**Arhitectura kernel-ului Linux**

Nechifor Cosmin Adrian (433A)

Sandu Ciprian Nicusor (433A)

Stoica Alexandru Sebastian (433A)

Cuprins

Nechifor Cosmin: Capitolul 1 - Introducere (p. 2)

 Capitolul 2 - User scape, kernel space + SCI (p. 9)

Sandu Ciprian: Capitolul 3 - Virtual File System (p. 11)

 -> Arhitectura VFS (p.13)

 -> Superblock (p. 14)

 -> Index node (p. 18)

 -> Directory entry (dentry) (p. 19)

 -> File object. Relatia intre obiecte (p.20)

Stoica Alexandru: Capitolul 4 - Process Management (p. 21)

 -> Reprezentarea proceselor (p. 21)

 -> Managementul de procese (p. 23)

 -> Crearea de procese (p. 24)

 -> Planificarea proceselor. Distrugerea proceselor (p.25)

**Capitolul 1.Introducere. Scurt istoric. Ce este acela un kernel si o comparatie succinta intre microkernel si monolithic kernel.**

Kernel-ul se defineste ca nucleul unui sistem de operare al carui rol este de a controla resursele sistemului si de a le pune la dispozitie proceselor/programelor de care are nevoie utilizatorul.

A fost dezvoltat ca urmare a faptului ca fara un SO, programatorul era nevoit sa lucreze in limbaj de programare de nivel scazut, gen Assembly, astfel incat incat utilizarea resurselor si programarea in sine era extrem de dificila. De aceea, in anii ‘60 MIT s-a ocupat de crearea unui sistem de operare experimental numit Multics (Multiplexed Information and Computing Service), acesta fiind necesar datorita puterii de calcul crescute ale noilor calculatoare. In anii ‘70, a fost pus la punct un sistem de operare bazat pe limbajul C denumit UNIX.

Dupa 20 de ani, Andrew Tanenbaum a realizat o versiune minimala a UNIXului ce putea rula lejer pe calculatoarele personale. A numit acest SO MINIX (Minimal UNIX) si a fost baza dupa care a fost creat Linuxul. Acesta a fost programat de catre Linus Torvalds in anii ‘90.

**In acest grafic este reprezentata variatia numarului de linii de cod din care este format kernelul de-a lungul anilor.**

Kernelul poate fi considerat programul principal al sistemului de operare, de care depind toate aplicatiile si procesele utilizatorului. Dupa initializarea PC-ului, el este prima parte a sistemului de operare ce este incarcata in memorie si ramane prezent pana la inchiderea calculatorului sau a SO. Deoarece este de o importanta critica pentru functionarea sistemului, kernelul este in incarcat intr-o zona protejata a memoriei astfel incat acesta sa nu poata fi modificat din greseala.

Functiile kernelului sunt numeroase si complexe, de aceea este de preferat ca marimea acestuia sa fie cat mai mica. Dintre cele mai importante pot fi mentionate urmatoarele :

**a) Managmentul de memorie** : se asigura ca fiecare program isi are propria zona de memorie care nu poate fi suprascrisa in mod normal. Creaza notiunea de memorie virtuala folosindu-se de segmente de valori mici(in general, 4kB) numite pagini. Prezinta si un mecanism de a crea memorie suplimentara in cazul in care memoria principala este plina folosindu-se de memoria externa (gen harddisk, stick, etc) pentru a stoca informatii pe ele; acest proces se numeste swapping.

Fiecare pagina a sistemului are asociata o structura ce contine caracteristicile acesteia (marimea/adresa, etc). In Linux, structura se declara in <linux/mm.h> si are forma aceasta :

**typedef struct page {**

**struct list\_head list;**

**struct address\_space \*mapping;**

**unsigned long index;**

**struct page \*next\_hash;**

**atomic\_t count;**

**unsigned long flags;**

**struct list\_head lru;**

**struct page \*\*pprev\_hash;**

**struct buffer\_head \* buffers;**

**#if defined(CONFIG\_HIGHMEM) || defined(WANT\_PAGE\_VIRTUAL)**

**void \*virtual;**

**#endif /\* CONFIG\_HIGMEM || WANT\_PAGE\_VIRTUAL \*/**

**} mem\_map\_t;**

Se definesc urmatoarele campuri importante :

* **list** :Paginile apartin mai multor liste iar acest camp este folosit ca inceputul listei. De asemenea, este folosit ca sa lege mai multe pagini libere impreuna.
* **mapping** : Adresa de inceput a fisierului de care apartine pagina.
* **index** : Este folosit ca offsetul adresei.
* **count** : Are rolul de a arata daca pagina este folosita (in acest caz are o valoare >0) sau daca este libera (valoare ei fiind 0)
* **flags** : Are rolul de a descrie pagina in functie de ce flaguri sunt activate.

