и-чвор чува само податке. Више фајлова може бити мапирано на један и-чвор. И-чвор је описан структуром `struct inode`, при чему се операције налазе у `struct inode_operations`.

Фајл системи су колекције и-чворова, од којих се један назива корени и има посебан третман. Фајл систем је објекат у оквиру *VFS*-а, што омогућава истовремено коришћење више различитих типова. И-чворовима једног фајлистема се приступа преко кореног и-чвора коришћењем придруженог имена. Фајл систем чува податке који су исти за све и-чворове фајл система (нпр. *READ-ONLY* начин приступа).

Сваки коришћени фајл систем у VFS подсистему је представљен објектом типа `struct super_block`. Како је могуће постојање различитих типова фајл система, сваки подржани тип је представљен објектом типа `struct file_system_type` која садржи метод `read_super`. Задатак овог метода је да створи одговарајући *super_ block*.

Имена се користе за приступ и-чвору. Проналажење одговарајућег и-чвора може бити јако скупа операција, па *VFS* користи кеш активних и скоро коришћених имена. Овај кеш се назива *dcache* и представљен је у меморији као дрво. Један и-чвор може бити повезан са више чворова у овом дрвету. Сваки чвор у дрвету се назива *dentry*.

Једна од корисних особина ове структуре (сем убрзане претраге) је и да она представља префикс стабла фајл система – ако се једно име налази у кешу и сви његови преци су у кешу.

dcache је веза између фајла и и-чвора, јер за сваки отворени фајл се чува одговарајући *dentry*, који показује на придружени и-чвор.

3.2 Регистровање и монтирање фајл система

Линукс кернел добија информацију о новим типовима фајл система позивом функције `register_filesystem`. Резултат ове функције је 0[†] ако је операција завршена, `-EINVAL` ако је параметар нула, или `-EBUSY` ако већ постоји фајл систем са истим именом. Декларација ове (и одговарајуће функције за дерегистрацију) је дата на слици 3.

```
1   #include <linux/fs.h>

    int register_filesystem(struct file_system_type * fs);
    int unregister_filesystem(struct file_system_type * fs);
```

Слика 3: "Декларације функција за регистровање и де-регистровање фајл система"

Регистрација фајл система треба да се обави или у методи `module_init` модула (ако је фајл систем имплементиран у модулу) или позивом функције `filesystem_setup`.

Структура `struct file_system_type` је приказана у наставку. Као што је већ речено, садржи показивач на `read_super` методу која треба да врати супер-блок – репрезент фајл система.

```
    struct file_system_type {
        const char *name;
        int fs_flags;
        struct super_block *(*read_super) (struct super_block ⇒
            *,
            void *, int);
6       struct file_system_type * next;
    };
```

Слика 4: "Структура `struct file_system_type`"

3.3 Супер-блок

Сваки регистровани и монтирани фајл систем је репрезентован структуром `struct super_block`. Информација да је фајл систем монтиран је представљена другом структуром – `struct vfsmount`. Ове структуре су увезане у листу на коју указује `vfsmountlist` и користе се за проналажење одговарајућег фајл

[†]за функције у кернелу је стандардно да враћају 0 у случају исправног завршетка и негативне вредности у случају грешке

система. Ова информација се такође налази и у `dcache` стаблу. Објекти (структуре) `struct super_block` и `struct vfsmount` су раздвојени, јер први објекат може да постоји и каде се уклони информација да је неки фајл систем монтиран. Исти фајл систем може бити монтиран и на више различитих места.

Изглед супер-блока и придружених операција је дат на сликама 5 и 6.

```
struct super_block {
    struct list_head        s_list;         /* Keep this ⇒
       first */
    kdev_t                  s_dev;
    unsigned long           s_blocksize;
    unsigned char           s_blocksize_bits;
    unsigned char           s_lock;
    unsigned char           s_dirt;
    struct file_system_type *s_type;
    struct super_operations *s_op;
    struct dquot_operations *dq_op;
    unsigned long           s_flags;
    unsigned long           s_magic;
    struct dentry           *s_root;
    wait_queue_head_t       s_wait;

    struct inode            *s_ibasket;
    short int               s_ibasket_count;
    short int               s_ibasket_max;
    struct list_head        s_dirty;        /* dirty inodes ⇒
       */
    struct list_head        s_files;

    union {
            /* Configured-in filesystems get entries here */
            void                    *generic_sbp;
    } u;
    /*
     * The next field is for VFS *only*. No filesystems have ⇒
         any business
     * even looking at it. You had been warned.
     */
    struct semaphore s_vfs_rename_sem;      /* Kludge */
};
```

Слика 5: "Структура `struct super_block`"

```
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);

    void (*read_inode) (struct inode *);

    void (*dirty_inode) (struct inode *);
    int (*write_inode) (struct inode *, int);
    void (*put_inode) (struct inode *);
    void (*drop_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    void (*write_super_lockfs) (struct super_block *);
    void (*unlockfs) (struct super_block *);
    int (*statfs) (struct super_block *, struct kstatfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);

    void (*sync_inodes) (struct super_block *sb,
                         struct writeback_control *wbc);
    int (*show_options)(struct seq_file *, struct vfsmount *) ⇒
       ;

    ssize_t (*quota_read)(struct super_block *, int, char *,
               size_t, loff_t);
    ssize_t (*quota_write)(struct super_block *, int,
               const char *, size_t, loff_t);
};
```

Слика 6: "Структура `struct super_operations`"

Неке од битнијих операција које су придружене супер-блоковима су:

`alloc_inode` Позива се када је потребно иницијализовати структуру и-чвора.

