

4 Sistemul de fişiere Unix

4.1 Structura arborescentă și legături suplimentare

4.1.1 Tipuri de fişiere și sisteme de fişiere

În cadrul unui sistem de fişiere, apelurile sistem Unix gestionează opt tipuri de fişiere și anume:

1. Normale (obișnuite)
2. Directori
3. Legături hard (hard links)
4. Legături simbolice (symbolic links)
5. Socketuri (sockets)
6. FIFO - pipe cu nume (named pipes)
7. Periferice caracter
8. Periferice bloc

Pe lângă aceste opt tipuri, mai există încă patru entități, pe care apelurile sistem le văd, din punct de vedere sintactic, tot ca și fişiere. Aceste entități sunt gestionate de nucleul Unix, au suportul fizic tot în nucleu și folosite la comunicări între procese. Aceste entități sunt:

9. Pipe (anonymous pipes)
10. Segmente de memorie partajată
11. Cozi de mesaje
12. Semafoare

În acest capitol, sau în cele care urmează, vom trata aceste entități, exceptând 5, 10, 11, și 12 care vor face obiectul unei lucrări viitoare, destinate special sistemelor de operare distribuite.

Fişierele obișnuite sunt privite ca șiruri de octeți, accesul la un octet putându-se face fie secvențial, fie direct prin numărul de ordine al octetului.

Fişierele directori. Un fișier director se deosebește de un fișier obișnuit numai prin informația conținută în el. Un director conține lista de nume și adrese pentru fişierele subordonate lui. Uzual, fiecare utilizator are un director propriu care punctează la fişierele lui obișnuite, sau la alți subdirectori definiți de el.

Fişierele speciale. În această categorie putem include, pentru moment, ultimele 6 tipuri de fişiere. În particular, Unix privește fiecare dispozitiv de I/O ca și un fișier de tip special. Din punct de vedere al utilizatorului, nu există nici o deosebire între lucrul cu un fișier disc normal și lucrul cu un fișier special.

Fiecare director are două intrări cu nume speciale și anume:

- " ." (punct) denumește generic (punctează spre) însuși directorul respectiv;
- " .. " (două puncte succesive), denumește generic (punctează spre) directorul părinte.

Fiecare sistem de fişiere conține un director principal numit **root** sau **/**.

În mod obișnuit, fiecare utilizator folosește un *director curent*, atașat utilizatorului la intrarea în sistem. Utilizatorul poate să-și schimbe acest director (`cd`), poate crea un nou director subordonat celui curent, (`mkdir`), să șteargă un director (`rmdir`), să afișeze *calea* de acces de la `root` la un director sau fișier (`pwd`) etc.

Apariția unui mare număr de distribuitori de Unix a condus, inevitabil, la proliferarea unui număr oarecare de "*sisteme de fișiere extinse*" proprii acestor distribuitori. De exemplu:

- Solaris utilizează sistemul de fișiere `ufs`;
- Linux utilizează cu precădere sistemul de fișiere `ext2` și mai nou, `ext3`;
- IRIX utilizează `xfs`
- etc.

Actualele distribuții de Unix permit utilizarea unor sisteme de fișiere proprii altor sisteme de operare. Printre cele mai importante amintim:

- Sistemele FAT și FAT32 de sub MS-DOS și Windows 9x;
- Sistemul NTFS propriu Windows NT și 2000.

Din fericire, aceste extinderi sunt transparente pentru utilizatorii obișnuiți. Totuși, se recomandă prudență atunci când se efectuează altfel de operații decât citirea din fișierele create sub alte sisteme de operare decât sistemul curent. De exemplu, modificarea sub Unix a unui octet într-un fișier de tip `doc` creat cu Word sub Windows poate ușor să compromită fișierul așa încât el să nu mai poată fi exploatat sub Windows!

Administratorii sistemelor Unix trebuie să țină cont de sistemele de fișiere pe care le instalează și de drepturile pe care le conferă acestora vis-a-vis de userii obișnuiți.

Principiul structurii arborescente de fișiere este acela că orice fișier sau director are un singur părinte. Automat, pentru fiecare director sau fișier există o singură cale (path) de la rădăcină la directorul curent. Legătura între un director sau fișier și părinte o vom numi *legătură naturală*. Evident ea se creează odată cu crearea directorului sau fișierului respectiv.

4.1.2 Legături hard și legături simbolice

În anumite situații este utilă partajarea unei porțiuni a structurii de fișiere între mai mulți utilizatori. De exemplu, o bază de date dintr-o parte a structurii de fișiere trebuie să fie accesibilă mai multor utilizatori. Unix permite o astfel de partajare prin intermediul *legăturilor suplimentare*. O legătură suplimentară permite referirea la un fișier pe alte căi decât pe cea naturală. Legăturile suplimentare sunt de două feluri: *legături hard* și *legături simbolice (soft)*.

Legăturile hard sunt identice cu legăturile naturale și ele pot fi create numai de către administratorul sistemului. O astfel de legătură este o intrare într-un director care punctează spre o substructură din sistemul de fișiere spre care punctează deja legătura lui naturală. Prin aceasta, substructura este văzută ca fiind descendentă din două directoare diferite! Deci, printr-o astfel de legătură un fișier primește efectiv două nume. Din această cauză, la parcurgerea unei structuri arborescente, fișierele punctate prin legături hard apar duplicate. Fiecare duplicat apare cu numărul de legături către el.

De exemplu, dacă există un fișier cu numele `vechi`, iar administratorul dă comanda:

```
#ln vechi linknou
```

atunci în sistemul de fișiere se vor vedea două fișiere identice: vechi și linknou, fiecare dintre ele având marcat faptul că sunt două legături spre el.

Legăturile hard pot fi făcute numai în interiorul aceluiași sistem de fișiere (detalii puțin mai târziu).

Legăturile simbolice sunt intrări speciale într-un director, care punctează (referă) un fișier (sau director) oarecare în structura de directori. Această intrare se comportă ca și un subdirector al directorului în care s-a creat intrarea.

În forma cea mai simplă, o legătură simbolică se creează prin comanda:

```
ln -s caleInStructuraDeDirectori numeSimbolic
```

După această comandă, *caleInStructuraDeDirectori* va avea marcată o legătură în plus, iar *numeSimbolic* va indica (numai) către această cale. Legăturile simbolice pot fi utilizate și de către utilizatorii obișnuiți. De asemenea, ele pot puncta și înafara sistemului de fișiere (detalii puțin mai târziu).

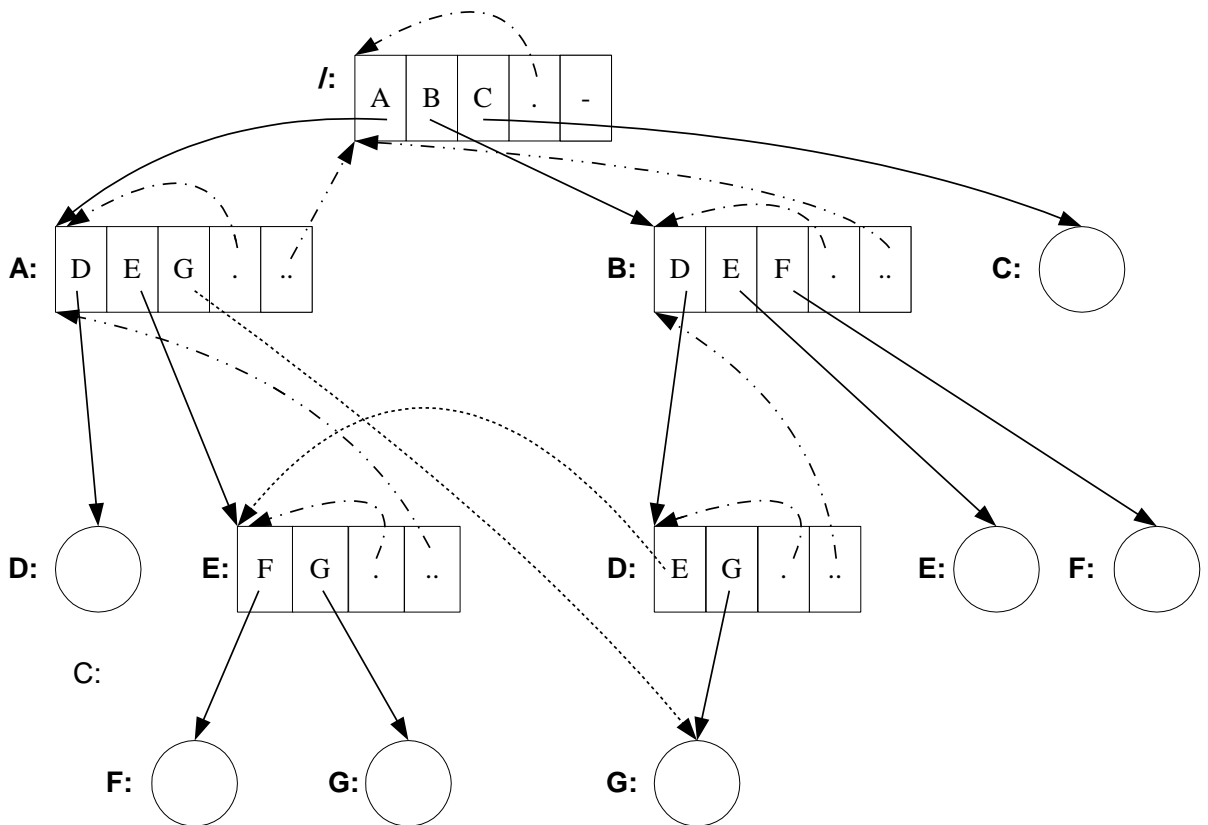


Figura 4.1 O structură arborescentă cu legături

Structura arborescentă împreună cu legăturile simbolice sau hard conferă sistemului de fișiere Unix o structură de graf aciclic. În fig. 4.1 este prezentat un exemplu simplu de structură de fișiere. Prin literele mari A, B, C, D, E, F, G am indicat nume de fișiere obișnuite, nume de directori și nume de legături. Este evident posibil ca același nume să apară de mai multe ori în structura de directori, grație structurii de directori care elimină ambiguitățile. Fișierele obișnuite sunt reprezentate prin cercuri, iar fișierele directori prin dreptunghiuri.