**b) Managmentul de procese** : pentru a putea utiliza la maxim capacitatea calculatorului, s-a creat notiunea de task(proces) care reprezinta o instanta unui program ; functionarea fiecarui task este dictata de kernel, care aloca timp de calcul/IO fiecaruia astfel incat sa nu existe conflicte.

Fiecare process este continut intr-o structura numita task\_struct. S-au ales doar partile importante in urmatoarele linii de cod :

**struct task\_struct {
 volatile long state;
 void \*stack;
 unsigned int flags;
 int prio, static\_prio;
 struct list\_head tasks;
 struct mm\_struct \*mm, \*active\_mm;
 pid\_t pid;
 pid\_t tgid;
 struct task\_struct \*real\_parent;
 char comm[TASK\_COMM\_LEN];
 struct thread\_struct thread;
 struct files\_struct \*files;

};**

Se definesc campuri tipice, gen :

- starea procesului (TASK\_RUNNING in cazul in care acesta ruleaza, TASK\_INTERRUPTIBLE daca este pus in pauza, TASK\_UNINTERRUPTIBLE difera de ultimul caz in sensul ca nu poate fi scos din aceasta stare, TASK\_STOPPED daca acesta nu functioneaza in acel moment)

- flaguri (PF\_STARTING in cazul in care procesul este nou creat, PF\_MEMALLOC daca tocmai se aloca memorie procesului)

- prioritatea (cu cat valoarea este mai mica, cu atat prioritatea procesului e mai mare)

 **c) Managmentul de fisiere si de I/O** : kernelul se ocupa de transferul de fisiere intre memoria principala si harddisk; de asemenea, are rolul de a pune la dispozitie drivere pentru dispozitivele interne/externe, cum ar fi placa de retea/tastatura/stickuri.

 **d) Managmentul resurselor prin intermediul Sistem Call** : pe scurt, procesele pot cere o resursa/o permsiune de la sistemul de operare pe care nu o detin in mod curent folosind o cerere de sistem. Se va discuta mai mult despre aceasta in capitolul urmator.

 **e) Protectia sistemului** : kernelul prezinta o serie de masuri de siguranta cum ar fi : nivele de securitate pentru fisiere, impartirea spatiului de memorie in user space si kernel space(la care nu are acces utilizatorul in mod normal), exectarea unui cod care a fost creat doar de un compilator sigur, etc.

**In aceasta poza sunt prezentate functiile tipice ale unui kernel.**

Un kernel monolitic reprezinta un tip complex de kernel, care pe langa functiile de baza ale sistemului de operare contine si drivere pentru dispozitive ; de asemenea, toate acestea exista si fuctioneaza in kernel space. Kernelurile monolitice moderne au posibilitatea de a incarca module suplimentare in functie de necesitatile sistemului. De asemenea, majoritatea proceselor se executa in kernel space.

Avantajele kernelului monolitic sunt urmatoarele :

- Viteza mare de prelucrare

- Dimensiune mica a sistemului

Dezavantajele kernelului monolitic sunt urmatoarele :

- Lipsa de portabilitate (fiecare kernel este scris pentru un anume tip de SO)

- Stabilitate redusa (o eroare a unui program poate conduce la erori succesive, ajungand pana la inchiderea sistemului de operare)

- Modificare greoaie a codului initial (la modificarea codului trebuie recompilat tot codul sursa)

- Dimensiunea mare a codului sursa

**In aceasta poza este prezentata impartirea functiilor intr-un kernel monolithic.**

Pe de alta parte, un microkernel prezinta doar functiile de baza ale kernelului, cum ar fi managmentul memoriei, managmentul proceselor. Celelalte functii sunt preluate de procese independente de kernel numite servere si care sunt legate printr-un sistem de mesaje. De asemenea, majoritatea proceselor sunt executate in user space.

Avantajele microkernelului sunt :

- Stabilitate crescuta (in cazul in care un server cade, restul nu sufera vreo modificare)

- Portabilitate

- Customizare usoara

Dezavantajele microkernelului sunt :

- Performanta scazuta datorata simplitatii kernelului si impartirii in servere

- Dimensiune mare a sistemului

**In aceasta poza este prezentata impartirea functiilor intr-un microkernel.**

In concluzie, microkernelul este destinat platformelor simple, cu un scop bine stabilit si nu foarte complex ; pe de alta parte, kernelul monolithic este avantajos pentru sistemele ce necesita diversitate si modularitate si in care se pune accent pe viteza de prelucrare.