`destroy_inode` Позива се када је потребно ослободити ресурсе који су били алоцирани за и-чвор.

`read_inode` Позива се када је потребно прочитати неки и-чвор са диска.

`write_inode` Позива се за све и-чворове који су маркирани као прљави, од стране `mark_dirty_inode`. Маркирање је последица захтева за синхронизацијом фајла и и-чвора.

`put_inode` Када се и-чвор уклони из кеша и-чворова, позива се ова метода.

`drop_inode` Како се за сваки и-чвор чува број референци, овај метод се позива када више не постоји ни једна референца на и-чвор.

`delete_inode` Када VFS жели уклонити и-чвор, позива ова методу. Може се позвати и из `drop_inode`.

`statfs` Користи се у имплементацији системског позива `statfs (2)`. У случају да није дефинисана, системски позив ће вратити грешку.

3.4 Фајлови и операције

Фајлови су објекти фајл система над којима се врши читање и писање. Традиционална UNIX филозофија налаже да је све фајл, па стога и комуникација преко мреже се посматра као упис и читање из фајла. Фајлови су видљиви корисничким процесима преко дескриптора фајлова. Како у систему може бити присутно баферисање и кеширање, садржај фајла није увек једнак подацима на диску, тј. и-чвору‡. Фајл чува стање, као што се може видети на слици 7.

Примећује се да се за фајл не чува име, јер више различитих фајлова може бити повезано са једним и-чвором, тако да тај податак нема смисла. Уштеда је још један разлог зашто се фајловима не чува име, јер корисник који отвара фајл зна под којим именом га је отворио. Кернелу није потребна информација о имену фајла, сем код системског позива `open(2)`.

```
struct file {
    union {
        struct list_head     fu_list;
        struct rcu_head      fu_rcuhead;
    } f_u;
    struct dentry         *f_dentry;
    struct vfsmount             *f_vfsmnt;
    struct file_operations  *f_op;
    atomic_t             f_count;
    unsigned int             f_flags;
    mode_t               f_mode;
    loff_t               f_pos;
```

‡мада постоје и фајл системи код којих су сви и-чворови искључиво у меморији

```
    struct fown_struct  f_owner;
    unsigned int        f_uid, f_gid;
    struct file_ra_state    f_ra;

    unsigned long       f_version;
    void            *f_security;

    /* needed for tty driver, and maybe others */
    void            *private_data;

    struct address_space    *f_mapping;
};
```

Слика 7: "Структура `struct file`"

```
struct file_operations {
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, ⇒
       loff_t *);
    ssize_t (*aio_read) (struct kiocb *, char __user *, ⇒
       size_t, loff_t);
    ssize_t (*write) (struct file *, const char __user *, ⇒
       size_t,
            loff_t *);
    ssize_t (*aio_write) (struct kiocb *, const char __user ⇒
       *,
            size_t, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct ⇒
       poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int ⇒
       ,
            unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, ⇒
       unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, ⇒
       unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int ⇒
```

```
        datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (struct file *, const struct iovec *,
            unsigned long, loff_t *);
25  ssize_t (*writev) (struct file *, const struct iovec *,
            unsigned long, loff_t *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t,
            read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *, int,
            size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *,
            unsigned long, unsigned long, unsigned long, ⇒
               unsigned long);
    int (*check_flags)(int);
    int (*dir_notify)(struct file *filp, unsigned long arg);
35  int (*flock) (struct file *, int, struct file_lock *);
};
```

Слика 8: "Структура `struct file_operations`"

Структура `struct file_operations` садржи методе за рад са фајловима. Може се приметити у листингу на слици 8 да називи метода одговарају називима системских позива за фајлове. Заиста, ове методе се позивају у склопу истоименог системског позива. Најзначајније методе су:

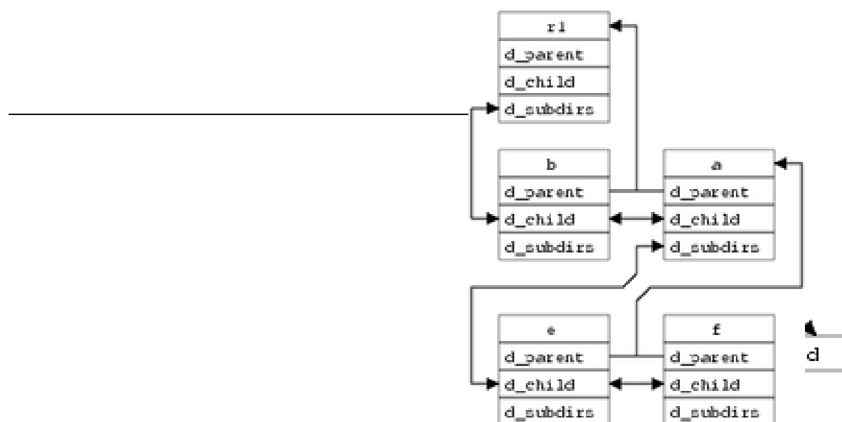
read Позива се када је потребно прочитати податке из фајла.

write Позива се када је потребно уписати нешто у фајл.