Legăturile sunt reprezentate prin săgeți de trei tipuri:

- linie continuă – legăturile naturale;
- linie întreruptă – spre propriul director și spre părinte;
- linie punctată – legături simbolice sau hard.

În fig. 4.1 există 12 noduri - fişiere obișnuite sau directori. Privit ca un arbore, deci considerând numai legăturile naturale, el are 7 ramuri și 4 nivele.

Să presupunem că cele două legături (desenate cu linie punctată) sunt simbolice. Pentru comoditate, vom nota legătura simbolică cu ultima literă din specificarea căii. Crearea celor două legături se poate face, de exemplu, prin succesiunea de comenzi:

```
cd /A
ln -s /A/B/D/G G Prima legătură
cd /A/B/D
ln -s /A/E E A doua legătură
```

Să presupunem acum că directorul curent este B. Vom parcurge arborele în ordinea director urmat de subordonații lui de la stânga spre dreapta. Următoarele 12 linii indică toate cele 12 noduri din structură. Pe aceeași linie apar, atunci când este posibil, mai multe specificări ale aceluiași nod. Specificările care fac uz de legături simbolice sunt subliniate. Cele mai lungi 7 ramuri vor fi marcate cu un simbol # în partea dreaptă.

```
/ ..
/A ../A
/A/D ../A/D #
/A/E ../A/E D/E ./D/E
/A/E/F ../A/E/F D/E/F ./D/E/F #
/A/E/G ../A/E/G D/E/G ./D/E/G #
/B .
/B/D D ./D
/B/D/G D/G ./D/G /A/G ../A/G #
/B/E E ./E #
/B/F F ./F #
/C ../C #
```

Ce se întâmplă cu ștergerea în cazul legăturilor multiple? De exemplu, ce se întâmplă când se execută una dintre următoarele două comenzi?

```
rm D/G
rm /A/G
```

Este clar că fişierul trebuie să rămână activ dacă este șters numai de către una dintre specificări.

Pentru aceasta, în descriptorul fişierului respectiv există un câmp numit *contor de legare*. Acesta are valoarea 1 la crearea fişierului și crește cu 1 la fiecare nouă legătură. La ștergere, se radiază legătura din directorul părinte care a cerut ștergerea, iar contorul de legare scade cu 1. Abia dacă acest contor a ajuns la zero, fişierul va fi efectiv șters de pe disc și blocurile ocupate de el vor fi eliberate.

4.1.3 Conceptul de montare

Spre deosebire de alte sisteme de operare ca DOS, Windows, etc. în specificarea fișierelor Unix *nu apare zona de periferic*. Acest fapt nu este întâmplător, ci este cauzat de filozofia generală de acces la fișierele Unix. Conceptul esențial prin care se rezolvă această problemă este cunoscut în Unix prin termenii de *montare* și *demontare* a unui sistem de fișiere.

Operația de *montare* constă în conectarea unui sistem de fișiere, de pe un anumit disc, la un director existent pe sistemul de fișiere implicit. Administratorul lansează comanda de montare sub forma:

```
# mount [ optiuni ] sistemDeFișiere directorDeMontare
```

Efectul este conectarea indicată prin `sistemDeFișiere` la `directorDeMontare` existent pe sistemul implicit de fișiere. Opțiunile pot să indice caracteristicile montării. De exemplu opțiunea `rw` permite atât citirea, cât și scrierea în subsistemul montat, în timp ce opțiunea `ro` permite numai citirea din subsistemul montat. Opțiunea `-t` indică tipul sistemului de fișiere care se montează, și în funcție de tip, argumentul `sistemDeFișiere` poate fi `/dev/periferic` sau `dev/dsk/periferic` sau `/root/periferic` ș.a.m.d. Pentru detalii se va putea consulta manualul `mount` al sistemului de operare curent.

Operația de *demontare* are efectul invers și ea se face cu comanda:

```
#/etc/unmount directorDeMontare
```

Să urmărim cele ilustrate în fig. 4.2. În fig. 4.2a este dată structura sistemului de fișiere activ. Directorul B este vid (nu are descendenți). În fig 4.2b este dată structura de fișiere de pe un disc aflat (să zicem) pe unitatea de disc nr. 3, cu care intenționăm să lucrăm. Pentru aceasta, se va da comanda *privilegiată* Unix:

```
#/etc/mount /dev/fd3 /B
```

Prin ea se indică legarea discului al cărui fișier special (driver) poartă numele `/dev/fd3` (și care se referă la discul nr. 3), la directorul `/B`. Urmare a legării, se obține structura de fișiere activă din fig. 4.2c.

Cu scuzele de rigoare față de cititorul pe care-l plictisim, vom face câteva precizări. Deși ele sunt firești, practica arată că sunt de multe ori încălcate, ceea ce provoacă neplăceri (și nu numai atât).

- Nu se va cere accesul la un fișier de pe un disc decât dacă acesta este montat în structura de fișiere implicită.
- Nu se va cere demontarea unei substructuri decât dacă a fost montată în prealabil.
- *Scoaterea unui suport montat de pe o unitate și înlocuirea lui cu un alt suport poate provoca pagube mari*. Să presupunem, spre exemplu, că s-a montat o structură existentă pe o anumită dischetă. Dacă înaintea demontării se scoate discheta din unitate și se introduce alta în loc, atunci este posibil să se piardă informații de pe noua dischetă, și, în unele cazuri, este posibilă blocarea întregului sistem! De altfel, sistemele Unix, mai nou, nici nu permit scoaterea din unitate a unui suport până când nu se efectuează operația `unmount`.

- Nu este posibilă efectuarea de legături simbolice decât dacă directorul de unde se leagă și fişierul / directorul care se leagă se află pe același suport fizic. (Deși pe unele sisteme s-ar putea ca o astfel de legare să fie posibilă, noi nu o recomandăm :(

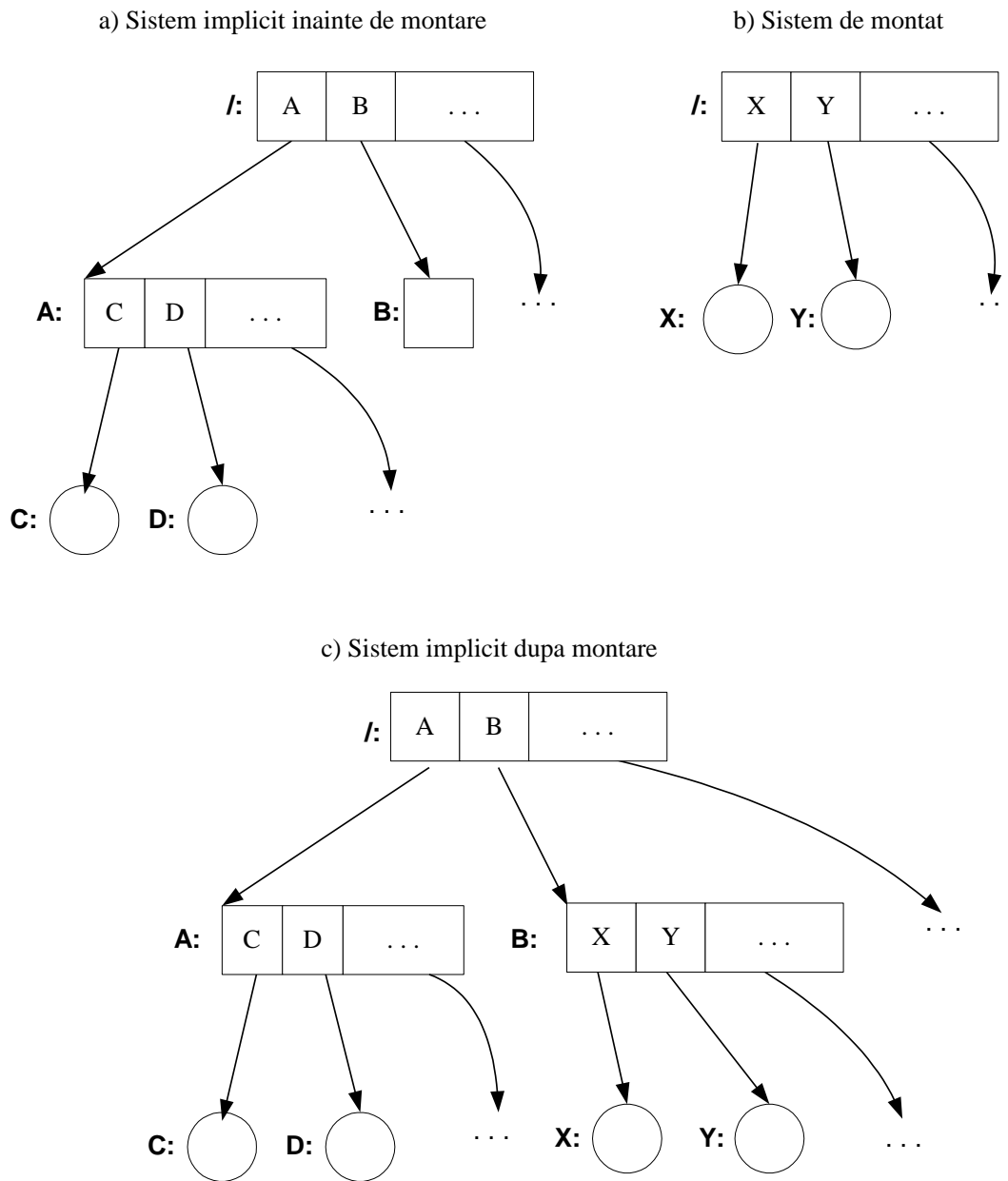


Figura 4.2 Operația de montare

În practica implementărilor Unix, la încărcarea **SO** se fac automat o serie de operații de montare, în conformitate cu configurarea sistemului. Indicațiile de montare automată sunt trecute în fişierul `/etc/fstab`. Acesta este un fişier text, având pe fiecare linie câte o montare, în care se specifică:

- partiția Unix (`periferic`, `sistemDeFişiere`) care se montează;
- `directorDeMontare` sub care este montată partiția;
- tipul sistemului de fişiere conţinut;
- diverse opțiuni.

Iată, spre exemplu, o porţiune din acest fişier:

```
/dev/hda2      /                ext3      defaults          1 1
/dev/hda7      /home            ext3      defaults,nosuid,nodev,noexec
/dev/cdrom     /mnt/cdrom       iso9660   noauto,owner,ro  0 0
/dev/fd0       /mnt/floppy      auto      noauto,owner     0 0
/dev/hda3      /usr             ext3      defaults,nodev    1 2
/dev/hda5      /usr/local       ext3      defaults,nodev    1 2
/dev/hda6      /var             ext3      defaults,nosuid,nodev,noexec
/dev/hda1      swap            swap      defaults          0 0
```

De asemenea, operaţia de montare este practică pentru *înglobarea SO Unix într-o reţea de calculatoare*. Cele mai cunoscute astfel de sisteme care dirijează montările de fişiere în reţele Unix (şi nu numai) sunt *NFS (Network File System* al firmei Sun Microsystems), *RFS (Remote File System* al firmei AT&T), *SAMBA* (produs open source). Sistemul care montează o structură de directoare de pe o altă maşină poartă numele de *client* (NFS, RFS, SAMBA, etc.). Maşina care oferă structura spre montare (exportă) se numeşte *server* (NFS, RFS, SAMBA, etc.).

Montările pentru NFS sunt specificate tot în fişierul `/etc/fstab`, prin linii de forma:

```
server1:/export/home          /home          nfs      rw,bg,intr    0 0
server1:/export/usr/local     /usr/local     nfs      rw,bg,intr    0 0
server2:/export/var/spool/mail /var/spool/mail nfs      rw,bg,intr    0 0
```

În acest context se poate da o nouă caracterizare a diferenţelor dintre legăturile hard şi cele simbolice: legăturile hard funcţionează numai în interiorul aceluiaşi sistem de fişiere, în timp ce legăturile simbolice pot puncta şi spre noduri ale altui sistem de fişiere montat împreună cu sistemul de fişiere ce conţine legătura limbolică.

4.1.4 Protecţia fişierelor Unix

4.1.4.1 Drepturi de acces

Reamintim iarăşi principalele entităţi participante la Unix: utilizatori, fişiere, procese şi mai ales reamintim faptul că între ele există multe interdependenţe. În această secţiune vom trata una dintre aceste interdependenţe.

Vis a vis de un fişier sau de un director, utilizatorii se împart în trei categorii:

- proprietarul fişierului (*u - user*).
- grupul de utilizatori (*g - group*), de exemplu o grupă de studenţi participă la un acelaşi proiect, motiv pentru care administratorul poate constitui un astfel de grup, cu drepturi specifice – de regulă mai slabe decât ale proprietarului, dar mai puternice decât ale restului utilizatorilor.
- restul utilizatorilor (*o - others*) cei care nu sunt în primele două categorii.

Nucleul SO Unix identifică utilizatorii prin numere naturale asociate unic, numite UID-uri (User IDentifications). De asemenea, identifică grupurile de utilizatori prin numere numite GID-uri (Group IDentifications).

Un utilizator aparte, cu drepturi depline asupra tuturor fișierelor este *root* sau *superuserul*.

Pentru fiecare categorie de utilizatori, fișierul permite maximum trei drepturi:

- dreptul de citire (*r* – *read*)
- dreptul de scriere (*w* – *write*) care include crearea de subdirectori, stergerea de subdirectori, adăugarea sau ștergerea de intrări în director, modificarea fișierului, etc.
- dreptul de execuție (*x* - *execution*) care permite lansarea în execuție a unui fișier. Acest drept, conferit unui director, permite accesul în directorul respectiv (*cd*).

În consecință, pentru specificarea drepturilor de mai sus asupra unui fișier sau director sunt necesari 9 (nouă) biți. Reprezentarea externă a acestei configurații se face printr-un grup de 9 (nouă) caractere: *rwXrwxrwx* în care absența unuia dintre drepturi la o categorie de useri este indicată prin – (minus).

Modul de atribuire a acestor drepturi se poate face cu ajutorul comenzii Shell *chmod*. Cea mai simplă formă a ei este:

```
chmod o1o2o3 fisier . . .
```

unde *o₁o₂o₃* sunt trei cifre octale. Biții 1 ai primeia indică drepturile userului, biții 1 ai celei de-a doua indică drepturile grupului, iar biții 1 ai celei de-a treia cifră indică drepturile restului userilor. De exemplu, comanda:

```
chmod 754 A B
```

fixează la fișierele *A* și *B* toate drepturile pentru user, drepturile de citire și de execuție pentru grup și doar dreptul de citire pentru ceilalți useri. În urma acestei comenzi, drepturile fișierelor *A* și *B* vor deveni: *rwXr-xr--*

Invităm cititorul să consulte manualul comenzii *chmod* pentru prezentarea completă a acestei comenzi.

4.1.4.2 Drepturi implicite: *umask*

În momentul intrării unui user în sistem, acestuia i se vor acorda niște drepturi implicite pentru fișiere nou create. Toate fișierele și directoarele create de user pe durata sesiunii de lucru îl vor avea ca proprietar, iar drepturile vor fi cele primite implicit. Fixarea drepturilor implicite, sau aflarea valorii acestora se poate face folosind comanda *umask*. Drepturile implicite sunt stabilite scăzând (octal) masca definită prin *umask* din *777*.

Pentru a se afla valoarea măștii se lansează:

```
umask
```

Rezultatul este afișarea măștii, de regulă *022*. Deci drepturile implicite vor fi: *777-022=755*, adică userul are toate drepturile, iar grupul și restul vor avea doar drepturi de citire și de execuție.

Dacă se doreşte schimbarea acestei măşti pentru a da userului toate drepturile, grupului de citire şi execuţie iar restului nici un drept, adică drepturile implicite 750, fiindcă $777-027=750$. Deci se va da comanda:

```
umask 027
```

Efectul ei va rămâne valabil până la un nou `umask` sau până la încheierea sesiunii.

4.1.4.3 Drepturi de lansare, drepturi program executabil, biţii `setuid` şi `setgid`

În această secţiune vom analiza, din punct de vedere al drepturilor de acces, relaţia dintre un utilizator, programul executabil pe care îl lansează şi fişierele asupra cărora acţionează programul în cursul execuţiei lui.

Pentru fixarea ideilor, să considerăm un exemplu. Presupunem că:

1. un utilizator U , care face parte dintr-un grup G , lansează în execuţie un program P . Pe durata execuţiei, programul P acţionează asupra unui fişier F .
2. programul P are ca proprietar utilizatorul UP care face parte din grupul GP , iar drepturile fişierului executabil P sunt `rwxr-xr-x`.
3. fişierul F asupra căruia se acţionează are proprietarul UF care face parte din grupul GF , iar drepturile fişierului F sunt `rwxr-xr--`.

În aceste ipoteze, userul U poate să lanseze în execuţie programul P , deoarece acesta conferă drepturi de execuţie tuturor utilizatorilor. (Dacă drepturile lui P ar fi fost `rwxr--r--`, atunci lansarea în execuţie ar fi fost posibilă numai dacă $U = UP$. Dacă drepturile lui P ar fi fost `rwxr-xr--`, atunci lansarea ar fi fost posibilă numai dacă $U = UP$ sau $G = GP$.)

După lansarea în execuţie de către U a lui P , în mod implicit acţiunile pe care programul P le poate efectua asupra fişierului F sunt cele permise de drepturile pe care U le are asupra lui F . În exemplul nostru, dacă $U = UF$ atunci P poate citi, scrie sau executa F . Dacă $G = GF$ atunci P poate citi sau executa F , iar dacă G şi GF sunt diferite atunci P poate numai să citească din F . (Într-o secţiune următoare vom arăta cum un program poate să lanseze în execuţie un alt program).

Această regulă, prin care drepturile lansatorului de program permit sau nu unele operaţii pe care programul lansat le aplică unui fişier, este regula cvasigenerală de acţiune asupra fişierelor.