**Capitolul 2. User space, kernel space si relatia din cele doua (+ System Call Interface)**

 Intr-un kernel, spatiul de memorie al acestuia poate fi impartit in 2 sectiuni: kernel space si user space. Aceasta diferentiere a fost facuta pentru a creste securitatea sistemului.

 User space se defineste ca fiindca zona de memorie in care se pastreaza informatii sau se executa programe care nu tin de kernel ; acestea au acces numai la zona lor de memorie si nu pot primi sau scrie cod in kernel space in mod normal.

Kernel space se defineste ca fiind zona de memorie in care este stocat kernelul si in care se executa serviciile/procesele acestuia. Este o zona protejata a sistemului, astfel incat kernelul sa nu poata fi modificat din greseala.

Pentru ca un proces aflat in user space sa poata accesa o resursa interzisa aflata in kernel space, trebuie folosita interfata System Call ce reprezinta un pod intre cele 2 spatii de memorie. Aceasta interfata poate fi comparata dupa rolul ei cu functiile publice ale unei clase private in C, ce ofera acces limitat la zonele private.

System Call se foloseste de intreruperi pentru a aduce in atentia kernelului o cerere de resursa, cum ar fi accesarea harddiskului, crearea de noi procese, folosirea unui anumit serviciu. Aceasta intrerupere are rolul de a pune CPU-ul intr-un regim privilegiat, ca apoi kernelul sa decida daca cererea poate sa fie acceptata sau nu ; daca a fost acceptata, kernelul executa instructiunile necesare si apoi readuce CPU-ul la starea initiala.

 System Call se poate imparti in mai multe categorii

 - Controlul proceselor : incarcare/executare/creare, etc.

 - Managmentul fisierelor : deschidere/stergere/creare, etc.

 - Managmentul dispozitivelor : citire/accesare, etc.

 - Informatia sistemului : citire data/setare proces, etc.

 - Communicare : trimitere/primire mesaj, etc.

**In aceasta figura este reprezentat modul in o aplicatie din User Space utilizeaza date din Kernel Space utilizand System Calls**

In concluzie, User Space reprezinta spatiul de memorie in care ruleaza aplicatii sau procese ale utilizatorului ce nu pot afecta functionarea normala a sistemului de operare ; accesul la Kernel Space, ce formeaza nucleul SO-ului si care trebuie protejat de date/comenzi eronate se face prin proceduri numite System Calls, al carui rol este de crea un pod intre User Space - Kernel Space astfel incat si programele utilizatorului sa aiba acces la resursele SO-ului, dar in acelasi sa nu aiba influente distructive asupra acestuia.

**Capitolul 3. Virtual file system**

Un sistem virtual de fisiere (il vom prescurta de acum VFS)constituie un nivel de abstractizare ce este pus peste un sistem de fisiere real. Acesta furnizeaza un set de interfete standard pentru ca aplicatiile sa poate accesa diferite tipuri de sisteme de fisier, fara ca aplicatiile sa stie de ce tip sunt acestea. De asemeneam acesta este facut astfel incat sa suporte simultan multiple sisteme de fisier in acelasi timp.

Spre exemplu, daca avem un sistem cu multiple partitii pe un hard disk, fiecare cu propriul ei format: una ext\* (extended file system - caracteristic unui sistem tipic cu Linux), una NTFS (New Technology File Sistem - avem instalat si Windows pentru dual boot) si, pe deasupra, mai conectam si un stick USB ce are formatul FAT32. Fara VFS, daca o aplicatie ar avea nevoie sa acceseze toate cele trei sisteme de fisiere, asta ar insemna ca cel ce dezvolta aplicatia ar trebui sa anticipeze toate sistemele de fisiere cu care aplicatia ar putea interactiona si sa creeze o interfata pentru fiecare. In mod evident, acest lucru reprezinta o munca excesiva si usor nonsensica, optandu-se, mai bine, pentru un nivel de abstractizare care sa permita accesul uniform. O interpretare vizuala foarte buna se poate observa in figura de mai jos.



Deasupra VFS este asezata interfata sistem-apel (de acum, prescurtata SCI - system-call interface), aceasta permitand apelurilor din user-space sa patrunda in kernel. Astfel, o aplicatie din user-space poate sa apeleze comanda POSIX (Portable Operating System Interface - familie de standarde introduse de IEEE pentru a mentine compatibilitatea intre sistemele de operare) *open* (apel de sistem folosit de majoritatea sistemelor de fisiere pentru a initia accesul la un fisier) ce va trece prin biblioteca GNU C (glibc) si va ajunge in kernel. VFS va fi apelat prin comanda *sys\_open*.

