Sistemul Virtual de Fişiere

Mihai Budiu — mihaib@cs.cmu.edu
`http://www.cs.cmu.edu/~mihaib/`

ianuarie 1998

Subiect: VFS: Virtual File System

Cunoştinţe necesare: cunoştinţe generale despre sistemele de fişiere; familiaritate cu arhitectura unui sistem de fişiere;

Cuvinte cheie: vnod, sistem de fişiere virtual, montare, modularitate

Cuprins

1 Interfaţă şi implementare

Ingineria programării ne învaţă că putem ascunde un acelaşi obiect sub o sumedenie de operaţii diferite, şi alternativ, că putem inventa seturi de operaţii distincte pentru a opera asupra unei aceleiaşi entităţi.

Avem deci o oarecare *independenţă* între *implementarea* unui obiect (de pildă o stivă poate fi implementată ca o listă sau ca un vector) şi *interfaţa* unui obiect, care este setul de operaţii pe care le efectuăm asupra lui (de exemplu o stivă va oferi operaţii de genul "pune element (push)", "extrage element (pop)", "verifică dacă sunt elemente (empty?)").

Aparent paradoxal, tradiţia ne învaţă că mai importantă decît implementarea unei creaturi informatice este interfaţa ei. De ce? Din motive de *compatibilitate*. Atîta vreme cît un obiect are aceeaşi interfaţă (şi un acelaşi comportament), toţi cei care care îl folosesc pot rămîne neschimbaţi, chiar dacă implementarea obiectului se schimbă. De aceea este *extrem* de important să proiectăm interfeţele cum trebuie de la început.

Putem vedea importanţa acestui precept la scară industrială într-un exemplu izbitor. Proiectanţii procesorului 8086 de la Intel nu au putut rezista tentaţiei de a introduce în setul de instrucţiuni al procesorului o mulţime de operaţii exotice (de exemplu operaţii pe şiruri de numere zecimale cu cifre pe cîte 4 biţi), pentru că aveau destui tranzistori nefolosiţi pe circuitul integrat. Şi-au zis: "în definitiv ce ne costă?". Eroare fatală. Setul de instrucţiuni este interfaţa unui procesor, şi ca atare trebuie să rămînă neschimbat chiar atunci cînd arhitectura internă (implementarea) procesorului se schimbă. Aşa că aceste instrucţiuni pe care nu le foloseşte (practic) nimeni există şi în setul de instrucţiuni din 80486, Pentium, Pentium Pro şi Pentium II. Şi ştiţi de ce nu există procesoare Pentium la mai mult de 300Mhz, dar există procesoare Alpha la 600Mhz? Din cauza setului de instrucţiuni: anumite instrucţiuni Pentium pur şi simplu nu pot fi implementate foarte rapid (una din ele fiind cea de mai sus).

Asta este drama: interfaţa unui produs longeviv tinde să supravieţuiască implementării. Din cauza aceasta cel care proiectează o interfaţă trebuie să fie extrem de grijuliu, pentru că are de luptat cu un adversar colosal: timpul. El trebuie să anticipeze evoluţia unei interfeţe şi a utilizării ei.

Vom vedea în acest articol o interfaţă excelent proiectată cu peste un sfert de secol în urmă, care a evoluat şi supravieţuit tuturor încercărilor la care a fost supusă. Este vorba de interfaţa (setul de operaţii) cu fişiere în sistemul de operare Unix[1].

2 Fişierul ca abstracţiune

Un fişier în Unix este un simplu container de date; un fel de array de octeţi de o lungime (teoretic) arbitrară. Independent de modul în care este implementat un fişier operaţiile pe el se fac cu un set extrem de redus de operaţii esenţiale (`open()`, `close()`, `read()`, `write()`, `lseek()`) la care se adaugă o sumedenie de operaţii secundare ca importanţă sau care sunt folosite pentru a manipula *directoare*. Să ne reamintim pe scurt cum se folosesc aceste operaţii:

- `open()` "deschide" un fişier şi are ca rezultat un "mîner" (file handle) cu care fişierul este apoi manipulat (un număr mic întreg);

[1] Am prezentat într-un articol din PC Report din decembrie 1997 această interfaţă pe larg; o copie a articolului este disponibilă din pagina de web a autorului. Vom relua însă aici ideile esenţiale.