Există însă situaţii, nu foarte frecvente, în care această regulă se poate schimba. Schimbarea este cunoscută sub numele *setuid* / *setgid* şi se referă, cu notaţiile de mai sus, la faptul că: După lansarea în execuţie de către U a lui P , în mod setuid / setgid acţiunile pe care programul P le poate efectua asupra fişierului F sunt cele permise de drepturile proprietarului programului P şi ale grupului din care face parte acesta.

Este vorba de doi biţi importanţi relativ la drepturile de acces, aşa numiţii *bit setuid* (*set-user-id*) care pus pe 1 schimbă drepturile lui U cu drepturile lui UP şi *bitul setgid*, care pus pe 1 schimbă drepturile lui G cu drepturile lui GP . (Aceste schimbări au loc numai la execuţia programului P .)

Cu alte cuvinte, dacă pentru un fișier executabil bitul `setuid` este 1, atunci **un utilizator care lansează în execuție acest fișier** (evident, dacă are dreptul să-l lanseze) **primește, pe timpul execuției, aceleași drepturi de acces la resurse** (fișiere, semafoare, zone de memorie etc.) **ca și proprietarul fișierului executabil.**

Să vedem o situație concretă în care este utilă folosirea bitului `setuid`. Un utilizator cu numele **profesor** întreține un fișier **note** al cărui proprietar este. Din rațiuni lesne de înțeles, drepturile fișierului **note** sunt fixate la **rw-----**. Utilizatorul **profesor** dorește să permită utilizatorilor din grupul `studenti` să vadă unele informații din fișierul **note**.

Pentru aceasta, **profesor** creează un fișier executabil **examen** (proprietarul lui **examen** este **profesor**) care permite citirea (eventual selectivă) de informații din fișierul **note**. Proprietarul atribuie pentru **examen** drepturile **rw-x--x--x** și pune bitul `setuid` al lui **examen** pe 1. Această atribuire se face cu comanda **chmod +s examen**. Noile drepturi afișate ale lui **examen** sunt: **rws--x--x**

Utilizatorii **studenti**, în momentul lansării programului **examen**, primesc aceleași drepturi de acces la fișiere ca și **profesor**. În particular programul **examen** poate accesa fișierul **note** (vezi drepturile acestui fișier) chiar dacă el nu a fost lansat în execuție de către **profesor**. În absența lui `setuid` pentru **examen**, acesta poate fi, totuși lansat în execuție, însă nu poate să acceseze fișierul **note**.

După cum spuneam mai sus, nucleul SO Unix identifică utilizatorii prin numere naturale asociate unic, numite UID-uri (User IDentifications). De asemenea, identifică grupurile de utilizatori prin numere numite GID-uri (Group IDentifications).

Pe parcursul execuției programului **examen** acestuia i se mai asociază în plus identificatorul EUID (effective UID), care coincide cu UID-ul lui **profesor**, prin care asigură accesul la resurse.

Mecanismul `setuid` permite o foarte elastică manevrare a fișierelor. În schimb, dacă superuserul gestionează prost acest mecanism, atunci potențialii infractori au un câmp larg de acțiune. Să considerăm, din rațiuni evidente, doar un scenariu simplu: Presupunem ca `root` este proprietar al unui fișier executabil cu bitul `setuid` setat și cu drept de scriere pentru alții. În această situație, un răuvoitor poate să modifice acest fișier executabil așa încât să aibă o acțiune malefică ce presupune acces la resurse ale superuserului!. Acțiunea se va putea executa deoarece EUID-ul este UID-ul lui `root`!

Modificarea drepturilor de acces la fișiere se poate face numai de către proprietarul fișierului (sau de către superuser), folosind comanda `chmod`. Modificarea proprietarului sau a grupului se poate face, în aceleași condiții folosind comanda `chown`.

Un exemplu tipic în care se folosește `setuid` este comanda `/usr/bin/passwd`. Aceasta este lansată de către fiecare utilizator atunci când dorește să-și schimbe parola. Efectul ei se răsfrânge asupra fișierului `/etc/shadow`. În acest scop, se stabilesc drepturile:

```
-r-s--x--x  1 root  root  22312 Sep 25 18:52 /usr/bin/passwd
-r-----  1 root  root  10256 Mar  2 14:40 /etc/shadow
```

Deci programul `passwd` are `setuid`, ceea ce permite accesul la `/etc/shadow` numai prin programul `/usr/bin/passwd`.

4.1.5 Principalele directoare ale unui sistem de fişiere Unix

De-a lungul evoluţiei sistemelor din familia Unix, partea superioară a structurii sistemului de fişiere a avut mai mult sau mai puţin o formă standard. De fapt, fiecare versiune Unix şi-a fixat o structură specifică a părţii superioare din sistemul de fişiere. Diferenţele între aceste structuri nu sunt prea mari. Mai mult, din raţiuni de compatibilitate, versiunile mai noi definesc legături suplimentare hard (nu legături simbolice), pentru a asigura compatibilitatea cu sistemele de fişiere mai vechi. Din această cauză este cel mai nimerit să se studieze o reuniune a celor mai răspândite structuri. O astfel de structură este prezentată de noi în fig. 4.3.

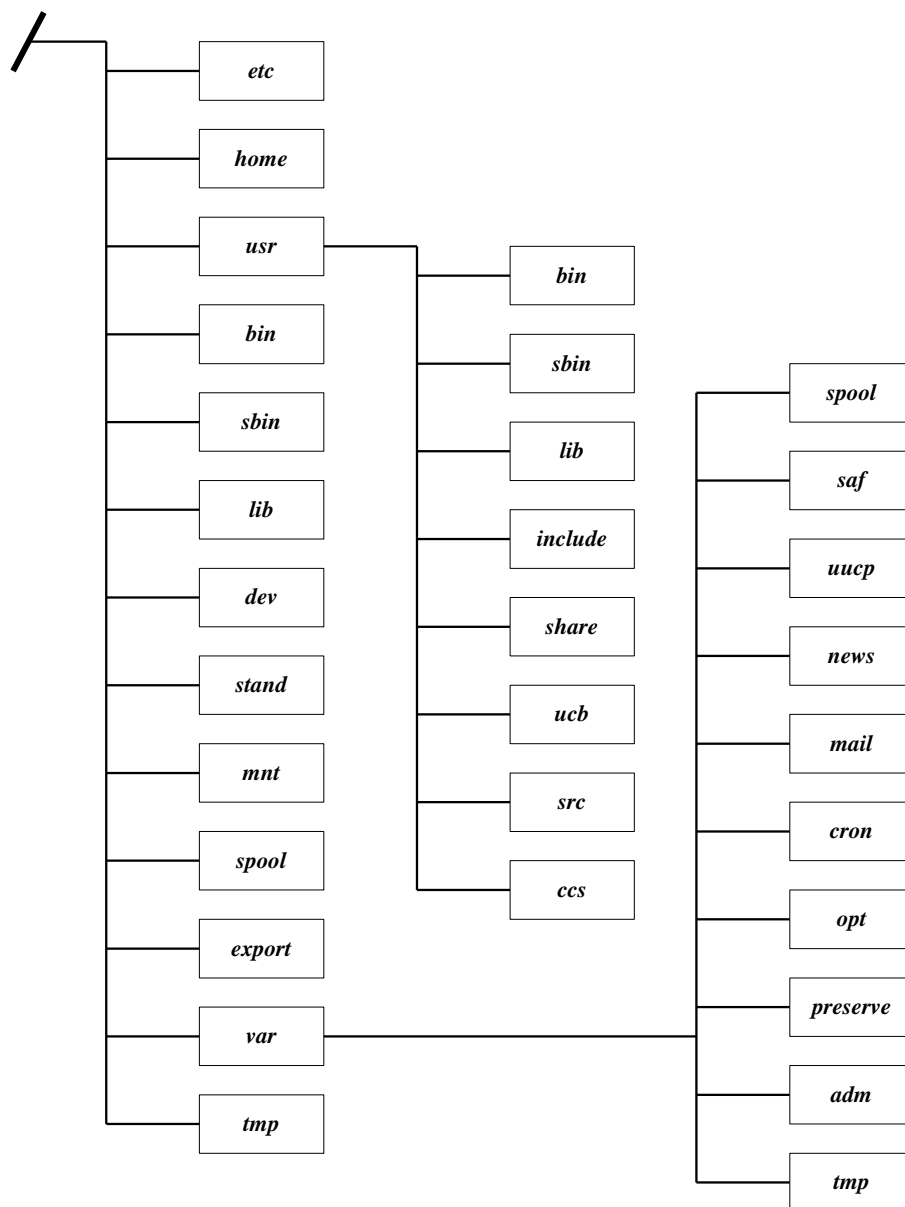


Figura 4.3 Structura superioară a unui sistem de fişiere Unix

Directorul `/etc` conţine informaţii specifice maşinii necesare întreţinerii sistemului. Acestea sunt de fapt datele restrictive şi periculoase din sistem. Un exemplu de fişier ce conţine informaţii specifice maşinii este, spre exemplu, `/etc/rc2.d` care este un director ce conţine programe shell executate de procesul *init* la schimbarea stării. Tot aici sunt plasate fişierele `/etc/passwd`, `/etc/group`, `/etc/shadow` care sunt folosite pentru administrarea utilizatorilor.

Directorul `/home` este folosit pentru directorii gazdă ai utilizatorilor. În momentul intrării în sistem, fiecare utilizator indică directorul lui, care este fixat ca director curent (*home directory*) în `/etc/passwd`. Deşi se poate trece uşor de la un director la altul, în mod normal fiecare utilizator rămâne în propriul lui director, unde îşi dezvoltă propria lui structură arborescentă de directori.