VFS furnizeaza un nivel de abstractizare pentru a separa API-ul (application programming interface) POSIX from the details of how a particular file system implements that behavior. Astfel, apelurile *open*, *read*, *write* sau *close* functioneaza la fel indiferent de sistemul de fisiere ce se afla la nivelul inferior. VFS furnizeaza un model comun de fisiere ce va trebui sa fie mostenit de acestea. De asemenea, un alt nivel de abstractizare din afara API-ului POSIX este folosit pentru a ascunde dispozitivul fizic.



**Arhitectura VFS**

Cele mai importante obiecte administrate dinamic de catre VFS sunt obiectele dentry si inode. Acestea sunt stocate in cache pentru a creste performantele accesului la sistemele de fisiere aflate la nivelul inferior. Cand un fisier este deschis, cache-ul dentry este populat cu intrari ce reprezinta nivelurile de director, acestea formand calea. Un inode este de asemenea creat pentru obiect, ce reprezinta fisierul.

Cache-ul dentry este construit folosind un hash table si este hashed prin numele obiectului. Intrarile pentru cache-ul dentry sunt alocate din dentry cache prin alocatorul “slab” si folosindu-se algoritmul least-recently-used pentru a elimina intrari cand exista probleme de memorie.



**Superblock**

Superblock-ul contine metadata de nivel inalt despre un sistem de fisiere, fiind o structura ce exista atat pe disc, cat si in memorie. Superblock-ul defineste parametrii de administrare ai sistemului de fisiere de pe disc (de ex:numarul total de blocuri, blocuri libere, root index node), astfel furnizand informatiile de baza pentru interactiunea cu acesta.

Pe disc, superblock-ul furnizeaza informatii kernel-ului despre structura sistemului de fisiere; iar, in memorie, furnizeaza informatiile necesare si starea pentru a putea administra sistemul activ de fisiere.

Deoarece Linux suporta multiple sisteme de fisiere simultan, fiecare structura spuer\_block este stocata intro lista(super\_blocks, defined in ./linux/fs/super.c, with the structure defined in /linux/include/fs/fs.h).

Mai jos, avem implementarea in cod C a obiectului superblock.

struct super\_block {
 struct list\_head s\_list; /\* list of all superblocks \*/
 dev\_t s\_dev; /\* identifier \*/
 unsigned long s\_blocksize; /\* block size in bytes \*/
 unsigned long s\_old\_blocksize; /\* old block size in bytes \*/
 unsigned char s\_blocksize\_bits; /\* block size in bits \*/
 unsigned char s\_dirt; /\* dirty flag \*/
 unsigned long long s\_maxbytes; /\* max file size \*/
 struct file\_system\_type s\_type; /\* filesystem type \*/
 struct super\_operations s\_op; /\* superblock methods \*/
 struct dquot\_operations \*dq\_op; /\* quota methods \*/
 struct quotactl\_ops \*s\_qcop; /\* quota control methods \*/
 struct export\_operations \*s\_export\_op; /\* export methods \*/
 unsigned long s\_flags; /\* mount flags \*/
 unsigned long s\_magic; /\* filesystem's magic number \*/
 struct dentry \*s\_root; /\* directory mount point \*/
 struct rw\_semaphore s\_umount; /\* unmount semaphore \*/
 struct semaphore s\_lock; /\* superblock semaphore \*/
 int s\_count; /\* superblock ref count \*/
 int s\_syncing; /\* filesystem syncing flag \*/
 int s\_need\_sync\_fs; /\* not-yet-synced flag \*/
 atomic\_t s\_active; /\* active reference count \*/
 void \*s\_security; /\* security module \*/
 struct list\_head s\_dirty; /\* list of dirty inodes \*/
 struct list\_head s\_io; /\* list of writebacks \*/
 struct hlist\_head s\_anon; /\* anonymous dentries \*/
 struct list\_head s\_files; /\* list of assigned files \*/
 struct block\_device \*s\_bdev; /\* associated block device \*/
 struct list\_head s\_instances; /\* instances of this fs \*/
 struct quota\_info s\_dquot; /\* quota-specific options \*/
 char s\_id[32]; /\* text name \*/
 void \*s\_fs\_info; /\* filesystem-specific info \*/
 struct semaphore s\_vfs\_rename\_sem; /\* rename semaphore \*/
};

###

### Lista super\_blocks

Toate superblocks-urile sunt adunate in lista *super\_blocks* cu legaturile aflate in campurile *s\_list*. Principalele functii se afla in super.c: *get\_super()* sau *user\_get\_super()*; acestea sunt folosite pentru a gasi superblock-ul unui anumit dispozitiv.

### Lista fs\_supers

Toate superblock-urile de un anumit tip sunt adunate intr-o lista headed de catre campul *fs\_supers* din structura *filesystem\_type*, cu legaturile aflate in campurile *s\_instances*.