- `close()` este folosită pentru a închide un fişier care nu mai este necesar;
- `lseek()` este folosită pentru a fixa locul în fişier unde se va face următoare operaţie de scriere/citire; scrierea şi citirea vor avansa acest "cursor" cu cantitatea de date pe care o transferă. Astfel este favorizat în mod implicit accesul *secvenţial* la fişiere;
- `read()`, respectiv `write()` citesc şi scriu date dintr-un buffer din memorie în fişier.

Avem un exemplu de folosire a funcţiilor de acces la fişiere în programul de mai jos, care copiază fişierul "sursa" în fişierul "destinatie".

```
#include <unistd.h>     /* pentru open(), exit() */
#include <fcntl.h>      /*        O_RDWR */
#include <errno.h>      /*        perror() */

void fatal(char * mesaj_eroare)
{
        perror(mesaj_eroare);
        exit(1);
}

int main(void)
{
        int miner_sursa, miner_destinatie;
        int copiat;
        char buf[1024];

        miner_sursa = open("sursa", O_RDONLY);
        miner_destinatie = open("destinatie", O_WRONLY | O_CREAT, 0644);
        if (miner_sursa < 0 ||
            miner_destinatie < 0)
                fatal("Nu pot deschide un fisier");
        lseek(miner_sursa, 0, SEEK_SET);
        lseek(miner_destinatie, 0, SEEK_SET);
        while ((copiat = read(miner_sursa, buf, sizeof(buf)))) {
                if (copiat < 0)
                        fatal("Eroare la citire");
                copiat = write(miner_destinatie, buf, copiat);
                if (copiat < 0)
                        fatal("Eroare la scriere");
        }
        close(miner_sursa);
        close(miner_destinatie);
        return 0;
}
```

2.1 Vnodul: un i-nod virtual

Un sistem de operare de tip Unix poate opera simultan cu mai multe tipuri de sisteme de fişiere. De pildă toate sistemele suportă pe lîngă un sistem de fişiere pe un disc local un sistem de fişiere la distanţă numit NFS: network file system, creat de firma SUN. Tot firma Sun Microsystems a construit mecanismele necesare suportării unei varietăţi de fişiere implementate complet diferit simultan de către un singur nucleu (de fapt nu cei de la Sun au inventat noţiunea, însă implementarea lor, descrisă în cele ce urmează, a devenit practic un standard). Terminologia folosită in acest articol este deci cea propusă de Sun, deşi exemplele de cod C vor fi din sistemul de operare Linux.

Cheia constă în următorul fapt: toate sistemele de fişiere prezintă *aceeaşi interfaţă*. Cu alte cuvinte, utilizatorul (şi vom vedea că şi nucleul în interiorul său) acţionează cu exact aceleaşi funcţii asupra tuturor fişierelor, indiferent că se află pe un disc local sau pe unul la distanţă sau pe un floppy DOS, etc.; programul de mai sus va fi scris în exact acelaşi fel pentru toate aceste cazuri.

În interiorul nucleului fiecare fişier este reprezentat printr-o structură de date numită *vnod*: un nod de informaţii virtual. Într-un articol despre structura sistemului de fişiere din Unix din PC Report din decembrie 1996 am văzut că sistemul de fişiere tradiţional din Unix folosea pe disc o structură de date numită *inod* pentru a descrie atributele fişierelor. Vnodul este o generalizare a inodului, care este însă rezidentă în memoria calculatorului, şi nu pe un disc, şi cu care nucleul reprezintă fiecare fişier deschis.

Vnodul conţine două feluri de cîmpuri în interiorul său:

- O serie de cîmpuri comune tuturor sistemelor de fişiere; din această cauză aceste cîmpuri se mai numesc "independente de sistemul de fişiere";
- Cîmpuri specifice fiecărui sistem de fişiere în parte.

Structura ar arăta deci (în principiu) cam aşa:

```
struct vnode {
    cimpuri_independente;
    union {
        struct msdos_inode_info msdos_i;
        struct nfs_inode_info   nfs_i;
        struct sysv_inode_info  sysv_i;
        ....
    } u;
};
```

Fiecare structură din uniunea `u` conţine atribute specifice fiecărui sistem de fişiere în parte. Dacă aveţi un sistem Linux (o variantă de Unix ale cărei surse sunt disponibile gratuit oricui) puteţi vedea definiţia vnodului în fişierul de surse al nucleului `/usr/src/linux/include/linux/fs.h` (din păcate Linux foloseşte pentru aceste structuri de date — cel puţin versiunile pînă la 2.0.30 — alte nume decît restul lumii; numele structurii este `struct inode` în loc de vnode, dar nu o să lăsăm asta să ne descurajeze).