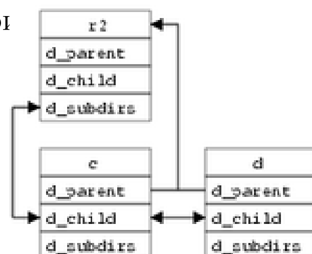
readdir У UNIX терминологији, све је фајл, па и директоријум, чији је садржај списак фајлова који му припадају. Овај метод се позива када је потребно прочитати садржај директоријума.

poll VFS позива овај метод када је потребно проверити да ли постоји нека активност над фајлом.

open Када је потребно отворити и-чвор, позива се овај метод, од стране VFS. Иако делује да овај метод припада операцијама над и-чворовима, постављен је међу операције над фајловима јер олакшава имплементацију фајл система. У овом методу се најчешће обавља иницијализација поља `private_data` (погледати листинг 7).



Слика 9: Пр ема



Слика 10: Стање `dcache` структуре за пример са слике 9

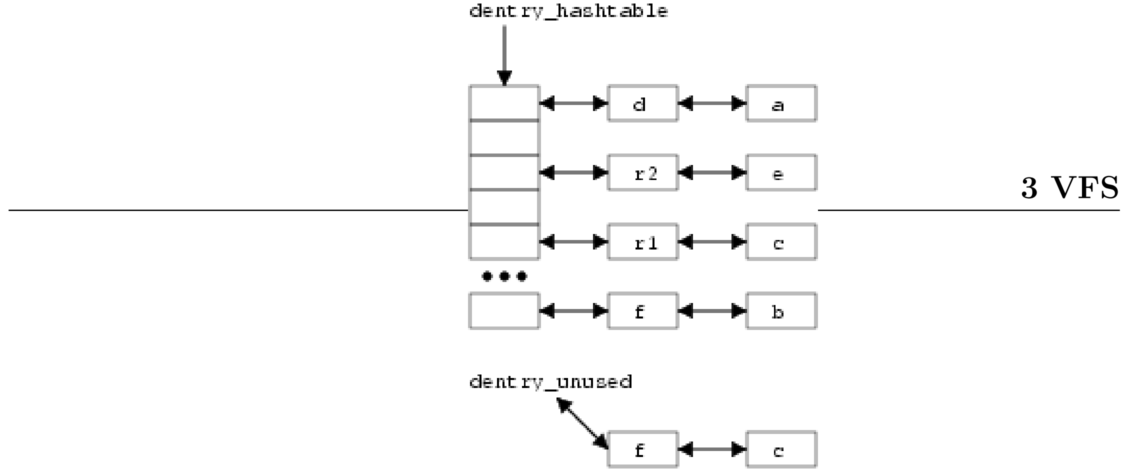
Као што је речено, фајлови су представа и-чвора са гледишта корисника. Ипак, операције над фајловима су имплементиране у фајл систему коме припада дати и-чвор.

3.5 Имена

`dcache` је јако битан део VFS подсистема, јер убрзава приступ и-чворовима. Убрзање не долази од бржег покретања механизма диска, бржег читања или уписа у фајл, већ од бржег налажења одговарајућег и-чвора за задату путању. Проналажење и-чвора се састоји од више фаза које се преплићу. Путања се разбија на компоненте и за сваку компоненту се испитује да ли је у `dcache` дрвету. Ако јесте, наставља се даље. Ако није, креира се одговарајући чвор у `dcache` стаблу и придружени и-чвор се учитава позивом одговајуће методе у родитељском и-чвору. Комплетна фаза се понавља за сваку компоненту и већи део је имплементиран у VFS методи `path_lookup`.

Однос између стања у фајл систему и `dcache` структуре је дат на сликама 9 и 10. `r1` и `r2` су корени и-чворови два фајлсистема, при чему је `r2` монтиран на директоријум `b`. Фајлу `g` се није приступало скоро, те се он не налази у `dcache` структури.

Да би се још више убрзала провера, сва имена су смештена и у мапу (hash table), која је придружена родитељском имену. Слика 11 приказује ову мапу. Приказана је и листа најређе скоро коришћених (*LRU - Least Recently Used*) имена. Сва имена из ове листе су најчешће и у мапи.



Слика 11: Пример мапе за убрзање приступа `dentry` објектима

Сав рад око путања и фајлова обавља VFS и конвертује компоненте у одговарајућа имена (`dentry`) у `dcache` стаблу, пре него што проследи податке даље у фајл систем. Једини изузетак су симболички линкови чији се садржај директно прослеђује фајл-систему.