### Lista file

Toate fisierele ce apartin unui superblock sunt inlantuite intr-o lista cu capul acesteia aflat in campul *s\_files* al superblock-ului, cu legaturile in campurile *f\_list* ale fisierelor. Aceasta lista este folosita, de exemplu, in *fs\_may\_remount\_ro()* pentru a verifica ca nu exista fisiere deschise pentru scriere.

Fiecare element din structura de mai jos reprezinta un pointer care o functie ce opereaza pe un obiect superblock. Operatiile de superblock executa operatii low-level asupra FS si inodu-urile acestuia.

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 \*);
 void (\*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 statfs \*);
 int (\*remount\_fs) (struct super\_block \*, int \*, char \*);
 void (\*clear\_inode) (struct inode \*);
 void (\*umount\_begin) (struct super\_block \*);
 int (\*show\_options)(struct seq\_file \*, struct vfsmount \*);
};

*struct inode \*alloc\_inode(struct super\_block \*sb)*
Aceasta functie creaza si initializeaza un nou obiect inode in cadrul superblock-ului.

*void destroy\_inode(struct inode \*inode)*

Aceasta functie dealoca inode-ul al carui pointer este pasat ca parametru.

*void read\_inode(struct inode \*inode)*

Aceasta functie citeste inode-ul specificat de pointer-ul *inode->i\_ino* de pe disc si umple restul structurii inode.

*void dirty\_inode(struct inode \*inode)*

Aceasta functie este chemata de catre VFS cand un inode este “murdarit” (modificat). Sistemele de fisier cu jurnal (precum ext3) folosesc aceasta functie pentru a-si actualiza jurnalul.

*void write\_inode(struct inode \*inode, int wait)*
Scrie inode-ul dat pe disc. Parametrul *wait* specifica daca operatia se face sau nu sincron.

*void drop\_inode(struct inode \*inode)*

Aceasta functie este apelata de catre VFS cand ultima referinta care un inode este dropped. In mod normal, sistemele de fisiere Unix nu definesc aceasta functie, caz in care VFS pur si simple sterge inode-ul.

*void delete\_inode(struct inode \*inode)*

Aceasta functie sterge inode-ul dat de pe disc.

*void put\_super(struct super\_block \*sb)*

Aceasta functie este apelata de catre VFS la unmount pentru a elibera obiectul superblock catre care avem pointer.

*void write\_super(struct super\_block \*sb)*

Aceasta functie actualizeaza superblock-ul de pe disc cu un superblock specificat. VFS foloseste aceasta functie pentru a sincroniza un superblock modificat in memorie cu discul.

*int sync\_fs(struct super\_block \*sb, int wait)*

Aceasta functie sincronizeaza metadata VFS cu metadata de sistemului de fisiere de pe disc.This function synchronizes filesystem metadata with the on-disk filesystem. Parametrul *wait* specifica daca operatia se face sau nu sincron.

*void write\_super\_lockfs(struct super\_block \*sb)*

Aceasta functie previne schimbarile aduse VFS, iar apoi actualizeaza spuerblock-ul de pe disc cu cel specificat. Functia este folosita de catre LVM (Logical Volume Manager).

*void unlockfs(struct super\_block \*sb)*
Aceasta functie deblocheaza VFS in urma schimbarilor, asemanator cu *write\_super\_lockfs()*.

*int statfs(struct super\_block \*sb, struct statfs \*statfs)*
Aceasta functie este apelata de catre VFS pentru a obtine statistici despre sistemul de fisiere. Acestea sunt stocate in *statfs*.

*int remount\_fs(struct super\_block \*sb, int \*flags, char \*data)*
Aceasta functie este apelata de catre VFS cand sistemul de fisiere este remounted cu noi optiuni de mount.

*void clear\_inode(struct inode \*)*

Aceasta functie este apelata de VFS pentru a elibera inode-ul si a curata pagini ce ce contin date legate de acesta.

*void umount\_begin(struct super\_block \*sb)*

Aceasta functie este apelata de catre VFS pentru a intrerupe operatie de mount. Este folosita de sisteme de fisiere de reteam, precum NFS.

**Index node (inode)**

Linux administreaza toate obiectele dinntr-un sistem de fisiere printr-un obiect numit *inode*. Acesta se poate referi la un fisier, un director sau o legatura simbolica la un alt obiect. A se nota faptul ca, fisierele fiind folosite sa reprezinte alte tipuri de obiecte (precum dispozitive sau memorii), inode-urile sunt, de asemenea, folosite pentru a le reprezenta.