Cel mai important cîmp independent de structură este o structură cu pointeri spre funcţii. Ca să fim concreţi vom folosi tot sursa Linux, unde numele structurii este `struct inode_operations` (numele ei corect ar fi `struct vnode_operations`).

Definiţia acestei structuri se găseşte în acelaşi fişier şi este extrem de interesantă; schematic arată aşa (am simplificat un pic pentru motive pedagogice):

```
struct inode_operations {
        int (*lseek) (struct inode *, struct file *, off_t, int);
        int (*read) (struct inode *, struct file *, char *, int);
        int (*write) (struct inode *, struct file *, char *, int);
        int (*open) (struct inode *, struct file *);
        void (*release) (struct inode *, struct file *);
        int (*fsync) (struct inode *, struct file *);
        int (*create) (struct inode *,const char *,int,int,struct inode **);
        int (*lookup) (struct inode *,const char *,int,struct inode **);
        int (*link) (struct inode *,struct inode *,const char *,int);
        int (*unlink) (struct inode *,const char *,int);
        ....
}
```

Vnodul conţine deci şi un pointer spre o structură care conţine pointeri către funcţiile care trebuie să opereze cu inodul însuşi!

Vom reveni mai jos asupra folosirii vnodurilor.

2.1.1 Partea independentă de arhitectură

Ce fel de atribute sunt prezente în orice vnod? Putem să ne facem o idee în Linux privind în fişierul `fs.h` indicat mai sus; printre altele: discul (perifericul) pe care se află acest fişier, numărul acestui inod pe acel periferic, drepturile şi posesorul fişierului, data modificării, etc. Mai sunt prezente structuri de date necesare nucleului pentru operaţii pe vnod: semafoare pentru sincronizarea accesului proceselor la vnod, liste înlănţuite de hash pentru căutarea rapidă a vnodurilor în memorie, structura cu funcţiile care operează asupra vnodului, descrisă mai sus.

2.1.2 Partea dependentă de arhitectură

Desigur, fiecare sistem de fişiere ţine informaţiile de care are nevoie în vnod în partea care-i este rezervată. De exemplu, pentru un sistem de fişiere Unix clasic (descris într-un articol mai vechi), vnodul va conţine lista blocurilor de pe disc care aparţin fişierului.

Această listă de blocuri nu va fi prezentă în cazul vnodurilor pentru fişiere de tip NFS, pentru că acestea nu sunt prezente pe un disc pe calculatorul local, ci pe unul la distanţă. Dimpotrivă, un vnod pentru un fişier NFS va conţine informaţii suficiente pentru a comunica cu serverul care deţine fişierul.

Pentru cazul sistemului de operare Linux, există pentru fiecare tip de sistem de fişiere care poate fi prezent în nucleu cîte un fişier header cu numele de tipul `include/linux/*_fs_i.h` care conţine structura de date privată respectivului sistem de fişiere; aruncaţi o privire prin ele.

Să facem în figura 1 un desen ca să rezumăm situaţia aşa cum este ea acum.

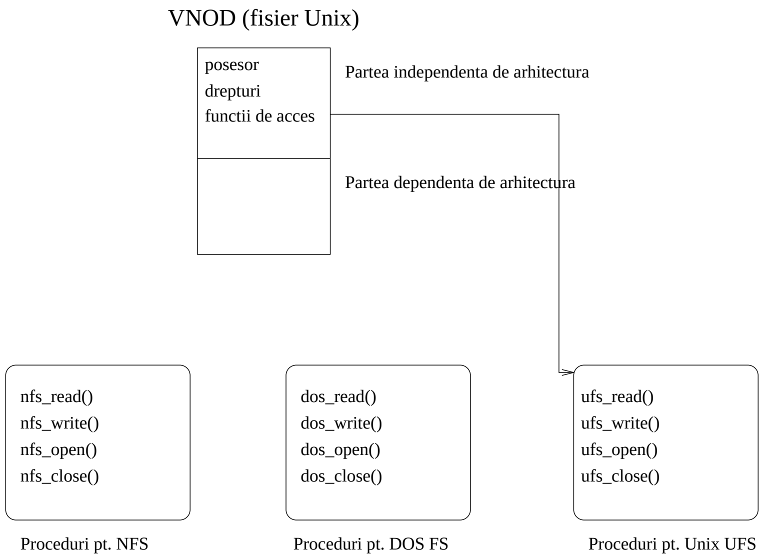

Figura 1: Vnodul

În limbajul programării orientate pe obiecte situaţia se descrie foarte sumar astfel: în nucleu există o clasă de bază virtuală numită vnod prin care nucleul reprezintă orice fişier; toate metodele acestei clase sunt pur virtuale (metodele sunt funcţiile din structura `inode_operations`). Fiecare tip de fişier particular suportat de un nucleu este o *clasă derivată* din această clasă de bază.