Directorul `/usr` este folosit în mod tradiţional pentru a stoca fişiere ce pot fi modificate. La primele versiuni de Unix conţinea şi fişierele utilizatorilor. În prezent este punctul de montare pentru partiţia ce conţine `usr`. El conţine programe executabile dependente şi independente de sistemul de fişiere, precum şi fişiere necesare acestora, dar care nu cresc dinamic. Asupra conţinutului de subdirectoare ale lui `/usr` vom reveni puţin mai târziu.

Directorul `/bin` conţine programele principalelor comenzi standard Unix: compilatoare, asamblatoare, editoare de texte, utilitare etc. Versiunile mai noi de Unix plasează acest director în `/usr/bin`.

Directorul `/sbin` (super-utilizator bin) conţine comenzi critice pentru procedura de încărcare a sistemului. Orice comenzi administrative care necesită lucrul mono-utilizator sunt în acest director. Copii ale acestor comenzi se află în directoarele `/usr/bin` şi `/usr/sbin`, astfel încât el (`/sbin`) poate fi refăcut dacă este necesar.

Directorul `/lib` conţine diverse biblioteci şi baze de date necesare apelurilor sistem. Doar versiunile mai vechi de Unix plasează acest director în `/`, cele actuale îl plasează în `/usr/lib`.

Directorul `/dev` este folosit pentru memorarea fişierelor speciale (fişierele devices). Practic, fiecare tip de terminal şi fiecare tip de unitate de disc trebuie să aibă asociat un astfel de fişier special. Incepând cu *SVR4* se permite ca în `dev` să existe şi subdirectoare care să grupeze astfel de device-uri.

Directorul `/stand` conţine informaţiile necesare încărcării sistemului.

Directorul `/mnt` este folosit pentru a monta un sistem de fişiere temporar. De exemplu, un sistem de fişiere de pe un disc flexibil poate fi montat în `/mnt` pentru a verifica fişierele de pe această dischetă.

Directorul `/spool` este plasat în `/` doar în versiunile mai vechi de Unix. În el sunt memorate fişierele tampon temporare destinate prelucrărilor asincrone: listări asincrone de fişiere (`/spool/lpd`) şi execuţia la termen a unor comenzi (`/spool/at`).

Directorul `/export` este folosit ca punct implicit de montare pentru un arbore de sistem de fişiere exportat, pentru fişierele în reţea gestionate prin pachetul NFS (Network File System).

Directorul `/var` este folosit pentru memorarea fișierelor care cresc dinamic. În particular, multe dintre versiunile de Unix țin în acest director fișierele `INBOX` cu căsuțele poștale ale utilizatorilor. Structura de subdirectoare a lui `/var` o vom descrie ceva mai încolo.

Directorul `/tmp` este folosit pentru a memora fișiere temporare pentru aplicații. În mod normal, aceste fișiere nu sunt salvate și sunt șterse după o perioadă de timp.

În continuare vom descrie pe scurt conținutul directorului `/usr`. După cum se poate vedea, unele dintre subdirectoare sunt plasate (și au fost descrise) la nivel de rădăcină `/`. Versiunile mai noi de Unix, începând cu *SVR4*, le-au coborât din rădăcină ca subdirectoare ale lui `/usr`. Este cazul directoarelor `bin`, `sbin`, `lib`.

Directorul `/usr/include` conține fișierele header (`*.h`) standard ale limbajului C de sub Unix.

Directorul `/usr/share` conține o serie de directoare partajabile în rețea. În multe dintre sistemele noi, în el se află directoarele: `man` cu manualele Unix, `src` cu sursele C ale nucleului Unix și `lib` mai sus prezentat.

Directorul `/usr/ucb` conține programele executabile compatibile Unix BSD.

Directorul `/usr/src` conține textele sursă C ale nucleului Unix de pe mașina respectivă.

Directorul `/usr/ccs` conține instrumentele de dezvoltare a programelor C oferite de Unix: `cc`, `gcc`, `dbx`, `cb`, `indent`, etc.

În continuare vom descrie conținutul directorului `/var`. Ca și mai sus, unele subdirectoare de la nivelele superioare au fost mutate aici de către versiunile mai noi de Unix. Este vorba de directoarele `spool` și `tmp`.

Directorul `/var/saf` conține fișiere jurnal și de contabilizare a serviciilor oferite.

Directorul `/var/uucp` conține programele necesare efectuării de copii de fișiere între sisteme Unix (Unix to Unix CoPy). Acest gen de servicii este primul pachet de comunicații instalat pe sisteme Unix, este operațional încă din 1978 și este utilizat și astăzi atunci când nu există alt mijloc mai modern de comunicații. Un astfel de sistem permite, de exemplu, apelul telefonic între două sisteme Unix, iar după luarea contactului cele două sisteme își schimbă între ele o serie de fișiere, ca de exemplu mesajele de poștă electronică ce le sunt destinate.

Directorul `/var/news` conține fișierele necesare serviciului de (știri) noutăți (`news`) care poate fi instalat pe mașinile Unix.

Directorul `/var/mail` conține căsuțele poștale implicite ale utilizatorilor (`INBOX`). Pe unele sisteme, ca de exemplu pe Linux, acestea se află în `/var/spool/mail`.

Directorul `/var/cron` conține fișierele jurnal necesare serviciilor executate la termen.

Directorul `/var/opt` constituie un punct de montare pentru diferite pachete de aplicații.

Directorul `/var/preserve` conţine, la unele implementări Unix (SVR4) fişiere jurnal destinate refacerii stării editoarelor de texte “picate” ca urmare a unor incidente.

Directorul `/var/adm` conţine fişiere jurnal (log-uri) de contabilizare şi administrare a sistemului. La versiunile mai noi acestea sunt în `/var/log`.

După cum se poate vedea uşor, structura de directori Unix începând de la rădăcină este relativ dependentă de tipul şi versiunea de Unix. De fapt, este vorba de “Unix vechi” şi “Unix noi”. De asemenea, multe dintre directoare au fost înlocuite sau li s-a schimbat poziţia în structura de directori. Tabelul de mai jos prezintă câteva corespondenţe între vechile şi noile plasări de fişiere.

Nume vechi	Nume nou
<code>/bin</code>	<code>/usr/bin</code>
<code>/lib</code>	<code>/usr/lib</code>
<code>/usr/adm</code>	<code>/var/adm</code>
<code>/usr/spool</code>	<code>/usr/spool</code>
<code>/usr/tmp</code>	<code>/var/tmp</code>
<code>/etc/termcap</code>	<code>/usr/share/lib/termcap</code>
<code>/usr/lib/terminfo</code>	<code>/usr/share/lib/terminfo</code>
<code>/usr/lib/cron</code>	<code>/etc/cron.d</code>
<code>/usr/man</code>	<code>/usr/share/man</code>
<code>/etc/<programe></code>	<code>/usr/bin/<programe></code>
<code>/etc/<programe></code>	<code>/sbin/<programe></code>

4.2 Structura internă a discului Unix

4.2.1 Partiţii şi blocuri

Un sistem de fişiere Unix este găzduit fie pe un periferic oarecare (hard-disc, CD, dischetă etc.), fie pe o *partiţie* a unui hard-disc. Partiţionarea unui hard-disc este o operaţie (relativ) independentă de sistemul de operare ce va fi găzduit în partiţia respectivă. De aceea, atât partiţiilor, cât şi suporturilor fizice reale le vom spune generic, *discuri Unix*.

Blocul 0	- bloc de boot
Blocul 1	- Superbloc
Blocul 2	- inod
- - - - -	- - - - -
Blocul n	- inod
Blocul n+1	zona fişier
- - - - -	- - - - -
Blocul n+m	zona fişier

Figura 4.4 Structura unui disc Unix

Un fişier Unix este o succesiune de octeţi, fiecare octet putând fi adresat în mod individual. Este permis atât accesul secvenţial, cât şi cel direct.

Unitatea de schimb dintre disc și memorie este *blocul*. La sistemele mai vechi acesta are 512 octeți, iar la cele mai noi poate avea și 1Ko sau 4Ko, pentru o mai eficientă gestiune a spațiului.

Un sistem de fişiere Unix este o structură de date rezidentă pe disc. Așa după cum se vede din fig. 4.4, un disc este compus din patru categorii de blocuri.

Blocul 0 conține programul de încărcare al **SO**. Acest program este dependent de mașina sub care se lucrează. Acțiunea lui se declanșează la pornirea sistemului - apăsarea butonului <**Reset**> sau <**Power**>. La acest moment, intră în lucru un mic program din memoria ROM, așa-numitul *cod startup din BIOS*. Acesta știe să citească primii 512 octeți de pe disc, să îi depună undeva în memoria RAM și să lanseze în execuție secvența de octeți citită. Evident, în primii 512 octeți de pe disc va fi un program, *bootprogramul* sau *programul de pornire a încărcării sistemului de operare*. Principala sarcină a *bootprogramului* este aceea de a încărca în RAM partea din *kernel*-ul Unix (sau a altui sistem de operare) special destinată încărcării comple a sistemului de operare.

Nu este obligatoriu ca întregul program de boot să se găsească în blocul 0, ci doar partea care odată executată să permită sistemului lucrul cu alte tipuri de memorie. Odată acest punct atins, execuția programului poate continua cu părți care se găsesc pe alte medii de stocare nevolatile, cum ar fi un harddisk, o dischetă, un CR-ROM sau chiar pe un server de boot în cazul stațiilor fără harddisk.