In inode, este inclusa metadata la nivel de obiect precum: timpii de manipulare, datele de posesor si permisiuni (group-id, user-id, permisiuni). De asemenea, acesta are referinte catre operatiile ce sunt posibile asupra acestuia, majoritatea fiind o legatura directa care interfetele apel sistem (de exemplu: open, read, write si flush). Exista si o referinta catre operatii specifice inode-ului (create, lookup, link, mkdir, etc).

Mai exista o structura ce administreaza datele efective ale obiectului ce este prezentar de un obiect spatiu de adresa ( prescurtam ASO - address space object). Un ASO este un obiect ce administreaza diversele pagini pentru inode in interiorului cache-ului de pagina. El este folosit atat pentru administrarea paginilor pentru un fisier, cat si pentru maparea sectiunilor de fisiere intr-un spatiu de adresa de proces individual. ASO vine uneori cu propriul sau set de operatii (writepage, readpage, releasepage, etc.)

**Directory entry(dentry)**

Natura ierarhica a unui sistem de fisiere este administrata de un alt obiect in VFS numit obiect *dentry*. Un sistem de fisiere va avea un dentry root (referentiat in superblock), acesta fiind singurul dentry fara parinte. Celelate dentry-uri vor avea parinti si unele pot avea copii.

De exemplu, daca deschidem un fisier din directorul “/home/user/name”, vor fi create 4 obiecte dentry: unul pentru root-ul “/”, unul pentru intrarea “home” din directorul root, unul pentru intrarea “user” din directorul “home” , unul pentru intrarea “user” din directorul “home”. Atfel, maparile dentry se integreaza foarte bine in sistemele ierarhice folosite in prezent.

Obiectul dentry este definit in structura dentry (in ./linux/include/fs/dcache.h). Acesta este compus din mai multe elemente ce urmaresc relatia dintre doua intrari in sistemul de fisiere, precum si pentru datele fizice (cum ar fi numele fisierului).



Dentry are o referinta catre super\_block, ce defineste instanta sistemului de fisiere in care se afla acesta. Urmeaza parintele dentry-ului urmat de o lista ce contine copii acestuia (in caz ca obiectul este un director). Apoi sunt definite operatiilor posibile ale unui dentry (ex: hash, compare, delete, release). Urmeaza numele numele obiectului, ce este in dentry si nu in inode. Finalmente, este pusa o referinta catre inode-ul VFS.

**File object**

Pentru fiecare fisier deschis intr-un sistem Linux exista un obiect fisier. Acesta contine informatii specifice instantei deschise pentru un utilizator.



O structura path furnizeaza referinta atat catre dentry, cat si catre vfsmount. Un set de operatii de fisier este definit pentru fiecare fisier(ex: open, close, read, write, flush, etc.). Un set de flag-uri si permisiuni este definit (inclusiv grup si posesor.)

**Relatia intre obiecte**

In partea superioara a imaginii se afla obiectul fisier deschis, ce este referentiat de lista descriptor de fisier a unui proces. Obiectul fisier face o referinta la un dentry, care - la randul lui - face referinta la un inode. Atat dentry-ul cat si inode-ul au o referinta catre obiectul super-block-ul inferior. Mai multe obiecte fisier pot face referinta la acelasi dentry (precum in cazul celor doi utilizatori ce impart acelasi fisier).

A se observa ca, in figura, un obiect dentry poate face referinta la un alt obiect dentry. In acest caz, un director face referinta la un fisier, care - la randul lui - face referinta catre inode-ul unui anumit fisier.



**Remarci finale**

Cu toate ca abordarea nu este una aprofundata, se poate vedea cum subsistemele Linux (in cazul de fata, VFS) face ca Linux sa fie scalabil, flexibil si extensibil: prin nivelul de abstactizare introdus de VFS, care elimina necesitatea scrierii de cod pentru fiecare aplicatie in parte in cazul accesarii diverselor tipuri de sisteme de fisiere.

**Capitolul 4. Process Management**

Linux este un sistem foarte dinamic cu necesitati care se schimba mereu. Reprezentarea nevoilor computationale ale sistemului Linux se exprima prin procese. Acestea pot dura foarte putin sau foarte mult, trebuie impartite in threaduri si trebuie sa respecte arhitectura multiuser a Linuxului fara a compromite securitatea sistemului.

In spatiul utilizatorilor procesele se pot crea in mai multe moduri. Se poate executa un program (fapt ce rezulta in crearea unui proces) sau se poate invoca o bifurcatie sau un apel de sistem in cadrul unui program. Bifurcatia duce la creerea unui proces copil, iar un apel de executie inlocuieste procesul curent in contextul noului program.