2.2 Vnoduri speciale

În articolul din decembrie 1997 am explicat faptul că în Unix sub abstracţia de fişier se ascund multe alte creaturi: ţevile (pipes), perifericele (fişierele speciale), pseudo-perifericele şi chiar procesele! Cu alte cuvinte toate aceste obiecte sunt accesate prin aceeaşi interfaţă, care include funcţiile descrise mai sus, `read(), close()`, etc.

În interior nucleul reprezintă toate aceste obiecte în acelaşi fel, şi anume prin vnoduri. În consecinţă, pentru nucleu fiecare periferic "deschis" este un vnod; în particular o partiţie de disc este reprezentată intern în nucleu tot ca un vnod.

3 Sistemul de fişiere ca abstracţiune

Toată povestea anterioară se repetă aproape identic pe un alt nivel: nucleul Unix se bazează pe lîngă abstracţia de fişier pe cea de *sistem de fişiere*. Structura prin care se reprezintă în nucleu un sistem de fişiere se numeşte Sistem de Fişiere Virtual (Virtual File System, VFS). Putem citi definiţia structurii pentru Linux în acelaşi fişier: `include/linux/fs.h`, unde (în terminologie specifică Linux) structura de date se cheamă `struct super_block`.

Dacă un vnod reprezintă un fişier individual care a fost deschis, un VFS reprezintă o întreagă partiţie care a fost montată în sistemul de fişiere (secţiunea următoare este consacrată acestei operaţii).

Tot aşa cum vnodul are o serie de metode (funcţii) care operează asupra lui, VFS conţine un pointer spre o serie de funcţii globale ale unui sistem de fişiere. Pentru Linux acestea sunt declarate în acelaşi fişier, în structura `struct super_operations`; cele mai importante operaţii sunt cele care citesc/scriu de pe o partiţie un inod. Iată un fragment din această structură:

```
struct super_operations {
        void (*read_inode) (struct inode *);
        void (*write_inode) (struct inode *);
        void (*put_inode) (struct inode *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        void (*statfs) (struct super_block *, struct statfs *);
        .....
};
```

4 Montarea

Cum ajung mai multe sisteme de fişiere (nu neapărat diferite arhitectural) să fie folosite simultan de un sistem Unix? În DOS sau Windows cînd se indică un fişier se indică (poate implicit) şi partiţia de disc unde fişierul se află (ex.: cu `C:`).

În Unix lucrurile stau puţin altfel: toate sistemele de fişiere disponibile utilizatorilor sunt *montate* (cu comanda `mount`) de către administratorul de sistem, de obicei cînd calculatorul bootează. Comanda aceasta (care se foloseşte de un apel de sistem cu acelaşi nume) are două argumente importante: un nume de partiţie, şi un nume de director[2].

De exemplu, presupunînd că am un hard disc numit `/dev/hda2`, pot să montez (dacă am drepturi de administrator) sistemul de fişiere aflat pe el peste directorul `/mnt` cu comanda:

```
mount /dev/hda2 /mnt
```

Să observăm în trecere că faptul că numai administratorul are dreptul de a monta sisteme de fişiere restrînge mult posibilitatea de propagare a viruşilor: un utilizator neprivilegiat nu poate accesa dischete sau discuri străine.

Ce înseamnă montarea? Poate fi montată numai o partiţie formatată şi pe care se află un sistem de fişiere. Prin montare directorul rădăcină al acelei partiţii este identificat cu directorul indicat la comanda `mount`, în cazul nostru `/mnt`. După execuţia comenzii, de la directorul `/mnt` în jos se va afla întregul arbore de directoare de pe partiţia `/dev/hda2`.

Am văzut cîteva lucruri foarte simple; am putea spune aproape banale. Uluitor este de pildă faptul că în interior nucleul foloseşte o singură structură de date, vnodul, pentru a reprezenta zeci de creaturi diferite:

[2] Articolul citat mai sus despre funcţionarea sistemelor de fişiere din Unix, din PC Report din decembrie 1997 explică cum se poate numi o partiţie folosind un nume ca al unui fişier, şi cum nucleul operează cu partiţiile. Vom vedea mai jos că nucleul trebuie să aibă de la bootare montată cel puţin o partiţie, numită *partiţia rădăcină* (root filesystem) ca să poată continua cu celelalte.

