Laborator 10 - Drivere de sisteme de fişiere (Linux) partea 2

Obiectivele laboratorului

Cuvinte cheie

- VFS
- inode
- dentry

Materiale ajutătoare

- Slide-uri de suport pentru laborator
- SO2 Reference Card

Inode-ul

Inode-ul este o componentă esenţială a unui sistem de fişiere UNIX (ext3, reiserfs) şi, în acelaşi timp, o componentă importantă a VFS. Un inode este o metadată (deţine informaţii despre informaţii). Un inode identifică în mod unic un fişier de pe disc şi deţine informaţii despre acesta (uid, gid, drepturi de acces, timpi de acces, pointeri la blocurile de date etc.). Un aspect important este faptul că un inode nu deţine informaţii despre numele fişierului (acesta este reţinut de structura struct dentry asociată).

Inode-ul referă un fişier de pe disc. Pentru referirea unui fişier deschis (asociat cu un descriptor de fişier din cadrul unui proces) se foloseşte structura struct file. Unui inode îi corespund zero sau mai multe structuri `file` (mai multe procese pot deschide acelaşi fişier, sau un proces poate deschide acelaşi fişier de mai multe ori).

Inode-ul există atât ca entitate VFS (în memorie), cât şi ca entitate pe disc (pentru sistemele de fişiere UNIX, HFS, NTFS etc.). Inode-ul din VFS este reprezentat de structura struct inode. Ca şi celelalte structuri din VFS, struct inode este o structură generică care acoperă opţiunile pentru toate tipurile de fişiere suportate, chiar şi acele sisteme de fişiere care nu au o entitate pe disc asociată (cum este FAT).

Structura inode

Structura struct inode este unică pentru toate sistemele de fişiere. În general sistemele de fişierele deţin şi informaţii particulare. Acestea sunt referite prin intermediul câmpului `i_private` al structurii. Convenţional, structura care păstrează acele informaţii particulare este denumită `fsname_inode_info`, unde `fsname` reprezintă numele sistemului de fişiere. Spre exemplu, sistemele de fişiere `minix` şi `ext3` păstrează informaţiile particulare în structurile struct minix_inode_info, respectiv struct ext3_inode_info.

Câteva din câmpurile importante ale structurii struct inode sunt:

- `i_sb`: superblocul sistemului de fişiere de care aparţine acest inode
- `i_rdev`: dispozitivul pe care este montat acest sistem de fişiere
- `i_ino`: numărul inode-ului (identifică în mod unic inode-ul în cadrul sistemului de fişiere)
- `i_blkbits`: log2(dimensiunea blocului)
- `i_mode`, `i_uid`, `i_gid`: drepturile de access, uid-ul, gid-ul
- `i_size`: dimensiunea fişierului/directorului/etc. în octeţi
- `i_mtime`, `i_atime`, `i_ctime`: timpul de modificare, access şi creare
- `i_nlink`: numărul de intrări de nume care folosesc acest inode; pentru sistemele de fişiere fără link-uri (fie ele hard sau simbolice) este întotdeauna setat pe 1
- `i_blocks`: numărul de blocuri folosite de fişier (toate blocurile, nu doar cele de date); sunt folosite doar de subsistemul de quota
- `i_op`, `i_fop`: pointeri către operaţiile: struct inode_operations şi struct file_operations; `i_mapping->a_ops` conţine

pointer-ul către operaţiile struct address_space_operations
- `i_count`: contorul inode-ului care indică de câte componente kernel este folosit

Câteva funcţii care pot fi utilizate în lucrul cu inode-uri:

- new_inode: creează un nou inode, setează câmpul `i_nlink` pe 1 şi iniţializează câmpurile `i_blkbits`, `i_sb` şi `i_dev`;
- insert_inode_hash: adaugă inode-ul la tabela hash de inode-uri; un efect interesant al acestui apel este că inode-ul va fi scris pe disc dacă este marcat dirty; **ATENŢIE!** un inode creat cu `new_inode()` nu este în hash table, şi în afară de cazul în care aveţi motive serioase, trebuie să îl introduceţi în hash table;
- mark_inode_dirty: marchează inode-ul dirty; la un moment ulterior el va fi scris pe disc;
- iget_locked: încarcă inode-ul cu numărul dat de pe disc, dacă nu este deja încărcat;
- unlock_new_inode: folosit în conjuncţie cu iget_locked, eliberează lock-ul de pe inode;
- iput: anunţă kernel-ul că s-a terminat de lucrat asupra inode-ului; dacă nu îl mai foloseşte nimeni el va fi distrus (după ce va fi scris pe disc dacă este dirty);
- make_bad_inode: anunţă kernel-ul că respectivul inode nu poate fi folosit; în general se foloseşte din funcţia care citeşte inode-ul atunci când nu s-a putut citi inode-ul de pe disc, el fiind invalid.