**Reprezentarea proceselor:**

In cadrul kernelului Linux, un proces e reprezentat de o structura mare numita *task\_struct*. Structura contine toate datele necesare pentru a reprezenta procesul impreuna cu o multe de alte date pentru a mentine relatiile cu alte procese. Exemplu succint al structurii task\_struct:

struct task\_struct {

 volatile long state;
 void \*stack;
 unsigned int flags;

 int prio, static\_prio;

 struct list\_head tasks;

 struct mm\_struct \*mm, \*active\_mm;

 pid\_t pid;
 pid\_t tgid;

 struct task\_struct \*real\_parent;

 char comm[TASK\_COMM\_LEN];

 struct thread\_struct thread;

 struct files\_struct \*files;

 ...

};

In codul de mai sus se pot observa: starea executiei, o stiva, un set de fanioane, procesul parinte, threadul de executie si fisierele deschise.

Variabila de stare este un set de biti care indica starea procesului. Cuvantul flags devineste un numar mare de fanioane care indica multe aspecte ale procesului precum: daca procesul este creat (PF\_STARTING) sau daca se termina (PF\_EXITING), sau daca procesul aloca memorie (PF\_MEMALLOC).

Fiecare proces are o prioritate proprie (static\_prio), dar prioritatea adevarata in sistem este decisa dinamic de catre sistemul de operare. Campul *tasks* (procese) permite abilitatea de liste inlantuite. Contine un pointer prev (anterior) si un pointer next (urmator).

Spatiul de adrese al procesului este reprezentat de campurile *mm* ce reprezinta descriptorii de memorie ai procesului si *active\_mm* reprezinta descriptorii de memorie ai procesului anterior.

*Thread\_struct* reprezinta structura firelor de executie si difera de la o arhitectura Linux la alta.

**Managementul de procese:**

In majoritatea cazurilor, procesele sunt create dinamic si sunt reprezentate de o *task\_struct* dinamica. Singura exceptie este procesul *init* in sine care exista mereu si e reprezentat static.

Toate procesele in Linux sunt colectate in doua moduri diferite: prin intermediul unui tabel hash sau de o lista circulara bidirectionala care este ideala pentru parcurgerea iterativa a listei de procese. Datorita faptului ca lista este circulara, nu exista cap sau coada, dar datorita faptului da *init\_task* mereu exista, se poate folosi ca ancora pentru urmatoarea iteratie.

Lista de procese nu este accesibila din spatiul utilizatorilor dar se poate insera cod in kernel sub forma de modul daca utilizatorul are drept de administrator. Un exemplu de un astfel de modul:

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>

int init\_module( void )
{
 /\* Set up the anchor point \*/
 struct task\_struct \*task = &init\_task;

 /\* Walk through the task list, until we hit the init\_task again \*/
 do {

 printk( KERN\_INFO "\*\*\* %s [%d] parent %s\n",
 **task->comm**, **task->pid**, **task->parent->comm** );

 } while ( (task = **next\_task**(task)) != &init\_task );

 return 0;

}

void cleanup\_module( void )
{
 return;
}

Acest modul se poate compila cu fisierul de creere urmator:

obj-m += procsview.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

default:
 $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules

Dupa compilare, modulul se poate insera in kernel cu insmod procsview.ko si se poate scoate din kernel cu rmmod procsview. Linux mentine un simbol numit “current” care este procesul curent de rulare (de tip task\_struct).

**Crearea de procese:**

Mecanismul de creare de procese este acelasi pentru procesele in spatiul utilizatorilor si pentru procesele kernelului deoarece ambele se bazeaza pe o functie numita *do\_fork* pentru crearea de noi procese. In contextul creeri unui thread de kernel, kernelul apeleaza o functie numita *kernel\_thread* care initializeaza, iar apoi apeleaza *do\_fork*.

O actiune similara are loc pentru crearea de procese in spatiul userilor. In spatiul userilor, un program apeleaza fork, care rezulta intrun apel de sistem la functia kernelului numita *sys\_fork*. Relatia intre functii este reprezentata grafic de catre urmatorul grafic:



Functia *do\_fork* incepe printrun apel la *alloc\_pidmap* care aloca un nou PID (identificator de proces). *Do\_fork* verifica sa vada daca debuggerul urmeaza procesul parinte. In cazul in care e, fanionul CLONE\_PTRACE este setat in *clone\_flags* in pregatire pentru bifurcatie. Functia *do\_fork* apeleaza in continuare *copy\_process*, transferand fanioanele, stiva, registrele, procesele parinte si noul alocat PID.