- Fişiere de naturi diferite de pe un acelaşi tip de sistem de fişiere (ex. directoare, fişiere);
- Fişiere de pe orice fel de tip de sistem de fişiere (ex. NFS, DOS, Unix);
- Fişiere speciale care reprezintă periferice (ex. partiţii de disc, consola, ecranul, benzi, mouse, etc);
- Pseudo-periferice (ex. ferestre din sistemul grafic, memoria virtuală a maşinii `/dev/mem`, gaura fără fund `/dev/null`, etc.);
- Procesele în curs de execuţie şi informaţiile despre ele (ex. `/proc/*`);
- Mecanisme de comunicare inter-proces (ex. ţevi (pipes, sockets, etc.)).

Toate aceste obiecte sunt reprezentate în acelaşi fel pentru că oferă fiecare (un subset) al aceleiaşi interfeţe, bazată pe `read(), write(), open(), close(), ioctl()`.

Acum vom încerca să vedem cum funcţionează aceste abstracţii. Să urmărim deci o serie de operaţii în sistemul de fişiere.

5 Sistemul de fişiere în acţiune

Vom parcurge mai multe etape:

1. Vom vedea de unde ştie nucleul să manipuleze atîtea feluri de sisteme de fişiere şi obiecte diferite;
2. Vom vedea cum apare prima partiţie montată;
3. Vom vedea cum celelalte partiţii sunt montate;
4. Vom vedea cum se execută o operaţie asupra unui fişier care trebuie să traverseze o cărare pe mai multe partiţii.

Partea spectaculoasă este că, datorită interfeţei identice a tuturor fişierelor din sistem, codul care operează cu fişiere poate fi scris în mare măsură *complet independent de natura fişierelor*, fie ele DOS, Unix sau NFS sau altceva. Vom vedea că cea mai mare parte a operaţiilor se petrec într-un nivel software care se comportă ca un comutator gigantic, care pe măsură ce acţionează asupra unor fişiere aflate pe partiţii diferite comută între codul feluritelor sisteme de fişiere. Acest comutator se numeşte Virtual File System Switch, şi se abreviază cîteodată cu `vfssw`.

Plasamentul acestui cod în interiorul nucleului este simbolizat în figura 2.

5.1 Compilarea nucleului

Fiecare sistem de fişiere îşi aranjează altfel datele pe disc; anumite sisteme ca NFS cer colaborarea unui client şi a unui server pentru a oferi servicii de fişiere. Fiecare este implementat prin alte proceduri, chiar dacă oferă acelaşi set de operaţii. Administratorul de sistem hotărăşte la compilarea nucleului care din sistemele de fişiere disponibile vor face parte din nucleu. Fiecare

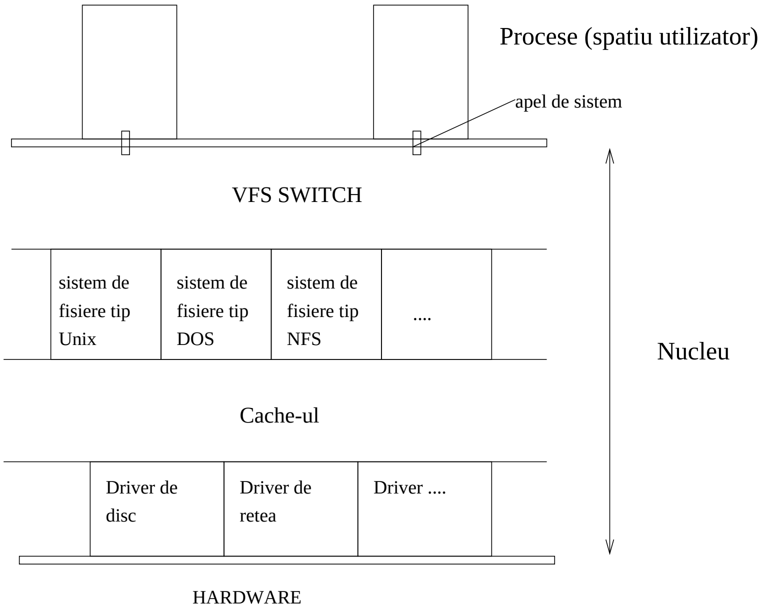

Figura 2: Sistemul de fişiere în nucleu

pune la dispoziţie un set de funcţii pentru manipularea structurii vfs (cele din structura `struct super_operations` de mai sus) şi pentru manipularea fişierelor însele (vnoduri).