`dcache` управља и кешом и-чворова, јер све док постоји `dentry` постојаће и и-чвор на који `dentry` указује.

```
struct dentry_operations {
    int (*d_revalidate)(struct dentry *, struct nameidata *);
    int (*d_hash) (struct dentry *, struct qstr *);
    int (*d_compare) (struct dentry *, struct qstr *, struct ⇒
       qstr *);
    int (*d_delete)(struct dentry *);
    void (*d_release)(struct dentry *);
    void (*d_iput)(struct dentry *, struct inode *);
};
```

Слика 12: "Структура `struct dentry_operations`"

У листингу 12 је приказан списак метода над једним `dentry` објектом.

d_revalidate Овај метод се позива када VFS треба да поново провери `dentry`. Уобичајено се позива када `lookup` нађе име у `dcache` стаблу. Већина фајл-система поставља ово поље на NULL јер је `dcache` конзистентан.

d_hash Позива се када је потребно додати објекат у мапу.

d_compare Служи за поређење два `dentry` објекта.

d_delete Позива се после уклањања последње референце на `dentry`. Овим се означава да нико више не користи `dentry`, али да је он и даље валидан.

d_release Овај метод се позива при деалоцирању.

d_iput Позива се пре претходног метода, а одмах пошто се уклони и-чвор повезан са овим објектом.

3.6 И-чворови

И-чвор представља објекат фајл система, јединицу смештања података. VFS чува у кешу активне и скоро коришћене и-чворове. И-чворовима се може приступити на два начина.

Први начин је преко `dcache` структуре, што је објашњено у претходној секцији. Други начин је преко мапе (хеш табеле). Кључ се креира од адресе супер-блока и броја и-чвора, репрезентован као 8-битна вредност. Сви и-чворови са истим кључем се увезују у листу. У овој листи се налазе и-чворови који нису повезани са објектима из `dcache` стабла.

И-чвор је велика структура која чува све што је потребно да би се описао податак са диска. У листингу 13 је дата стуктура `struct inode_operations`, јер су ове операције најбитније за рад са и-чворовима.

```
struct inode_operations {
2       int (*create) (struct inode *,struct dentry *,int, struct ⇒
            nameidata *);
        struct dentry * (*lookup) (struct inode *,struct dentry ⇒
           *, struct nameidata *);
        int (*link) (struct dentry *,struct inode *,struct dentry ⇒
            *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct inode *,struct dentry *,const char ⇒
            *);
        int (*mkdir) (struct inode *,struct dentry *,int);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,int,dev_t);
        int (*rename) (struct inode *, struct dentry *,
                struct inode *, struct dentry *);
12      int (*readlink) (struct dentry *, char __user *,int);
            void * (*follow_link) (struct dentry *, struct ⇒
                nameidata *);
            void (*put_link) (struct dentry *, struct nameidata ⇒
                *, void *);
        void (*truncate) (struct inode *);
        int (*permission) (struct inode *, int, struct nameidata ⇒
           *);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (struct vfsmount *mnt, struct dentry *, ⇒
           struct kstat *);
        int (*setxattr) (struct dentry *, const char *,const void ⇒
            *,size_t,int);
```

```
    ssize_t (*getxattr) (struct dentry *, const char *, void ⇒
        *, size_t);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
22  int (*removexattr) (struct dentry *, const char *);
};
```

Слика 13: "Структура `struct inode_operations`"

Све наведене методе се позивају без закључавања, осим ако није напоменуто. Најважније операције су:

create Позива се од стране `open(2)` и `creat(2)` системских позива. Ову операцију је потребно имплементирати ако се желе подржати регуларни фајлови. `dentry` који се прослеђује нема попуњено поље за и-чвор.

lookup VFS позива ову операцију када је потребно потражити и-чвор у родитељском и-чвору. Име које се тражи је дато у прослеђеном `dentry` објекту. Ако и-чвор не постоји, у `dentry` је потребно уметнути NULL и-чвор. Потребно је вратити код за грешку само ако се заиста догоди нека грешка (недостатак меморије, илегална секвенца карактера и сл.).

rename Истоимени системски позив користи ову операцију, којом се објекту чије је име (дато у другом параметру), мења у име дато у четвртом параметру. Првобитни родитељски и-чвор је указан првим параметром и објекат се премешта тако да му родитељски и-чвор постаје и-чвор указан четвртим параметром.

follow_link Ова операција се позива када VFS треба да isprati симболички линк. Повратна вредност се прослеђује методу `put_link()`.

put_link Позива се од стране VFS када је потребно ослободити ресурсе које је алоцирао `follow_link`

Поглавље 4

MirageFS

MirageFS је фајл систем који омогућава коришћење домаћиновог фајл система. Основа за израду MirageFS-а је *hostfs*. *hostfs* је дело аутора УМЛ-а Џефа Дајка (Jeff Dyke), који врши исту функцију. Оно што разликује између ова два фајл система је механизам копирај-по-писању. MirageFS користи скривени директоријум у који уписује све измене, чиме не прља домаћинов фајл систем. Коришћење додатног скривеног директоријума за смештање разлика и механизма копирај-по-упису је омогућило да се УМЛ подигне (*boot*) са истог фајл система као и домаћин и тиме створи истоветно окружење.