Operaţii cu inode-uri

Obţinerea unui inode

Una din principalele operaţii cu inode-uri este obţinerea unui inode (a unei structuri struct inode în VFS). Până la versiunea `2.6.24` a nucleului Linux, dezvoltatorul definea o funcţie `read_inode`. Începând cu versiunea 2.6.25, dezvoltatorul trebuie să definească o funcţie `fsname_iget`. Această funcţie este responsabilă cu aflarea inode-ului VFS în cazul în care acesta există sau crearea unui inode nou şi completarea acestuia cu informaţiile de pe disc.

În general, în cadrul funcţiei se va apela iget_locked pentru obţinerea structurii struct inode din VFS. În cazul în care inode-ul este nou creat, atunci va trebui citit inode-ul de pe disc (folosind sb_bread) şi completate informaţiile utile.

Un exemplu de astfel de funcţie este minix_iget:

```
static struct inode *V1_minix_iget(struct inode *inode)
{
        struct buffer_head * bh;
        struct minix_inode * raw_inode;
        struct minix_inode_info *minix_inode = minix_i(inode);
        int i;

        raw_inode = minix_V1_raw_inode(inode->i_sb, inode->i_ino, &bh);
        if (!raw_inode) {
                iget_failed(inode);
                return ERR_PTR(-EIO);
        ...
}

struct inode *minix_iget(struct super_block *sb, unsigned long ino)
{
        struct inode *inode;

        inode = iget_locked(sb, ino);
        if (!inode)
                return ERR_PTR(-ENOMEM);
        if (!(inode->i_state & I_NEW))
                return inode;

        if (INODE_VERSION(inode) == MINIX_V1)
                return V1_minix_iget(inode);
    ...
}
```

În cadrul funcţiei minix_iget se obţine inode-ul VFS folosind iget_locked. Dacă inode-ul este deja existent (nu este nou = nu este configurat flag-ul I_NEW) funcţia se întoarce. Altfel, se apelează funcţia V1_minix_iget care va citi inode-ul de pe disc folosind minix_V1_raw_inode şi apoi va completa inode-ul VFS cu informaţiile citite.

Superoperaţii

O bună parte din superoperaţii (componentele structurii struct super_operations, utilizate de superbloc) sunt folosite în lucrul cu inode-uri. În continuare sunt prezentate acestea:

- `alloc_inode`: alocă un inode. De obicei, în cadrul acestei structuri se alocă o structură `fsname_inode_info` şi se efectuează iniţializările de bază ale inode-ului VFS (folosind inode_init_once). În minix, pentru alocare este folosită funcţia kmem_cache_alloc care interacţionează cu subsistemul SLAB. La fiecare alocare se apelează constructorul cache-ului, în cazul minix funcţia init_once. Alternativ, se poate folosi kmalloc, caz în care va trebui apelată funcţia inode_init_once. Funcţia `alloc_inode` va fi apelată de funcţiile new_inode şi iget_locked.
- `write_inode`: salvează/actualizează pe disc inode-ul primit ca parametru; pentru actualizarea inode-ului, deşi ineficient, pentru începători este recomandat să folosească următoarea secvenţă de operaţii:
  - încarcă inode-ul de pe disc cu ajutorul funcţiei sb_bread;
  - modifică buffer-ul în concordanţă cu inode-ul de salvat;
  - marchează buffer-ul murdar (dirty) cu ajutorul funcţiei mark_buffer_dirty; kernel-ul se va ocupa apoi de scrierea lui pe disc;
  - un exemplu este funcţia minix_write_inode din sistemul de fişiere minix.
- `clear_inode`: eliberează resurse asociate unui inode. Este apelată de fiecare dată când kernel-ul a terminat de lucrat cu inode-ul şi urmează să îl distrugă.
- `delete_inode`: şterge de pe disc şi din memorie orice informaţie referitoare la numărul inode-ului primit în câmpul `i_ino` (atât inode-ul de pe disc cât şi blocurile de date asociate). Aceasta implică realizarea următoarelor operaţii:
  - şterge inode-ul de pe disc;
  - actualizează hărţile de biţi pe disc (în cazul în care există)
  - şterge inode-ul din page cache prin apelarea funcţiei truncate_inode_pages;
  - şterge inode-ul din memorie prin apelarea funcţiei `clear_inode` ;
  - un exemplu este funcţia minix_delete_inode din sistemul de fişiere minix.
- `destroy_inode` eliberează memoria ocupată de inode