În majoritatea cazurilor în care nu este vorba de o primă instalare a sistemului, restul programului de boot se află stocat pe harddisk într-o zonă specială a acestuia numită *partiție* sau *segment de boot*. Această partiție are o structură de date extrem de simplă (mult mai simplă decât a sistemului de fişiere) și poate fi accesată foarte ușor.

Blocul 1 este numit și *superbloc*. În el sunt trecute o serie de informații prin care se definește sistemul de fişiere de pe disc. Printre aceste informații amintim:

- numărul n de *inoduri* (detaliem imediat);
- numărul de zone definite pe disc;
- pointeri spre harta de biți a alocării inodurilor;
- pointeri spre harta de biți a spațiului liber disc;
- dimensiunile zonelor disc, etc.

Blocurile 2 la n , unde n este o constantă a formatării discului, se constituie în zona de *inoduri*. Un *inod* (sau *i-nod*) este numele, în terminologia Unix, a *descriptorului* unui fișier. Inodurile sunt memorate pe disc sub forma unei liste (numită *i-listă*). Numărul de ordine al unui inod în cadrul *i-listei* se reprezintă pe doi octeți și se numește *i-număr*. Acest *i-număr* constituie legătura dintre fișier și programele utilizator.

Blocurile $n+1$ la $n+m$ reprezintă *zona fișierelor*. Este partea cea mai mare din cadrul discului. Alocarea spațiului pentru fișiere se face printr-o variantă elegantă de indexare. Informațiile de plecare pentru alocare sunt fixate în inoduri. La discurile Unix actuale există, de regulă, mai multe zone de inoduri intercalate cu mai multe zone de fișiere.

4.2.2 Directori și inoduri

Structura unei intrări într-un fișier director este ilustrată în fig. 4.5.

Numele fişierului (practic oricât de lung)	inumăr
--	--------

Figura 4.5 Structura unei intrări în director

Deci, în director se află numele fişierului şi referinţa spre inodul descriptor al fişierului.

Un *inod* are, de regulă, între 64 şi 128 de octeţi şi el conţine informaţiile din tabelul următor:

mode	Drepturile de acces şi tipul fişierului.
link count	Numărul de directoare care conţin referiri la acest inumăr, adică numărul de legături spre acest fişier.
user ID	Numărul (UID) de identificare a proprietarului.
group ID	Numărul (GID) de identificare a grupului.
Size	Numărul de octeţi (lungimea) fişierului.
access time	Momentul ultimului acces la fişier.
mod time	Momentul ultimei modificări a fişierului.
inode time	Momentul ultimei modificări a structurii inodului.
block list	Lista adreselor disc pentru primele blocuri care aparţin fişierului.
indirect list	Referinţe către celelalte blocuri care aparţin fişierului.

4.2.3 Schema de alocare a blocurilor disc pentru un fişier

Fiecare sistem de fişiere Unix are câteva constante proprii, printre care amintim:

- lungimea unui inod,
- lungimea unui bloc,
- lungimea unei adrese disc (implicit câte adrese disc încap într-un bloc),
- câte adrese de prime blocuri se înregistrează direct în inod,
- câte referinţe se trec în lista de referinţe indirecte.

Indiferent de valorile acestor constante, principiile de înregistrare / regăsire sunt aceleaşi şi le vom prezenta în cele ce urmează. Pentru fixarea ideilor, vom alege aceste constante cu valorile întâlnite mai frecvent la sistemele de fişiere deja consacrate. Cu aceste constante, în fig. 4.6 este prezentată structura pointerilor spre blocurile ataşate unui fişier Unix. Aceste constante sunt:

- un inod se reprezintă pe 64 octeţi,
- un bloc are lungimea de 512 octeţi,
- adresă disc se reprezintă pe 4 octeţi, deci încap 128 adrese disc într-un bloc,
- în inod trec direct primele 10 adrese de blocuri,
- lista de adrese indirecte are 3 elemente.

În inodul fişierului se află o listă cu 13 intrări, care desemnează blocurile fizice aparţinând fişierului.

- Primele 10 intrări conţin *adresele primelor* 10 blocuri de câte 512 octeţi care aparţin fişierului.
- Intrarea nr. 11 conţine adresa unui bloc, numit *bloc de indirectare simplă*. El conţine adresele următoarelor 128 blocuri de câte 512 octeţi, care aparţin fişierului.
- Intrarea nr. 12 conţine adresa unui bloc, numit *bloc de indirectare dublă*. El conţine adresele a 128 blocuri de indirectare simplă, care la rândul lor conţin, fiecare, adresele a câte 128 blocuri, de 512 octeţi fiecare, cu informaţii aparţinând fişierului.

- Intrarea nr. 13 conține adresa unui bloc, numit *bloc de indirectare triplă*. În acest bloc sunt conținute adresele a 128 blocuri de indirectare dublă, fiecare dintre acestea conținând adresele a câte 128 blocuri de indirectare simplă, iar fiecare dintre acestea conține adresele a câte 128 blocuri, de câte 512 octeți, cu informații ale fișierului.

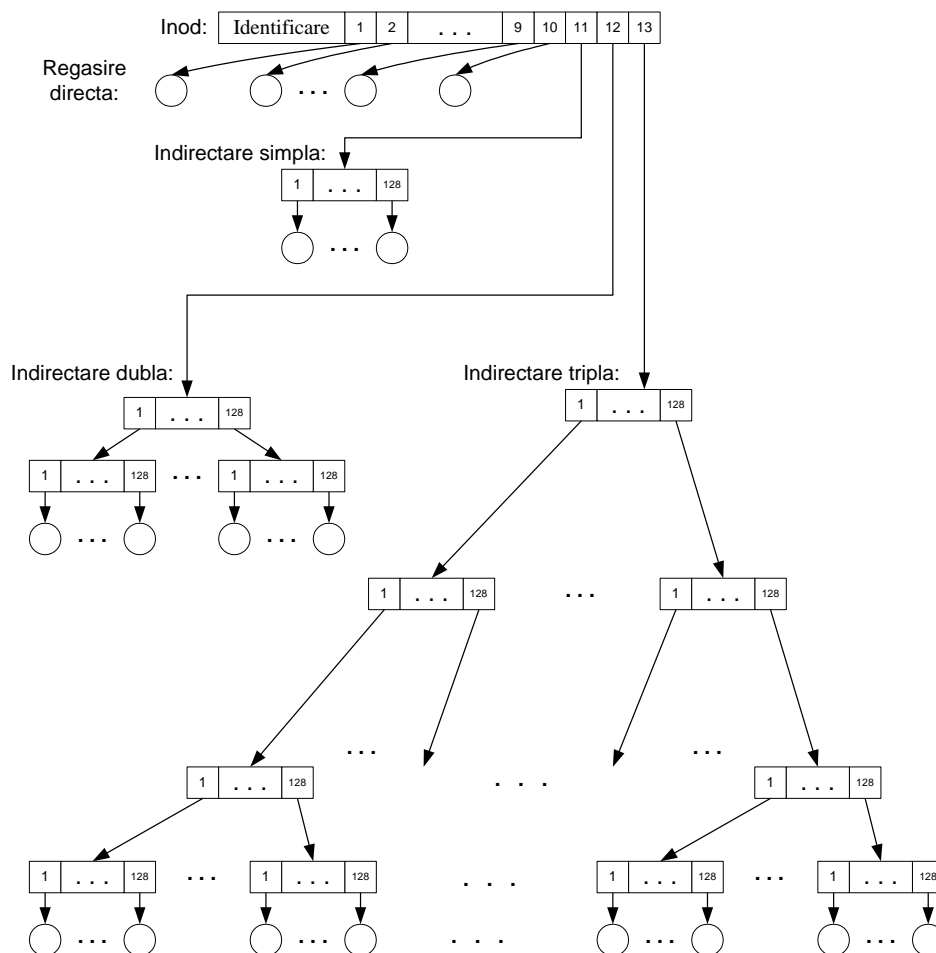


Figura 4.6 Structura unui inod și accesul la blocurile unui fișier

În fig. 4.6 am ilustrat prin cercuri blocurile de informație care aparțin fișierului, iar prin dreptunghiuri blocurile de referințe, în interiorul acestora marcând referințele.

Numărul de accese necesare pentru a obține direct un octet oarecare este cel mult 4. Pentru fișiere mici acest număr este și mai mic. Atât timp cât fișierul este deschis, inodul lui este prezent în memoria internă. Tabelul următor dă numărul maxim de accese la disc pentru a obține, în acces direct orice octet dintr-un fișier, în funcție de lungimea fișierului.

Lungime maximă (blocuri)	Lungime maximă (octeți)	Accese indirecte	Accese la informație	Total accese
10	5120	-	1	1
$10+128 = 138$	70656	1	1	2
$10+128+128^2 = 16522$	8459264	2	1	3
$10+128+128^2+128^3=2113674$	1082201088	3	1	4

La sistemele Unix actuale lungimea unui bloc este de 4096 octeți care poate înregistra 1024 adrese, iar în inod se înregistrează direct adresele primelor 12 blocuri. În aceste condiții, tabelul de mai sus se transformă în:

Lungime maximă (blocuri)	Lungime maximă (octeți)	Accese indirecte	Accese la informație	Total accese
12	49152	-	1	1
$12+1024 = 1036$	4243456	1	1	2
$12++1024+1024^2 = 1049612$	4299210752	2	1	3
$12+1024+1024^2+1024^3 = 1073741824$	4398046511104 (peste 5000Go)	3	1	4

Practic, orice fișier actual, indiferent de mărimea lui, poate fi reprezentat printr-o astfel de schemă.