În cazul sistemului de operare Linux fiecare sistem de fişiere are sursele în propriul lui arbore de directoare plecînd din `/usr/src/linux/fs/`.

5.2 Boot-area

Cînd sistemul de operare bootează, înainte de lansarea proceselor, se execută o secţiune de iniţializări în care sunt chemate procedurile de iniţializare ale tuturor subsistemelor nucleului; fiecare driver se iniţializează şi apoi sistem de fişiere are ocazia să se iniţializeze.

Înainte de a lansa orice proces nucleul montează prima partiţie, partiţia rădăcină. Această partiţie a fost configurată de administratorul de sistem la construirea sistemului. Pe această partiţie se găsesc cele mai importante directoare ale sistemului, fără de care acesta nu poate funcţiona. Acestea sunt:

`/dev/` care conţine toate numele fişierelor speciale şi inodurile lor; aici fiecare hard-disc are un nume şi un număr major şi minor[3] care indică nucleului ce partiţie fizică corespunde fiecărui nume;

`/etc/` care conţine toate fişierele de configurare ale sistemului şi scripturile care se execută la iniţializare;

`/bin/` care conţine toate fişierele executabile esenţiale, printre care shell-ul, comanda `mount`, utilitare de reparat discul, etc.

[3]Despre numerele majore şi minore vedeţi articolul citat anterior.

După montarea partiţiei nucleul deschide directorul rădăcină care va rămîne deschis pînă la oprirea sistemului. Asta înseamnă că îi alocă un vnod pe care îl iniţializează corespunzător şi pe care îl păstrează în memorie; o variabilă globală punctează la acest vnod. (Aceste operaţii sunt executate de nucleul Linux în funcţia `mount_super()` pe care o puteţi găsi prin surse într-un loc depinzînd de versiunea nucleului pe care o aveţi.)

Din clipa asta sistemul este funcţional.

Să vedem mai departe cum procedează nucleul pentru a monta o nouă partiţie în arborele de directoare şi cum nucleul procedează pentru a deschide un fişier.

5.3 Montarea

Să presupunem că avem doar o partiţie montată, cea rădăcină. Să vedem ce face nucleul la executarea comenzii:

```
mount -t msdos /dev/hda1 /mnt
```

care îi cere să monteze o partiţie cu sistem de fişiere de tip MS-DOS peste directorul `/mnt`.

Nucleul face următoarele operaţii:

1. Descifrează tipul sistemului de fişiere pentru a şti care porţiune de cod (dintre multiplele sisteme de fişiere ale nucleului) o să manipuleze structurile de date de pe această partiţie;

2. Deschide directorul indicat (`/mnt`); asta înseamnă citirea de pe disc a inodului acelui director într-un vnod din memorie; acest vnod va rămîne deschis atîta vreme cît sistemul de fişiere va fi montat deasupra lui;

3. Cheamă funcţia de iniţializare corespunzătoare acestui sistem de fişiere (msdos), care funcţie construieşte structura VFS (numită în Linux `super_block`) şi îi iniţializează cîmpurile, de obicei citind primele sectoare de pe hard-disc din partiţia indicată (`/dev/hda1`);

4. Deschide fişierul `/dev/hda1`, care este un fişier special de tip bloc corespunzînd unei partiţii de disc;

5. Deschide directorul rădăcină al partiţiei montate (`/dev/hda1`) într-un vnod în memorie;

6. Marchează faptul că cele două vnoduri (ale directorului `/mnt` şi al rădăcinii partiţiei noi) practic "coincid": asta va permite traversarea unor cărări de genul `/mnt/tmp`, care încep pe o partiţie şi se termină pe alta.

Puteţi citi codul apelului de sistem `mount(2)` pentru Linux în fişierul `/usr/src/linux/fs/super.c`; pe lîngă cele spuse codul mai face o sumedenie de verificări pe care le-am sărit.

5.4 Deschiderea unui fişier

Există două mari categorii de apeluri de sistem care operează cu fişiere: apeluri care primesc cărări (path) către un fişier, şi care primesc un descriptor de fişier (mîner, handle). Descriptorii am văzut că se obţin deschizînd un fişier cu `open()`, căreia i se dă o cărare.