Operaţii pentru inode - inode_operations

Operaţiile pentru inode sunt descrise de structura struct inode_operations.

Inode-urile sunt de mai multe tipuri: fişier, director, fişier special (pipe, fifo), dispozitiv de tip bloc, dispozitiv de tip caracter, link etc. Din acest motiv, operaţiile pe care trebuie un inode să le implementeze sunt diferite pentru fiecare tip de inode. Mai jos vor fi prezentate în detaliu operaţiile implementate pentru un inode de tip fişier şi un inode de tip director.

Operaţiile unui inode sunt iniţializate şi accesate folosind câmpul `i_op` al structurii struct inode.

Structura file

Structura `file` corespunde unui fişier deschis de un proces şi există doar în memorie, fiind asociată unui inode. Este entitatea din VFS cea mai apropiată de user-space; câmpurile structurii conţin informaţii familiare ale unui fişier din user-space (modul de acces, poziţia în fişier, etc.), iar operaţiile cu aceasta sunt apeluri de sistem cunoscute (`read`, `write`, etc.).

Operaţiile pentru `file` sunt descrise de structura struct file_operations.

Pentru a iniţializa operaţiile pe `file` pentru un sistem de fişiere, se foloseşte câmpul `i_fop` al structurii struct inode. La deschiderea unui fişier, VFS-ul iniţializează câmpul `f_op` al structurii struct file cu adresa din `inode->i_fop`, astfel încât apeluri de sistem ulterioare să folosească valoarea stocată în `file->f_op`.

Inode-urile de tip fişier

Pentru lucrul cu inode-ul trebuie completate câmpurile `i_op` şi `i_fop` ale structurii inode. Tipul inode-ului determină operaţiile pe care trebuie să le implementeze.

Operaţii asupra inode-urilor de tip fişier

În sistemul de fişiere minix, pentru operaţiile pe un inode este definită structura minix_file_inode_operations, iar pentru operaţiile pe `file` se defineşte structura minix_file_operations:

```
const struct file_operations minix_file_operations = {
        .llseek         = generic_file_llseek,
        .read           = do_sync_read,
        //...
        .write          = do_sync_write,
        //...
        .mmap           = generic_file_mmap,
```

```
        //...
};

const struct inode_operations minix_file_inode_operations = {
        .truncate       = minix_truncate,
        //...
};

        //...
        if (S_ISREG(inode->i_mode)) {
                inode->i_op = &minix_file_inode_operations;
                inode->i_fop = &minix_file_operations;
        }
        //...
```

Funcţiile generic_file_llseek, generic_file_mmap, do_sync_read şi do_sync_write sunt implementate în kernel.

Pentru sistemele de fişiere simple nu trebuie să se implementeze decât funcţia de trunchiere (`truncate`), care primeşte ca parametru un inode în care câmpul `i_size` a fost modificat astfel încât să indice noua dimensiune a fişierului. Acţiunile de executat de această funcţie sunt:

- să dezaloce blocurile de date de pe disc care acum sunt în plus şi să actualizeze hărţile de biţi pe disc (în cazul în care sunt folosite);
- să marcheze acest lucru în inode;
- să completeze cu zero spaţiul care a rămas nefolosit din ultimul bloc cu ajutorul funcţiei block_truncate_page.

Un exemplu este funcţia minix_truncate din sistemul de fişiere minix.

ATENŢIE! Funcţiile `generic_*`, `do_sync_read`, `do_sync_write` sunt deja implementate!

Operaţii asupra spaţiului de adresă

Între spaţiul de adrese al unui proces şi fişiere există o strânsă legătură: execuţia programelor se face aproape exclusiv prin maparea fişierului în spaţiul de adresă al procesului. Întrucât această abordare funcţionează foarte bine şi este destul de generală, poate fi folosită şi în cazul apelurilor de sistem obişnuite cum ar fi `read` şi `write`.