Functia *copy\_process* este functia unde noul proces este creat ca o copie al parintelui. Aceasta functie efectueaza toate actiunile cu exceptia procesului de start. Primul pas in *copy\_process* este validarea fanioanelor CLONE pentru a asigura ca sunt consistenti. Daca nu sunt, este returnata o eroare EINVAL. In continuare, LSM (modulul de securitate Linux) este consultat pentru a vedea daca procesul curent poate crea un nou proces.

In continuare, functia *dup\_task\_struct* este apelata, care aloca un nou *task\_struct* si copiaza descriptorii procesului curent in acesta. Dupa ce se seteaza o noua stiva de threaduri, anumite informatii de stare sunt initializate si este returnat controlul asupra lui *copy\_process*. O secventa de functii de copiere este invocata si se copiaza individual aspecte ale procesului precum copierea descriptorilor fisierului deschis (copy\_files), copierea informatiilor despre semnal (copy\_sighand si copy\_signal), copierea memoriei de procese (copy\_MM) si copierea threadului.

Noul proces este desemnat unui procesor si anumite verificari sunt facute in functie de procesoarele pe care se poate executa procesul (cpus\_allowed). Dupa ce prioritatea noului proces mosteneste prioritatea parintelui, se efectueaza o cantitate mica de intretinere si controlul revine la *do\_fork*. La acest punct, noul proces exista dar nu ruleaza. Functia *do\_fork* apeleaza *wake\_up\_new\_task*. Aceasta functie plaseaza noul proces intro coada de rulare, apoi il trezeste pentru executie. In urma revenirii la *do\_fork*, valoarea PID este returnata apelantului si procesul se incheie.

**Planificarea proceselor**

Un proces in Linux poate fi programat prin planificatorul Linux. Acesta mentine un set de liste pentru fiecare nivel de prioritate unde se afla *task\_struct*. Procesele sunt invocate prin functia de planificare, care determina cele mai bune procese care ruleaza bazate pe istoria anterioara de incarcare si executie.

**Distrugerea proceselor**

Distrugerea proceselor este determinata de anumite evenimente: terminare formala de proces, terminare prin semnal, sau terminare printro apelarea unei functiei *exit*. Indiferent, procesul se termina printrun apel la functia kernelului *do\_exit*. Acest proces este afisat grafic in urmatoarea figura:

 

Scopul lui *do\_exit* este de a inlatura toate referintele la procesul curent de catre sistemul de operare. Procesul de terminare de proces indica intai faptul ca procesul se termina prin setarea fanionului PF\_EXITING. Alte aspecte ale kernelului folosesc aceasta indicatie pentru a evita manipularea procesului in timp ce este invocat. Detasarea procesului de diferite resurse obtinute de acesta in timpul vietii se face printro serie de apeluri, inclusiv *exit\_mm* (pentru a inlatura paginile de memorie) si *exit\_keys* (care elimina cheile de securitate pentru fiecare thread in care este implicat procesul). Functia *do\_exit* tine seama de parametrii pentru distrugerea procesului, apoi o serie de notificari sunt efectuate printrun apel la *exit\_notify*. La sfarsit, starea procesului este schimbata la PF\_DEAD, si functia de planificare este apelata pentru a selecta un nou proces pentru executie.

**Bibilografie**

[**http://www.linfo.org/kernel.html**](http://www.linfo.org/kernel.html)

[**http://www.linux-arm.org/pub/LinuxKernel/WebHome/gorman\_linux\_vm.pdf**](http://www.linux-arm.org/pub/LinuxKernel/WebHome/gorman_linux_vm.pdf)

[**http://cs-pub.bu.edu/fac/richwest/cs591\_w1/notes/wk1.pdf**](http://cs-pub.bu.edu/fac/richwest/cs591_w1/notes/wk1.pdf)

[**http://linuxkernel51.blogspot.ro/2011/02/difference-between-microkernel-and.html**](http://linuxkernel51.blogspot.ro/2011/02/difference-between-microkernel-and.html)

[**http://www.ibm.com/developerworks/library/l-linux-kernel/**](http://www.ibm.com/developerworks/library/l-linux-kernel/)

[**http://www.ibm.com/developerworks/library/l-linux-process-management/**](http://www.ibm.com/developerworks/library/l-linux-process-management/)

[**http://www.ibm.com/developerworks/library/l-virtual-filesystem-switch/l-virtual-filesystem-switch-pdf.pdf**](http://www.ibm.com/developerworks/library/l-virtual-filesystem-switch/l-virtual-filesystem-switch-pdf.pdf)

[**http://www.makelinux.net/books/lkd2/ch12lev1sec5**](http://www.makelinux.net/books/lkd2/ch12lev1sec5)

[**http://www.win.tue.nl/~aeb/linux/lk/lk-8.html**](http://www.win.tue.nl/~aeb/linux/lk/lk-8.html)