Идеја на којој се заснива MirageFS је да је УМЛ само процес у оквиру домаћина. Као процесу, доступни су му сви фајлови. MirageFS је у имплементацији подељен у два слоја – кернел слој и кориснички слој. Кернел слој имплементира VFS интерфејс и позива кориснички слој, који претвара позиве у одговарајуће позиве функција из `libc` библиотеке. Сва логика је смештена у кернел слој, а кориснички слој се понаша као једноставни драјвер за виртуелни диск. Ово је изведено да би се направила логичка сепарација слојева, јер кернел слој врши испитивање постојања фајлова у домаћиновом систему и извршава потребне операције над подацима у `dcache` стаблу.

Структура фајл система на диску такође има два слоја. Један слој одговара реалном фајл систему који види домаћин. Изнад овог слоја је слој специфичан за УМЛ и у њему су фајлови који су промењени од стране процеса из УМЛ-а, као и одговарајући мета-подаци (о избрисаним фајловима у домаћину и УМЛ-у). Наравно, други слој је доступан и домаћину као обичан директоријум. Управо ова двослојна организација чини кернел слој фајл система MirageFS прилично компликованим.

MirageFS користи неколико једноставних правила:

- Фајл који треба обрисати остаје у фајл систему, са наставком `.dead`.
- Провера се увек врши у горњем, другом слоју и ако се ништа не може наћи, приступа се нижем слоју.
- `dcache` чува информације о слоју које су кодиране у пољу `d_fsdata`

```
mount -t miragefs none /tmp -o /

./linux root=/dev/root rootfstype=miragefs rootflags=/
```

Слика 14: Монтирање MirageFS фајл система

- Подаци о и-чворовима који припадају нижем слоју можда не одговарају реалном стању, јер домаћин може да промени и-чвор без знања УМЛ-а. Провера се увек врши за ове чворове.

4.1 Монтирање фајл система и опције

MirageFS прихвата неколико опција, које се кернелу предају преко параметра `miragefs=`. Прва опција дефинише путању у домаћину која ће бити корени директоријум. На овај начин, свако монтирање из УМЛ ће бити ограничено на било који под-директоријум кореног директоријума. На овај начин, УМЛ се „затвара“ у дати директоријум.

Ако је као други параметар наведена кључна реч `append` свако уписивање ће надодавати на крај фајла, а не на било које место у фајлу.

Други параметар може да буде и `base=`*путања*, чиме се поставља директоријум који ће чувати податке вишег слоја. Ако се овај параметар не наведе, подразувена се смештање у под-директоријум `.uml` кореног директоријума.

На слици 14 приказан је начин монтирања. Прва команда ће у УМЛ-у монтирати домаћинов `/` (root) на `/tmp`. Друга команда се позива из домаћина и омогућава да се УМЛ подигне са домаћиновог root директоријума (`/`).

4.2 Сигурност

MirageFS се може искористити за извршавање DoS (*Denial of Service*) напада на домаћина, тако што се може препунити простор на диску. Овај напад се може избећи постављањем другог слоја на неки други диск, јер сви уписи се врше у овом слоју.

Други тип напада је ескалација привилегија, али само у случају да се MirageFS користи да би се УМЛ подигао са истог фајл система као и домаћин. Да би се ово постигло, потребно је покренути УМЛ као `root` корисник. Малициозни програм у УМЛ-у онда може преузети привилегије `root` корисника у УМЛ-у, а самим тим и под домаћином. Нажалост, не постоји одбрана од овог напада.

4.3 Имплементација

У овом поглављу ће бити описане неке операције. За разумевање кода је битно имати у виду да `dcache` унутар УМЛ-а не „види“ други слој. Структура стабла одговара структури коју види домаћин, тј. пролажењем од истоимених чворова би се добиле исте путање, иако та имена указују на различите и-чворове.

Све операције користе неколико помоћних метода:

get_dentry_name(struct dentry *dentry, int extra, int dofake)
: На основу имена датог у `dentry`, конструише путању. Параметром `extra` се дефинише број додатних карактера који ће бити остављен. Ако `dofake` има ненулту вредност, комплетној путањи ће бити додата путања до директоријума који чува податке другог нивоа.

dentry_name(struct dentry *dentry, int extra)
: Позив претходне функције, при чему је `dofake` постављен на 0.

get_inode_name(struct inode *ino, int extra, int fake)
: Слично као и функција `get_dentry_name`, али конструише име на основу параметра `ino`.

inode_name(struct inode *ino, int extra)
: Позив претходне функције, при чему је `dofake` постављен на 0.

get_inode_dentry_name(struct inode *, struct dentry *, int)
: Конструише име на основу `ino` као родитељског директоријума и имена датог у `dentry`.

4.3.1 Операција `unlink`

```
int miragefs_unlink(struct inode *ino, struct dentry *dentry)
```

Операција `unlink` се позива када је потребно уклонити ime на које указује `dentry` из директоријума одређеног са `ino`. Први корак је конструисање имена фајла у односу на домаћинов фајл систем, функцијом `get_inode_name`. MirageFS не уклања фајл са диска, већ га фајлу додаје екстензију `.dead` (екстензија се чува у глобалној промењивој `deadmarker`) и ако се фајл не налази у вишем нивоу, премешта га. У линији 20 је позив функције `duplicate_path_i`, која реплицира путању са домаћина у путању на другом нивоу. Ово је потребно урадити пре промене назива фајла.