Structura care descrie spaţiul de adresă este struct address_space , iar operaţiile cu acesta sunt descrise de structura struct address_space_operations. Pentru iniţializarea operaţiilor asupra spaţiului de adresă, se completează câmpul `inode->i_mapping->a_ops` al inode-ului de tip fişier.

Un exemplu este structura minix_aops din sistemul de fişiere minix:

```
static const struct address_space_operations minix_aops = {
        .readpage = minix_readpage,
        .writepage = minix_writepage,
        .sync_page = block_sync_page,
        .write_begin = minix_write_begin,
        .write_end = generic_write_end,
        .bmap = minix_bmap
};

//...
if (S_ISREG(inode->i_mode)) {
        inode->i_mapping->a_ops = &minix_aops;
}
//...
```

Funcţia block_sync_page este deja implementată. Majoritatea funcţiilor specifice sunt foarte uşor de implementat, după cum urmează:

```
static int minix_writepage(struct page *page, struct writeback_control *wbc)
{
        return block_write_full_page(page, minix_get_block, wbc);
}

static int minix_readpage(struct file *file, struct page *page)
{
        return block_read_full_page(page,minix_get_block);
}

static int minix_write_begin(struct file *file, struct address_space *mapping,
                        loff_t pos, unsigned len, unsigned flags,
                        struct page **pagep, void **fsdata)
{
```

```
        *pagep = NULL;
        return block_write_begin(file, mapping, pos, len, flags, pagep, fsdata,
                                minix_get_block);
}

static sector_t minix_bmap(struct address_space *mapping, sector_t block)
{
        return generic_block_bmap(mapping,block,minix_get_block);
}
```

Tot ce mai trebuie făcut este să se implementeze minix_get_block, care trebuie să translateze un bloc al unui fişier într-un bloc de pe device. Dacă flag-ul `create` primit ca parametru este activat, trebuie alocat un nou bloc. În cazul în care se creează un bloc nou, trebuie marcată corespunzător harta de biţi. Pentru a înştiinţa nucleul să nu mai citească blocul de pe disc, trebuie marcat `bh` cu set_buffer_new. Trebuie asociat buffer-ul cu blocul cerut prin funcţia map_bh.

Structura dentry

Operaţiile pe directoare folosesc structura struct dentry. Principala sarcină a acesteia este realizarea de legături între inode-uri şi numele fişierelor. Câmpurile importante ale acestei structuri sunt prezentate mai jos:

```
struct dentry {
        //...
        struct inode            *d_inode;     /* associated inode */
        //...
        struct dentry           *d_parent;    /* dentry object of parent */
        struct qstr             d_name;       /* dentry name */
        //...

        struct dentry_operations *d_op;       /* dentry operations table */
        struct super_block      *d_sb;        /* superblock of file */
        void                    *d_fsdata;    /* filesystem-specific data */
        //...
};
```

Semnificaţiile câmpurilor:

- `d_inode`: inode-ul referit de acest dentry;
- `d_parent`: dentry-ul asociat directorului părinte;
- `d_name`: o structură de tip struct qstr ce conţine câmpurile `name` şi `len` cu numele, respectiv lungimea numelui;
- `d_op`: operaţii cu dentry-uri, reprezentate printr-o structură struct dentry_operations. Nucleul implementează operaţii implicite, astfel încât nu este nevoie să fie (re)implementate. Unele sisteme de fişiere pot face optimizări legate de structura specifică a dentry-urilor;
- `d_fsdata`: câmp rezervat sistemului de fişiere ce implementează operaţii dentry;

Operaţii cu dentry

Operaţiile care se aplică cel mai adesea asupra dentry-urilor sunt:

- d_alloc_root: alocă dentry-ul rădăcină. Se foloseşte în general în funcţia care este apelată pentru citirea superblocului (`fill_super`), care trebuie să iniţializeze directorul rădăcină. Aşadar se obţine inode-ul rădăcină din superbloc şi se foloseşte ca argument pentru această funcţie, pentru a completa câmpul `s_root` din structura struct super_block;
- d_add: asociază un dentry unui inode; dentry-ul primit ca parametru în apelurile discutate mai sus, semnifică intrarea (nume, lungime nume) ce trebuie să fie creată. Se va folosi această funcţie atunci când se creează / încarcă un nou inode care nu are asociat un dentry şi care nu a fost introdus încă în hash (la `lookup`);
- d_instantiate: versiunea mai light a apelului precedent, în care dentry-ul a fost în prealabil introdus în hash.