Pe scurt, pentru a opera asupra unui fişier trebuie întîi parcursă o cărare pînă la el. După cum ştim există două feluri de cărări: relative la directorul curent al procesului sau absolute, care pornesc de la rădăcina sistemelor de fişiere (cele din urmă se scriu începînd cu semnul /). Diferenţa constă doar în vnodul de la care porneşte operaţia de traversare a cărării: într-un caz este vnodul directorului curent al procesului, care este permanent menţinut de nucleu într-o variabilă asociată procesului, iar în celălalt caz directorul rădăcină, care este menţinut într-o altă variabilă asociată procesului (în Unix un proces poate să-şi schimbe ceea ce crede că este rădăcina întregului arbore de directoare cu apelul de sistem `chroot(2)`).

Pentru nucleu deschiderea unui fişier înseamnă:

- Alocarea unui vnod liber în memorie;
- Citirea informaţiilor dependente şi independente de pe disc (sau de unde-or fi) în vnod.

5.4.1 Starea iniţială

Să revizuim însă structurile de date ale nucleului aşa cum se prezintă ele în această clipă; ele sunt înfăţişate în figura 3.

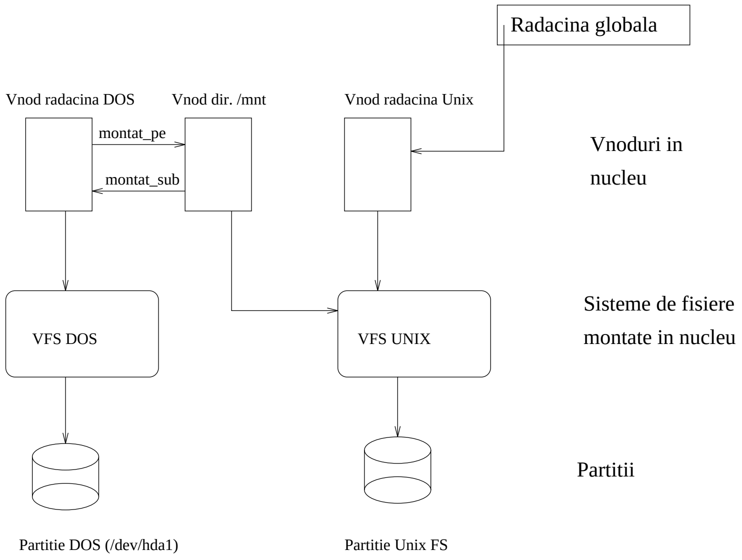

Figura 3: Structurile de date în nucleu

5.4.2 Funcţia `namei`

Funcţia din nucleu care traduce o cărare într-un vnod în acest fel se numeşte în mod tradiţional în sistemele derivate din Berkeley Unix (şi în Linux) `namei()`, datorită faptului că traduce un nume de fişier într-un i(v)nod. (Numele funcţiei în sisteme descendente din System V de la AT&T este `lookuppn()`, de la lookup path name.) Într-un pseudo-cod funcţia `namei()` arată cam aşa (cu detalii e mult mai complicată):

```
vnode * namei(char * carare, vnode * start)
{
        char * componenta;
        vnode * curent = start;

        while (carare) {
                if (carare && !DIRECTOR(curent))
                        return NULL;
                componenta = extrage_prima_componenta(carare);
                carare = elimina_prima_componenta(carare);
                if (!strcmp(componenta, "..") &&
                    curent->montat_pe) {
                        if (curent != ROOT_VNODE)
                                curent = curent->montat_pe;
                        continue;
                }
                if (curent->montat_sub) {
                        curent = curent->montat_sub;
                        continue;
                }
                curent = curent->operatii->lookup(curent, componenta);
                if (!curent)
                        return NULL;
        }
        return curent;
}
```

Să vedem pas cu pas cum operează nucleul pentru a deschide fişierul `/mnt/tmp/a`.

1. Este invocată funcţia `namei("mnt/tmp/a", ROOT_VNODE);`
2. Pentru că mai avem componenente în cărare trebuie ca `curent` să fie un vnod de director; se verifică acest lucru;
3. Este extrasă prima componentă din cărare (`mnt`); cărarea rămîne `tmp/a`;
4. `curent = ROOT_VNODE`; peste `curent` nu e montat nimic;
5. Se cheamă indirect funcţia `curent->operatii->lookup(curent, "mnt")`, care caută în directorul dat de vnodul `curent` un fişier cu numele `mnt` şi returnează vnodul său de pe aceeaşi partiţie (partiţia rădăcină în cazul nostru); aceasta este noua valoare a lui `curent`; dacă vnodul căutat nu se afla anterior căutării în memorie, funcţia `lookup()` îl alocă şi îl citeşte de pe disc; funcţia `lookup()` construieşte şi structura de `->operatii` din cadrul vnodului, păstrînd operaţiile de la directorul în care se caută (dacă două vnoduri sunt pe aceeaşi partiţie sunt deci din acelaşi sistem de fişiere şi deci au aceleaşi funcţii de acces!);
6. Se reia bucla `while`;