4.2.4 Accesul proceselor la fișiere

Unix privește conceptul de fișier într-un sens ceva mai larg decât o fac alte sisteme de operare. Așa cum am mai arătat mai sus, există opt tipuri de fișiere:

- normale,
- directori,
- legături hard,
- legături simbolice,
- FIFO, (pipe cu nume),
- socketuri,
- periferice caracter,
- periferice bloc.

Pe lângă acestea, nucleul mai gestionează, într-o sintaxă similară fișierelor, următoarele patru tipuri de comunicații între procese:

- pipe anonime (a se deosebi de FIFO - pipe cu nume);
- segmente de memorie partajată;
- cozi de mesaje;
- semafoare.

Suporturile fizice pentru aceste 12 tipuri de fișiere sunt, în ultimă instanță:

- *Zona fișierelor pe disc*, pentru fișierele normale, directori, legături hard și simbolice, FIFO și socket din familia Unix.
- *Perifericul respectiv* pentru perifericele caracter și bloc.
- *Zone rezervate de nucleu în memoria internă*, pentru pipe, memorie partajată, cozi de mesaje și semafoare.
- *Interfața de comunicație prin rețea*, pentru socket din familia Internet.

Pentru a putea asigura o tratare uniformă, traseul accesului unui proces la un fișier trece prin mai multe nivele: proces, sistem, inod, fișier, așa cum se vede în fig. 4.7. În fig. 4.7 prezentăm un exemplu în care există trei procese: **A**, **B**, **C** și patru fișiere **F1**, **F2**, **F3**, **F4**. Intrările

pentru legături la diversele nivele le-am notat cu litere mici **a – w**. În cele ce urmează descriem cele patru nivele de legătură.

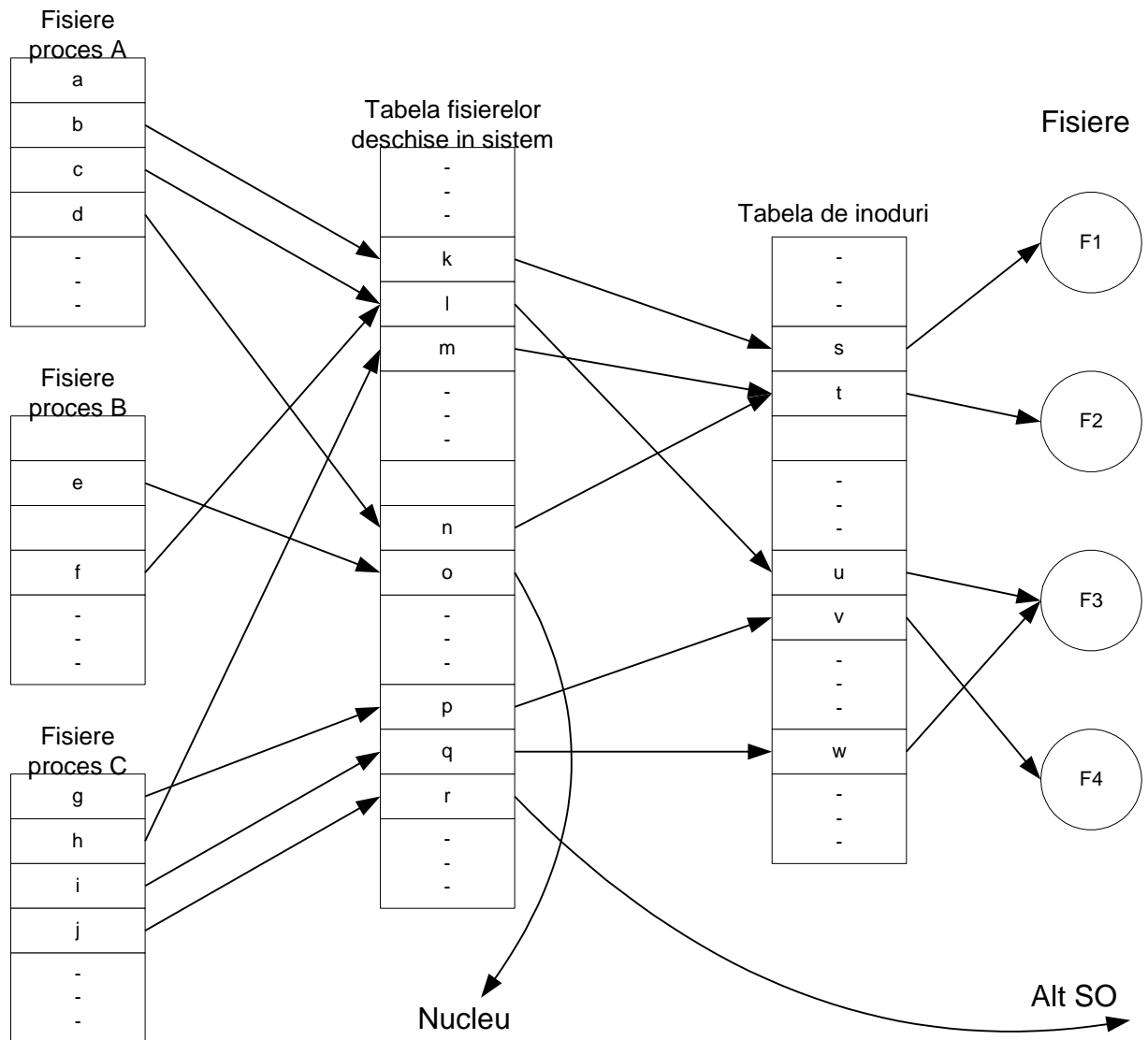


Figura 4.7 Corespondența Unix între procese și fişiere

Nivelul proces. Fiecare proces își întreține o tabelă proprie în care înregistrează toate fişierele lui deschise. În fig. 4.7 am notat cu **a–j** câteva intrări de pe acest nivel.

Nivelul sistem întreține o tabelă unică cu toate fişierele deschise de către toate procesele din sistem. În fig. 4.7 am notat prin **k–r** câteva intrări din această tabelă.

Nivelul inod este de fapt zona (zonele) de inoduri de pe disc. Pentru fişierele deschise, se păstrează în memoria internă copii ale inodurilor corespunzătoare. În fig. 4.7 am notat prin **s–w** câteva astfel de intrări.

Nivelul fişier este reprezentat de blocurile disc ce aparțin fişierului. În fig. 4.7 am notat **F1–F4** astfel de fişiere.

Tabela de fișiere la nivel proces are intrările numerotate începând de la 0. Primele trei intrări sunt rezervate astfel:

- intrarea 0 este rezervată intrării standard a procesului (vezi în fig. 4.7 intrările **a** din procesul **A** și **g** din procesul **C**);
- intrarea 1 este rezervată ieșirii standard (vezi intrările **b**, **e**, **h** din fig. 4.7);
- intrarea 2 este rezervată fișierului standard de erori (unde sistemul afișează mesajele de eroare pentru proces - vezi intrările **c**, **i** din fig. 4.7).

După cum se va vedea în secțiunile următoare, toate apelurile sistem de lucru cu fișiere folosesc pentru identificarea fișierului un număr întreg numit *handle* sau *descriptor de fișier*. Acest întreg este chiar indexul intrării fișierului în tabela procesului respectiv.

În gestiunea fișierelor deschise pe aceste patru nivele, marea majoritate a fișierelor au exact câte o singură intrare pe fiecare nivel, corespunzătoare fișierului respectiv. De exemplu, procesul **A** din fig. 4.7 vede fișierul **F1** prin intermediul intrărilor **b**, **k**, **s** din cele trei tabele. De asemenea, procesul **C** vede fișierul **F4** prin intermediul intrărilor **g**, **p**, **v**.

În Unix este posibil ca mai multe procese să deschidă, în același timp, un același fișier - *multiacces la fișiere*. Acest lucru este posibil prin faptul că două intrări de fișiere din două procese pot puncta spre aceeași intrare din tabela sistem. În fig. 4.7 intrarea **c** de la procesul **A** și intrarea **f** de la procesul **B** folosesc în comun același fișier, cel localizat de intrarea **l** din tabela sistem.

De regulă, fiecare intrare din tabela sistem punctează spre un inod, iar intrări diferite punctează spre inoduri diferite. Așa sunt, de exemplu, legăturile **k-s**, **l-u**, **p-v**, **q-w**.

Sunt interesante abaterile de la regula de mai sus. Astfel, spre exemplu avem legăturile **m-t** și **n-t**. Această corespondență este posibilă în cazul în care cel puțin una dintre intrările **m** sau **n** se referă la o *legătură simbolică*.

Legăturile din **o** și din **r** nu punctează spre nici un inod. Legătura **o** este un canal de tip pipe, memorie partajată, coadă de mesaje sau semafor, acestea fiind găzduite în nucleu. Legătura **r** este un socket, prin care se realizează legătura prin rețea cu un alt sistem de operare.

În fine, în marea majoritate a cazurilor există o corespondență biunivocă între inoduri și fișierele corespunzătoare. În fig. 4.7 avem corespondențele **s-F1**, **t-F2**, **v-F4**.

Abaterea de la această regulă este făcută doar de *legăturile hard*. Fiecare astfel de legătură creează un nou inod pentru același fișier. În fig. 4.7 avem corespondențele **u-F3** și **w-F3**, cel puțin una dintre **u** și **w** fiind o legătură hard.