ATENŢIE! Trebuie să se folosească `d_instantiate` şi NU `d_add` pentru apelurile `create`, `mkdir`, `mknod`, `rename`, `symlink`.

Operaţii asupra inode-urilor de tip director

Operaţiile de lucru cu inode-urile de tip director au un nivel de complexitate mai ridicat decât cele de tip fişier. Dezvoltatorul trebuie să definească operaţii pentru inode-uri şi operaţii pentru file-uri. În minix, aceste operaţii sunt definite în structurile minix_dir_inode_operations, respectiv minix_dir_operations:

```
struct inode_operations minix_dir_inode_operations = {
      create: minix_create,
```

```
      lookup: minix_lookup,
      link: minix_link,
      unlink: minix_unlink,
      symlink: minix_symlink,
      mkdir: minix_mkdir,
      rmdir: minix_rmdir,
      mknod: minix_mknod,
      //...
};

struct file_operations minix_dir_operations = {
      read: generic_read_dir,
      readdir: minix_readdir,
      //...
};

        //...
        if (S_ISDIR(inode->i_mode)) {
                inode->i_op = &minix_dir_inode_operations;
                inode->i_fop = &minix_dir_operations;
                inode->i_mapping->a_ops = &minix_aops;
        }
       //...
```

Singura funcţie deja implementată este generic_read_dir.

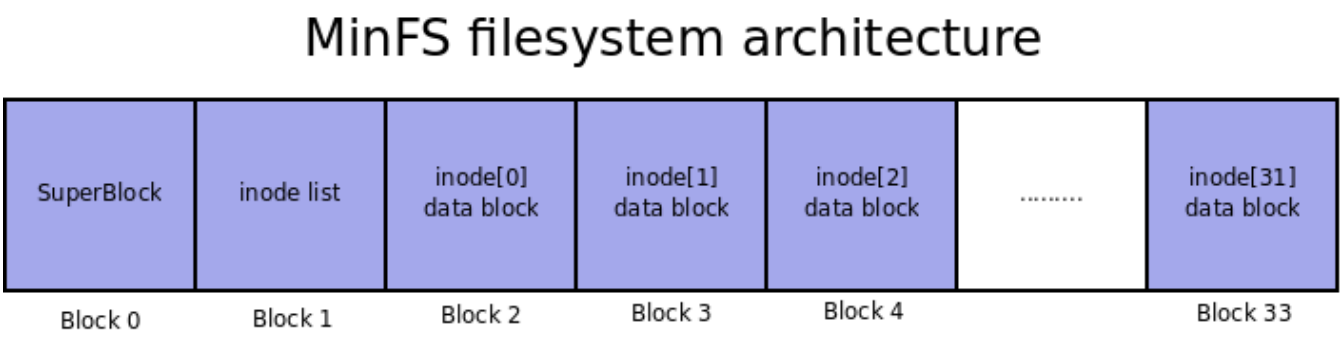
Funcţiile care implementează operaţiile asupra inode-urilor de tip director sunt:

- minix_create: creează un nou inode. Se apelează în urma apelurilor de sistem `creat` şi `open`. Trebuie să realizeze următoarele operaţii:
  - introduce în structura fizică a discului o nouă intrare; nu trebuie uitată actualizarea hărţilor de biţi pe disc;
  - se configurează drepturile de acces la cele primite ca parametru;
  - se setează inode-ul ca murdar (dirty) cu ajutorul funcţiei mark_inode_dirty;
  - se instanţiază intrarea de director cu ajutorul funcţiei d_instantiate;
- minix_mkdir creează un nou director. Se apelează în urma apelului de sistem `mkdir`. Trebuie să realizeze următoarele operaţii:
  - minix_create
  - alocă un bloc de date
  - introduce intrările "." şi ".."
- minix_link: creează un link hard; se apelează în urma apelului de sistem `link`; trebuie să realizeze următoarele operaţii:
  - leagă dentry-ul nou la inode
  - incrementează câmpul `i_nlink` la inode
  - se setează inode-ul ca murdar (dirty) cu ajutorul funcţiei mark_inode_dirty
- minix_symlink: creează un link simbolic; se apelează în urma apelului de sistem `symlink`; operaţiile care trebuiesc realizate sunt similare cu cele de la minix_link cu deosebirile date de faptul că se creează o legătură simbolică;
- minix_unlink: şterge un inode; se apelează în urma apelului de sistem `unlink`; trebuie să realizeze următoarele operaţii:
  - şterge din structura fizică a discului intrarea dată ca parametru
  - decrementează contorul `i_nlink` al inode-ului către care puncta intrarea (altfel inode-ul nu va fi niciodată şters)
- minix_rmdir: şterge un director; se apelează în urma apelului de sistem `rmdir`; trebuie să realizeze următoarele operaţii:
  - operaţiile de la `minix_unlink`;
  - se asigură că directorul este gol; în caz contrar, se întoarce `ENOTEMPTY`;
  - şterge şi blocul/blocurile de date asociat(e);
- minix_lookup: caută un inode într-un director; trebuie să realizeze următoarele operaţii:
  - caută în directorul indicat în `dir` intrarea cu numele `dentry->d_name.name`;
  - în cazul în care intrarea este găsita se va întoarce `NULL` şi se va asocia inode-ul cu numele, cu ajutorul funcţiei d_add;
  - în caz contrar, se întoarce ERR_PTR;
- minix_readdir: listează intrările dintr-un director începând de la poziţia specificată. Se apelează în urma apelului de sistem `readdir`. Funcţia trebuie să întoarcă o parte din intrările conţinute în acest director. Acest lucru este realizat folosind o funcţie de tipul filldir_t unde:
  - `dirent` este transmis ca parametru al funcţiei readdir;
  - `name` este numele intrării;
  - `name_len` este lungimea numelui;
  - `pos` este poziţia la care am ajuns în director;
  - `ino` este numărul inode-ului intrării;
  - `flags` identifica tipul intrării: DT_REG (fişier), DT_DIR(director), DT_UNKNOWN etc.

Enumerarea intrărilor din director începe de la poziţia specificată. De asemenea, dacă filldir_t atinge limita superioară (întoarce 1), atunci apelul trebuie să se oprească.

Exerciţii

- Folosiţi arhiva de sarcini a laboratorului.
- Primele două exerciţii se vor efectua în directorul `myfs/`. Reprezintă operaţii pe un sistem de fişiere virtual.
- Ultimele exerciţii se vor efectua în directorul `minfs`. Reprezintă operaţii pe un sistem de fişiere cu suport pe disc.
- Arhitectura sistemului de fişiere minfs este prezentată în următoarea diagramă:



- minfs conţine maxim 32 de inode-uri, datele fiecărui inod fiind conţinute de un singur bloc.
- Superblocul conţine un câmp imap pe 32 de biţi, în care fiecare bit indică utilizarea inode-ului respectiv.