Промену назива фајла врши `__miragefs_user_rename_file`[§]. Ова функција прихвата параметре и позива одговарајућу функцију библиотеке `libc`.

[§]све функције корисничког слоја почињу са `__miragefs_user_`

```
int miragefs_unlink(struct inode *ino, struct dentry *dentry)
{
    char *file;
    int err;
    int len;
    char *name;
    char *to, *from;

    if(append)
        return(-EPERM);

    file = get_inode_name(ino, dentry->d_name.len + 1 + 5, 1) ⇒
        ;
    if(file == NULL) return -ENOMEM;

    strcat(file, "/");
    len = strlen(file);
    strncat(file, dentry->d_name.name, dentry->d_name.len);
    file[len + dentry->d_name.len] = '\0';

    err = duplicate_path_i(ino);
    if (err)
            goto out_free;

    if ((err = __miragefs_user_file_type(file, NULL, NULL)) < ⇒
        0)
    {
        /* doesn't exists in .uml */
        strncat(file, deadmarker, 5);
        file[len+dentry->d_name.len+5] = '\0';

        err = -ENOMEM;
        name = get_inode_dentry_name(ino, dentry, 0);
        if (name == NULL)
            goto out_free;

        copy_file(name, file);
        kfree(name);
    }
    else
    {
        char *old = get_inode_dentry_name(ino, dentry, 1);
```

```
            err = -ENOMEM;
            if (old == NULL)
                goto out_free;

            file = strncat(file, deadmarker, 5);
47          __miragefs_user_rename_file(old, file);
            kfree(old);
        }

out:
        err = 0;

out_free:
        kfree(file);
        return err;
57 }
```

Слика 15: "Имплементација unlink операције"

4.3.2 Операција `readdir`

`int miragefs_readdir(struct file *file, void *ent, filldir_t filldir)`

Ово је најкомплекснија операција у MirageFS фајл систему, јер мора читати податке из два слоја. Задатак ове операције је да за сваки члан директоријума на који указује промењива `file` позове функцију на коју указује `filldir`.

Прво се читају објекти из другог, вишег нивоа, пазећи да се прескоче објекти који су означени као обрисани. Када се прође кроз други слој, прелази се на први. Операција `readdir` узима у обзир тренутну позицију у фајлу, тако да тренутна позиција може да одговара нечему што није из другог нивоа. Стога је искључиво за потребе ове операције, уведено ново поље у објекат `struct file` – `f_offset`. Ако овај офсет има ненулту вредност, то значи да `readdir` треба да прескочи читање објеката на другом нивоу, као што је назначено у реду 13 листинга на слици 16.

Код читања објеката из првог нивоа, проверава се да ли постоји истоимени објекат у другом нивоу, као и да ли је објекат означен као обрисан – и у том случају се прескаче. Дохватање објеката из оба нивоа се врши функцијом `__miragefs_user_read_dir`.

```
int miragefs_readdir(struct file *file, void *ent, filldir_t ⇒
    filldir)
```

```
{
3     void *dir;
      char *name, *fakename=NULL, *sname=NULL;
      unsigned long long ino;
      struct miragefs_readdir_file *rfi;
      int error, len, namelen, fakerr;
      int slen = 0;
      int no_fake = 0;
      loff_t next, next2;

      /* if we are looking for the other member of union, we ⇒
         must call this */
13    if (file->f_offset != 0 && file->f_offset < file->f_pos)
          goto ttofake;

      /* to this point, dcache should have all the info on non-⇒
         existing parents */
      name = get_dentry_name(file->f_dentry, 0, 1);
      if(name == NULL)
          return(-ENOMEM);
      dir = __miragefs_user_open_dir(name, &error);
      kfree(name);

23    if (dir == NULL && error == ENOENT)
      {
          /* no dir in fake tree */
          name = get_dentry_name(file->f_dentry, 0, 0);
          if (name == NULL)
              return -ENOMEM;
          dir = __miragefs_user_open_dir(name, &error);
          kfree(name);
          no_fake = 1;
      }
33    if (dir == NULL)
          return(-error);
      else
      {
          spin_lock(&file->f_dentry->d_lock);
          file->f_dentry->d_fsdata = file->f_dentry;
      }

      next = file->f_pos;
      next2 = file->f_pos;
```

```
43
    while((name = __miragefs_user_read_dir(dir, &next, &ino, ⇒
        &len)) != NULL){
        if (len > 5 && !strcmp(&name[len-5], deadmarker))
            goto skip;

        error = (*filldir)(ent, name, len, file->f_pos, ino, ⇒
            DT_UNKNOWN);
        if(error) break;
skip:
        file->f_pos = next;
    }
53  file->f_offset = next;
    __miragefs_user_close_dir(dir);
    if (error)
        goto out;
ttofake:
    if (no_fake)
        goto out_ok;

    name = get_dentry_name(file->f_dentry, 0, 0);
    if (name == NULL)
63      return -ENOMEM;
    dir = __miragefs_user_open_dir(name, &error);
    kfree(name);
    if (dir == NULL && error != ENOENT)
        return(-error);
    else if (dir == NULL)
        goto out_ok;

    next2 = file->f_pos - file->f_offset;
    while((sname = __miragefs_user_read_dir(dir, &next2, &ino ⇒
        , &slen)) != NULL){
73      /* skip . and .. */
        if ((slen == 1 && sname[0] == '.') || (slen == 2 && ⇒
            sname[0] == '.' && sname[1] == '.'))
            continue;

        fakerr = -ENOMEM;
        fakename = get_dentry_name(file->f_dentry, slen + 1 +⇒
            5, 1);
        namelen = strlen(fakename);
        strcat(fakename, "/");
```

```
        strcat(fakename, sname);
        if (fakename == NULL)
83      {
            error = fakerr;
            goto out;
        }
        fakerr = __miragefs_user_file_type(fakename, NULL, ⇒
           NULL);
        if (fakerr < 0 && fakerr != -ENOENT)
            kfree(fakename);
        else if (fakerr == -ENOENT)
        {
            fakename = strncat(fakename, deadmarker, 5);
93          fakerr = __miragefs_user_file_type(fakename, NULL⇒
               , NULL);
            kfree(fakename);
            if (fakerr > 0)
                continue;
        }
        else {
            kfree(fakename);
            continue;
        }

103     error = (*filldir)(ent, sname, slen, file->f_pos, ino ⇒
           , DT_UNKNOWN);
        if(error) break;
        file->f_pos = file->f_offset + next2;
    }
    __miragefs_user_close_dir(dir);
    if (error)
        goto out;

out_ok:
    error = 0;
113 out:
    return(error);
}
```