7. Pentru că pe directorul `/mnt` este montată partiţia `/dev/hda1`, valoarea lui `curent->montat_sub` este chiar vnodul care este rădăcina acestei partiţii (care este în memorie); `curent` ia această valoare iar bucla se reia prin execuţia lui `continue`;

8. În clipa asta `curent` punctează la un vnod dintr-un sistem de fişiere complet diferit (de tip MS-DOS); pe nucleu însă acest lucru nu-l interesează, pentru că vnodul are *aceleaşi operaţii*, inclusiv `lookup` (chiar dacă ele sunt implementate într-un mod complet diferit);

9. Într-un mod perfect similar se extrage prima componentă a cărării, `tmp` care este căutată (cu apelul funcţiei `curent->operatii->lookup(curent, "tmp")` în directorul al cărui vnod este `curent`; se schimbă valoarea lui `curent`;

10. O ultimă trecere prin buclă va găsi la fel vnodul fişierului `a` şi-l va încărca în memorie; acesta este rezultatul funcţiei `namei()`.

5.5 O operaţie pe un fişier

Cheia în implementarea funcţiei `namei` este apelul indirect de funcţie `curent->operatii->lookup()`. Vom înţelege la ce foloseşte acest apel privind modul în care nucleul execută o operaţie pe un fişier deja deschis; să vedem un posibil cod al apelului de sistem `write()`:

```
int write(int miner, char * buffer, unsigned cantitate)
{
        struct vnode * v;
        struct file * f;

        f = proces_curent->fisiere_deschise[miner];
        v = f->vnode;
        return v->operatii->write(v, buffer, cantitate);
}
```

(Am eliminat toate testele de corectitudine a argumentelor.) Nucleul întîi indexează într-un array al procesului curent pentru a găsi vnodul `v` al fişierului deschis anterior al cărui "mîner" a fost returnat utilizatorului (rolul structurii `struct file` nu ne interesează deocamdată; vom reveni asupra ei într-un alt articol). Apoi nucleul cheamă din nou indirect funcţia `write`, aşa cum apare ea între operaţiile asociate vnodului găsit.

Frumuseţea acestei scheme este următoarea: fiecare sistem de fişiere îşi organizează altfel datele pe disc; Unix foloseşte o schemă complicată în care fişierele sunt descrise prin inoduri, MS-DOS descrie blocurile fişierelor printr-o structură numita FAT (file access table), etc. Dar atîta vreme cît ambele sisteme de fişiere pun la dispoziţie o funcţie care scrie într-un fişier date, nu contează prea tare că această funcţie este complet diferit implementată pentru cele două sisteme. Important este că are *aceeaşi interfaţă*!

Fiecare vnod poartă cu el din momentul în care este adus în memorie propriul lui vector de operaţii. Vnodurile pentru sisteme Unix şi vnodurile pentru sisteme de fişiere MS-DOS au ambele o operaţie `write()`, care primeşte aceleaşi argumente şi returnează aceleaşi rezultate, chiar dacă intern se comportă complet diferit.

Din această cauză utilizatorii pot trata în Unix fişiere de naturi foarte diferite (şi nu numai fişiere) ca pe obiecte de acelaşi tip.

6 Concluzie

Există o oarecare independenţă între operaţiile pe care le putem face asupra unui obiect (interfaţa sa) şi modul în care acele operaţii sunt realizate (implementarea). O interfaţă bine proiectată poate avea consecinţe dramatice.

Cu siguranţă flexibilitatea acestei interfeţe este unul dintre ingredientele care a asigurat succesul sistemului de operare Unix şi a paradigmelor sale. Avem aici un exemplu splendid de *modularitate*: părţi complet diferite constitutiv au aceeaşi interfaţă încît pot fi practic substituite una alteia, ca nişte bucăţi de Lego.

Mai mult de jumătate din sursele C ale unui nucleu sunt drivere (pentru Linux 2.0.30 asta înseamnă aproape o jumătate de milion de linii de cod!). Nici un om nu poate înţelege atît de mult cod. Dar datorită modularităţii nimeni nu trebuie să înţeleagă toate piesele: trebuie doar să cunoşti interfeţele; implementarea poate fi oricare. Înţelegînd interfeţele înţelegi şi cum funcţionează întregul.