1. **(2 puncte)** Pornind de la modulul creat la laboratorul precedent, adăugaţi operaţiile necesare pentru lucrul cu directoare.
   - Trebuie să precizaţi următoarele operaţii pentru directoare:
     - căutare (`lookup`)
     - creare de director (`mkdir`)
     - creare de fişier (`create`)
     - ştergere (`rmdir` şi `unlink`)
     - redenumire (`rename`).
     - link (`link`).
   - Nu trebuie să implementaţi operaţiile de citire/scriere din/în fişiere sau cele de lucru cu dispozitive / fişiere speciale / etc.
   - Folosiţi scriptul `test-myfs-1.bash` pentru testare.
   - **Hints**:
     - Definiţi o structură `myfs_dir_inode_operations` în care iniţializaţi funcţiile necesare.
     - Inode-urile de tip director vor folosi structura de mai sus (câmpul `i_op`).
     - Puteţi folosi `simple_dir_operations` pentru câmpul `i_fop`.
     - Va trebui să implementaţi operaţiile `mkdir` şi `create` pe inode-urile director. Pentru celelalte funcţii folosiţi apeluri generice (`simple_*`, `generic_*`).
     - Recomandăm modularizarea codului folosind o funcţie `mknod`, pe care o veţi putea folosi şi la exerciţiul următor. Folosiţi funcţia deja implementată de citire şi alocare de inode ? `myfs_get_inode`.
     - Pentru restul operaţiilor există deja definite funcţii generice.
     - Pentru mai multe detalii şi exemple, studiaţi sistemul de fişiere ramfs
     - Consultaţi secţiunea Operaţii asupra inode-urilor de tip director.
   - **Cum ştiu dacă merge?**
     - Odată montat sistemul de fişiere, veţi putea lista conţinutul directorului rădăcină, veţi putea creea/şterge noi fişiere, directoare, ierarhii.
     - **NU** veţi putea scrie în fişiere
       - echo "abc" > file va avea ca rezultat crearea fişierului `file`, împreună cu un cod de eroare
2. **(2 puncte)** Adăugaţi operaţiile de lucru cu fişiere (citire/scriere).
   - Va trebui să precizaţi operaţiile descrise în cadrul structurilor struct inode_operations, struct file_operations şi struct address_space_operations pentru fişier.
   - Folosiţi scriptul `test-myfs-2.bash` pentru testare.
   - **Hints**:
     - Definiţi structurile `myfs_file_inode_operations` şi `myfs_file_operations` şi completaţi câmpurile `i_op`, respectiv `i_fop` ale structurilor `inode` asociate fişierelor (regular file).
       - Folosiţi funcţiile **generice** oferite de VFS.
     - Definiţi structura `myfs_aops` şi iniţializaţi câmpul `inode->i_mapping->a_ops` al structurii inode.
       - Folosiţi funcţiile **generice** oferite de VFS.
     - Pentru mai multe detalii şi exemple, studiaţi sistemul de fişiere ramfs
       - Nu este nevoie să definiţi funcţie de tipul `set_page_dirty`.
     - Consultaţi secţiunea Operaţii asupra inode-urilor de tip fişier
   - **Cum ştiu dacă merge?**

    - Veţi putea face orice operaţii asupra fişierelor (cat, echo, vi).
3. **(2 puncte)** Intraţi în directorul `minfs/stage1`.
    - Implementaţi operaţia de listare a conţinutului unui director.
    - Trebuie să implementaţi operaţia `readdir` (struct inode_operations).
    - Urmăriţi în `minfs.c` şi `minfs.h` definiţiile structurilor `struct minfs_inode_info`, `struct minfs_inode` şi `struct minfs_dir_entry`.
    - **Hints**:
        - În funcţia `minfs_readdir` trebuie să parcurgeţi conţinutul directorului indicat de `filp`.
        - Folosiţi funcţia `filldir`.
        - Pentru a afla blocul de date obţineţi structura `struct minfs_inode_info` asociată inode-ului (folositi list_entry sau container_of). Din cadrul acestei structuri obţineţi câmpul `data_block`.
        - Folosiţi sb_bread pentru citirea blocului dat.
        - Datele din bloc sunt referite de câmpul `b_data` al structurii struct buffer_head (codul uzual va fi `bh->b_data`).
        - Blocul de date al directorului conţine un vector de cel mult `MINFS_NUM_ENTRIES` intrări de tipul `struct minfs_dir_entry`.
        - Pentru parcurgerea începeţi de la poziţia `filp->f_pos`. Nu uitaţi să actualizaţi câmpul `filp->f_pos` cu ultima intrare de director încărcată cu `filldir`.
        - Pentru debug actualizaţi variabila `entries`.
        - Folosiţi brelse pentru eliberarea blocului.
        - Urmăriţi funcţia minix_readdir. Funcţia este destul de complicată pentru că foloseşte un cache în memorie. Ignoraţi acea parte.
        - Consultaţi secţiunea Operaţii asupra inode-urilor de tip director.
    - **Cum ştiu dacă merge?**
        - ls -la în directorul montat ar trebui să arate a.txt
        - Există şi garbage în output-ul ls? Cărui fapt se datorează acest lucru?
4. **(1.5 puncte)** Trebuie să implementaţi operaţia `lookup` (struct inode_operations).
    - **Hints**:
        - Urmăriţi funcţia `minfs_lookup`.
        - Implementaţi funcţia `minfs_find_entry`.
        - Trebuie parcurs directorul în care se află intrarea `dentry` (`dentry->d_parent->d_inode`).
        - Blocul de date al directorului conţine un vector de cel mult `MINFS_NUM_ENTRIES` intrări de tipul `struct minfs_dir_entry`
        - Trebuie verificată prezenţa numelui (`dentry->d_name.name`) în director.
        - Structura de tip `buffer_head` obţinută va fi eliberată în apelant.
        - Urmăriţi funcţia minix_find_entry.
        - Consultaţi secţiunea Operaţii asupra inode-urilor de tip director.
    - Fişierul `mk_minfs.c` este folosit pentru formatarea unui disc cu sistemul de fişiere `minfs`.
    - Formatarea creează fişierul `a.txt` în directorul rădăcină.
    - Pentru testare folosiţi `test-minfs-1.bash`
    - **Cum ştiu dacă merge?**
        - ls -la nu mai arată garbage în dreptul intrării `a.txt`
5. **(3.5 puncte)** Intraţi în directorul `minfs/stage2`.
    - Completaţi funcţiile `minfs_readdir` şi `minfs_find_entry` cu implementarea de la exerciţiul anterior.
    - Implementaţi operaţia de creare a unui fişier (regular file).
    - Trebuie să implementaţi operaţia `create` (struct inode_operations).
    - **Hints**:
        - Urmăriţi funcţia `minfs_create`.
        - Implementaţi funcţia `minfs_new_inode`.
            - Trebuie găsit primul inode liber din imap (`sbi->imap`). Folosiţi operaţii de lucru pe biţi (find_first_zero_bit şi _set_bit).
            - Trebuie creat (new_inode) şi iniţializat un inode.
            - Urmăriţi funcţia minix_new_inode.
        - Implementaţi funcţia `minfs_add_link`.
            - Trebuie adăugat un nou dentry (`struct minfs_dir_entry`) în blocul de date al directorului părinte (`dentry->d_parent->d_inode`).
            - Urmăriţi funcţia minix_add_link.
        - Consultaţi secţiunea Operaţii asupra inode-urilor de tip director.
    - Pentru testare folosiţi `test-minfs-2.bash`