Слика 16: 'Имплементација `readdir` операције'

4.3.3 Операција `file_write`

```
ssize_t miragefs_file_write(struct file *file,
     const char *buf, size_t count, loff_t *ppos)
```

Суштина механизма копирај-по-писању је имплеметирана у овој операцији. Сем копирања фајла и репликације структуре директоријума у другом нивоу, потребно је променити податаке који се чувају у `dcache` стаблу. Име је можда од раније било кеширано и повезано са оригиналним и-чвором. Зато се у 88. линији кода закључава родитељски директоријум и мења показивач на и-чвор, доделом вредности новоствореног и-чвора.

```
static ssize_t miragefs_file_write(struct file *file, const ⇒
   char *buf, size_t count, loff_t *ppos)
{
    int ret, err = 0;
    struct inode *inode = file->f_dentry->d_inode;
5   struct inode *dir   = file->f_dentry->d_parent->d_inode;
    struct dentry *dentry = file->f_dentry;
    struct dentry *pos;
    struct file * cow_file;
    struct inode *new_inode = NULL;
    char *from, *to;
    int len, mode=0, r=0, w=0, fd;

    /* we need to check whether the file in question is ⇒
       stamped */
    if (dentry->d_fsdata != NULL && dentry->d_fsdata != ⇒
       dentry)
15  {
        from = get_dentry_name(dentry, 0, 0);
        if (from == NULL)
            return -ENOMEM;
        to = get_dentry_name(dentry, 5, 1);
        if (to == NULL)
        {
            kfree(from);
            return -ENOMEM;
        }
25      duplicate_path_i(dir);

        len = strlen(to);
        strncat(to, deadmarker, 5);
        err = __miragefs_user_file_type(to, NULL, NULL);
```

```
        if (err < 0 && err != -ENOENT)
        {
            kfree(to);
            return err;
        }
35      else if (err > 0 && err == OS_TYPE_DIR)
            __miragefs_user_do_rmdir(to);
        else if (err > 0 && (err == OS_TYPE_FILE && err == ⇒
           OS_TYPE_SYMLINK))
            __miragefs_user_unlink_file(to);

        err = copy_file(from, to);
        kfree(from);
        if (err)
        {
            kfree(to);
45          return err;
        }

        /* first, open inode */
        err = -ENOMEM;
        new_inode = iget(inode->i_sb, 0);
        if (new_inode == NULL)
        {
            kfree(to);
            return err;
55      }
        err = init_inode(new_inode, dentry);
        if (err)
        {
            iput(new_inode);
            kfree(to);
            return err;
        }
        err = read_name(new_inode, to);
        if (err)
65      {
            iput(new_inode);
            kfree(to);
        }

        MIRAGEFS_I(new_inode)->mode = MIRAGEFS_I(inode)->mode ⇒
            ;
```

```
        if(MIRAGEFS_I(inode)->mode & FMODE_READ)
            r = 1;
        if(MIRAGEFS_I(inode)->mode & FMODE_WRITE)
            w = 1;
        if(w)
            r = 1;

        fd = __miragefs_user_open_file(to, r, w, append);
        kfree(to);
        if(fd < 0)
        {
            return(fd);
        }
        MIRAGEFS_I(new_inode)->fd = fd;

        return(0);

        spin_lock(&dir->i_lock);
        list_for_each_entry(pos, &inode->i_dentry, d_alias) {
            pos->d_inode = new_inode;
            pos->d_fsdata = pos;
        }
        spin_unlock(&dir->i_lock);

        miragefs_delete_inode(inode);
        iput(inode);

    }

    return generic_file_write(file, buf, count, ppos);
}
```

Слика 17: 'Имплементација `file_write` операције'

Поглавље 5

Резултати

Примењени тестови су слични тестовима из тестног окружења МАБ (*Modified Andrew Benchmark*)[4]. Урађено је 6 различитих тестова и мерено је време извршавања операције¶. Сва мерења су вршена на директоријуму који садржи изворни кôд Линуксовог кернела, величине 253MB. Реузултати су приказани у табели 1. Мерења за *ReiserFS* су вршена у домаћину.

Табела 1: Упоредни резултати

	ReiserFS	MirageFS	hostfs
make	15m35.653s	31m19.607s	24m09.121s
make clean	0m7.174s	0m23.170s	0m12.493
cp	0m17.276s	1m19.462s	0m25.613s
grep	0m1.643s	0m8.424s	0m5.147s
du	0m0.097s	0m2.462s	0m0.272s
rm	0m1.322s	0m31.430s	0m2.097s

Први тест се састојао у покретању команде `make` и оптерећење одговара стандардном оптерећењу. Овај процес укључује доста копирања, брисања и стварања нових фајлова. Као што се може приметити, постоји успорење од 20% у односу на `hostfs` и скоро више од 100% у односу на *ReiserFS*. Оволико успорење се оправдава тиме што се тест извршава у оквиру виртуелне машине и укључује доста копирања. Велики део успорења је последица вишеструких провера и репликације структуре директоријума на други ниво.

Следећи тест се састојао од извршавања `make clean`, чиме се поништава претходна операција. Успорење у односу на домаћина од преко 400% је последица одлуке да се фајл у `MirageFS` фајл систему не брише, већ преименује. Успорење је последица провере да ли већ постоји фајл маркиран као обрисан и преименовање. Све промене се врше испитивањем домаћина, без провере података у `dcache` структури.

¶стандардном командом окружења – *time*

Трећи тест је био копирање комплетне структуре директоријума (253MB). Разлог оволиког успорења је искључиво у додатним проверама које се врше на првом нивоу.

Наредни тест је била претрага по фајловима за низ карактера који не постоје (чиме се пролази кроз све фајлове). Успорење у односу на `hostfs` и није велико, што води до закључка да је успорење последица провера које се врше у `readdir` операцији.

Пети тест је сличан претходном, с тим да се не врши читање комплетног фајла. `du` наредба враћа збирну величину свих фајлова у директоријуму, проласком кроз све фајлове. Стога, важе иста напомене као и у претходном случају.

Последњи тест је било брисање свих фајлова, које се обавља наизменичним позивањем операција `readdir` и `unlink`.

Поглавље 6

Резиме

У овом раду је описан MirageFS, фајл систем који омогућава УМЛ кернелу да користи исти фајл систем као и домаћин. Оно што посебно издваја MirageFS је могућност да се УМЛ подигне користећи исту конфигурацију као и домаћин, што је посебно погодно за потребе тестирања. Управо из ове потребе је и настао MirageFS. Аутор у свом свакодневном раду са УМЛ кернелом користи MirageFS.

Детаљно су изложене све битне компоненте MirageFS фајл система. Описани су принципи рада УМЛ-а и VFS интерфејса. Показане су перформансе MirageFS фајл система, које су упоређене је са фајл системом који је на домаћину и фајл системом `hostfs`. Може се уочити да успорења нису велика и скоро да су занемарљива у поређењу са јединим алтернативом.

Даља унапређења овог фајл система базирају се на повећању брзине додатним кеширањем података. Планирана је и имплементација сервиса у домаћину и УМЛ-у, којим би се УМЛ обавештавао о променама у домаћину и преузимао фајлове пре него што их домаћин измени. Применом ових сервиса, УМЛ би користио копију домаћиновог фајл система која је постојала у тренутку покретања УМЛ-а.

Литература

[1] Michael Beck, Harold Bohme, Ulrich Kunitz, Robert Magnus, Mirko Dziadzka, and Dirk Verworner. *Linux Kernel Internals*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[2] Jeff Dyke. User mode linux. www site, `http://user-mode-linux.sourceforge.net`.

[3] Richard Gooch. Overview of the virtual file system. www site, `http://www.atnf.csiro.au/people/rgooch/linux/docs/vfs.txt`, 1999.

[4] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[5] Paul McKenney. Scaling dcache with rcu. www site, `linuxjournal.com/article/7124`, November 2004.

[6] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, E. Zadok, and M. N. Zubair. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-01b, Computer Science Department, Stony Brook University, October 2004. `www.fsl.cs.sunysb.edu/docs/unionfs-tr/unionfs.pdf`.

Додатак A

Изворни кôд MirageFS фајл система

A.1 Датотека miragefs_kern.c

miragefskern.c.tex

A.2 Датотека miragefs_user.c

miragefsuser.c.tex

A.3 Датотека miragefs.h

miragefs.h.tex