Soluţii

- Soluţii exerciţii laborator 10

Resurse utile

1. Robert Love ? Linux Kernel Development, Second Edition ? Chapter 12. The Virtual Filesystem
2. Understanding the Linux Kernel, 3rd edition - Chapter 12. The Virtual Filesystem
3. Linux Virtual File System (presentation)
4. Understanding Unix/Linux Filesystem
5. Creating Linux virtual filesystems
6. The Linux Virtual File-system Layer
7. The Linux Documentation Project - VFS
8. The "Virtual File System" in Linux
9. A Linux Filesystem Tutorial
10. The Linux Virtual File System
11. File Systems
12. Documentation/filesystems/vfs.txt
13. Sursele sistemelor de fişiere
    - Sistemul de fişiere procfs (*Process File System*) - sursele sistemului de fişiere procfs şi documentaţia
    - Sistemul de fişiere ramfs (*Random Access Memory Filing System*) - sursele sistemului de fişiere ramfs şi documentaţia
    - Sistemul de fişiere romfs (*Read-Only Memory File System*) - sursele sistemului de fişiere romfs şi documentaţia
    - Sistemul de fişiere minix - sursele sistemului de fişiere minix şi informaţii: Minix Homepage, Operating Systems Design and Implementation, Third Edition - Chapter 5. File Systems
    - Sistemul de fişiere bfs (*UnixWare Boot FileSystem*)- sursele sistemului de fişiere bfs, documentaţia şi informaţii: The BFS filesystem structure, Linux Kernel 2.4 Internals - 3.7. Example Disk Filesystem: BFS

Note

1. `super_operations` pentru mai multe detalii despre superoperaţii revedeţi Laboratorul 9. Drivere sisteme de fişiere (Linux) partea 1
2. `truncate_inode_pages, clear_inode` după cum scrie şi într-un comentariu din funcţia generic_delete_inode: *Filesystems implementing their own s_opdelete_inode are required to call truncate_inode_pages() and clear_inode()internally*
3. `address_space` mai multe detalii despre structura `address_space` găsiţi în *Robert Love ? Linux Kernel Development, Second Edition ? Chapter 15. The Page Cache and Page Writeback*
4. `minix_get_block` mai multe despre implementarea funcţiei `minix_get_block` găsiţi în secţiunea Operaţii asupra spaţiului de adresă
5. `minix_aops` structura `minix_aops` a fost descrisă în secţiunea Operaţii asupra spaţiului de adresă
6. `d_alloc_root` pentru mai multe detalii despre iniţializarea superblocului şi un exemplu de utilizare al funcţiei `d_alloc_root` revedeţi Laboratorul 9. Drivere sisteme de fişiere (Linux) partea 1