4.3 Apeluri sistem pentru lucrul cu fișiere

4.3.1 Operații I/O

Există două posibilități de efectuare a operațiilor I/O din programe C:

- Prin funcţiile standard C (`fopen`, `fclose`, `fgets`, `fprintf`, `fread`, `fwrite`, `fseek`, etc.) existente în bibliotecile standard C; prototipurile acestora se află în fişierul header `<stdio.h>` (*nivelul superior de prelucrare al fişierelor*).
- Prin funcţii standardizate POSIX (`open`, `close`, `read`, `write`, `lseek`, `dup`, `dup2`, `fcntl`, etc.) care reprezintă puncte de intrare în nucleul Unix şi ale căror prototipuri se află de regulă în fişierul header `<unistd.h>`, dar uneori se pot afla şi în `<sys/types.h>`, `<sys/stat.h>` sau `<fcntl.h>` (*nivelul inferior de prelucrare al fişierelor*).

Prima categorie de funcţii o presupunem cunoscută deoarece face parte din standardul C (ANSI). Funcţiile din această categorie reperează orice fişier printr-o structură `FILE *`, pe care o vom numi *descriptor de fişier*.

Funcţiile din a doua categorie constituie apeluri sistem Unix pentru lucrul cu fişiere şi fac obiectul secţiunii care urmează. Ele (antetul lor) sunt cuprinse în standardul POSIX. Funcţiile din această categorie reperează orice fişier printr-un întreg nenegativ, numit *handle*, dar atunci când confuzia nu este posibilă îl vom numi tot *descriptor de fişier*. Fiecare astfel de descriptor este index în tabela de fişiere deschise a procesului. De exemplu, să considerăm procesul **C** din fig. 4.7. Aici, aceste handle sau descriptori sunt indecşi cu valorile **0**, **1**, **2**, **3** ... care punctează la intrările **g**, **h**, **i**, **j**, ... din tabela de fişiere a procesului **C**.

4.3.2 Apelul sistem open

Prototipul funcţiei sistem este:

```
int open (char *nume, int flag [, unsigned int drepturi ]);
```

Funcţia `open` întoarce un întreg - *handle* sau *descriptor de fişier*, folosit ca prim argument de către celelalte funcţii POSIX de acces la fişier. În caz de eşec `open` întoarce valoarea -1 şi poziţionează corespunzător variabila `errno`. În cele ce urmează vom numi `descr` acest număr.

`nume` - specifică printr-un string C, calea şi numele fişierului în conformitate cu standardul Unix.

Modul de deschidere este precizat de parametrul de deschidere `flag`. Principalele lui valori posibile sunt:

- `O_RDONLY` deschide fişierul numai pentru citire
- `O_WRONLY` deschide fişierul numai pentru scriere
- `O_RDWR` deschide fişierul atât pentru citire şi pentru scriere
- `O_APPEND` deschide pentru adaugarea - scrierea la sfârşitul fişierului
- `O_CREAT` creează un fişier nou dacă acesta nu există, sau nu are efect dacă fişierul deja există; următoarele două constante completează crearea unui fişier
- `O_TRUNC` asociat cu `O_CREAT` (şi exclus `O_EXCL` vezi mai jos) indică crearea necondiţionată, indiferent dacă fişierul există sau nu
- `O_EXCL` asociat cu `O_CREAT` (şi exclus `O_TRUNC`), în cazul în care fişierul există deja, `open` eşuează şi semnalează eroare

- `O_NDELAY` este valabil doar pentru fişiere de tip pipe sau FIFO şi vom reveni asupra lui când vom vorbi despre pipe şi FIFO.

În cazul în care se folosesc mai multe constante, acestea se leagă prin operatorul `|`, ca de exemplu: `O_CREAT | O_TRUNC | O_WRONLY`.

Parametrul `drepturi` este necesar doar la crearea fişierului şi indică drepturile de acces la fişier (prin cei 9 biţi de protecţie) şi acţionează în concordanţă cu specificarea `umask`.

4.3.3 *Apelul sistem close*

Inchiderea unui fişier se face prin apelul sistem:

```
int close (int descr);
```

Parametrul `descr` este cel întors de apelul `open` cu care s-a deschis fişierul. Funcţia întoarce 0 la succes sau -1 în caz de eşec.

4.3.4 *Apelurile sistem read şi write*

Acestea sunt cele mai importante funcţii sistem de acces la conţinutul fişierului. Prototipurile lor sunt:

```
int read (int descr, void *mem, unsigned int noct);  
int write (int descr, const void *mem, unsigned int noct);
```

Efectul lor este acela de a citi (scrie) din (în) fişierul indicat de `descr` un număr de `noct` octeţi, depunând (luând) informaţiile în (din) zona de memorie aflată la adresa `mem`. În majoritatea cazurilor de utilizare, `mem` referă un şir de caractere (`char* mem`).

Cele două funcţii întorc numărul de octeţi efectiv transferaţi între memorie şi suportul fizic al fişierului. De regulă, acest număr coincide cu `noct`, dar sunt situaţii în care se transferă mai puţin de `noct` octeţi.

Aceste două operaţii sunt atomice - indivizibile şi neîntreruptibile de către alte procese. Deci, dacă un proces a început un transfer între memorie şi suportul fişierului, atunci nici un alt proces nu mai are acces la fişier până la terminarea operaţiei `read` sau `write` curente.

Operaţia se consideră terminată dacă s-a transferat cel puţin un octet, dar nu mai mult de maximum dintre `noct` şi ceea ce permite suportul fişierului. Astfel, dacă în fişier există doar `n` octeţi rămaşi necitiţi atunci se vor transfera în memorie doar `n` octeţi şi `read` va întoarce valoarea `n`. Dacă în suportul fişierului există cel mult `n` octeţi disponibili, atunci se depun din memorie în fişier doar `n` octeţi şi `write` va întoarce valoarea `n`. În ambele situaţii, pointerul curent al fişierului avansează cu numărul de octeţi transferaţi. Dacă la lansarea unui `read` nu mai există nici un octet necitit din fişier - s-a întâlnit marcajul de sfârşit de fişier, atunci funcţia întoarce valoarea 0.

Dacă operația de citire sau de scriere nu se poate termina - a apărut o eroare în timpul transferului - funcția întoarce valoarea -1 și se poziționează corespunzător variabila `errno`.

Ca exemplu de folosire a apelurilor sistem `open`, `close`, `read` și `write` vom scrie un program care face același lucru ca și comanda shell `cp`, adică copiază conținutul unui fișier într-altul. Cele două fișiere se vor da ca parametri în linia de comandă. Dacă al doilea fișier (fișierul destinație) nu există, el se va crea, iar dacă există deja, el se va suprascrive. Sursa programului este prezentată în fig. 4.8.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

main(int argc, char* argv[]) {
    int fd_sursa, fd_dest, n, i;
    char buf[100], *p;
    if (argc!=3) {
        fprintf(stderr, "Eroare: trebuie dati 2 parametrii.\n");
        exit(1);
    }
    //deschidem primul fișier în modul read-only
    fd_sursa = open(argv[1], O_RDONLY);
    if (fd_sursa<0) {
        fprintf(stderr, "Eroare: %s nu poate fi deschis.\n", argv[1]);
        exit(1);
    }
    /* deschidem al doilea fișier în modul write-only. Dacă el nu
     * există, se va crea, sau dacă există va fi trunchiat. 0755
     * specifică drepturile de acces ale fișierului nou creat
     * (citire+scriere+executie pentru proprietar, citire+executie
     * pentru grup și alții).
     */
    fd_dest = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0755);
    if (fd_dest<0) {
        fprintf(stderr, "Eroare: fisierul %s nu poate fi deschis.\n",
                argv[2]);
        exit(1);
    }
    //citim din fișierul fd_sursa bucăți de maxim 100 octeți și le
    //scriem în fișierul fd_dest, până când nu mai avem ce citi.
    for ( ; ; ) {
        n=read(fd_sursa, buf, sizeof(buf));
        if ((n == 0) break; // S-a terminat de citit fișierul
        p = buf;
        for( ; n > 0 ; ) { // Poate nu se pot scrie odata n octeti
            i = write(fd_dest, p, n);
            if (i == n) break;
            p += i;
            n -= i;
        }
    }
    //închidem cele două fișiere
    close(fd_sursa);
    close(fd_dest);
}
```

Figura 4.8 Copierea unui fișier cu apelurile sistem `read` și `write`

4.3.5 Apelul sistem *lseek*

Funcția sistem `lseek` facilitează accesul direct la orice octet din fișier. Evident, pentru aceasta trebuie ca suportul fișierului să fie unul adresabil. Prototipul acestei funcții sistem este:

```
long lseek (int descr, long noct, int deUnde);
```

Se modifică pointerul curent în fișierul indicat de `descr` cu un număr de `noct` octeți. Punctul de unde începe numărarea celor `noct` octeți este indicat de către valoarea parametrului `deUnde`, astfel:

- de la începutul fișierului, dacă are valoarea `SEEK_SET` (valoarea 0)
- de la poziția curentă dacă are valoarea `SEEK_CUR` (valoarea 1)
- de la sfârșitul fișierului dacă are valoarea `SEEK_END` (valoarea 2)

4.3.6 Apelurile sistem *dup* și *dup2*

Aceste două funcții sistem permit ca un același fișier să fie accesibil prin doi descriptori diferiți. Presupunem că `descrvechi` este o intrare în tabela de fișiere deschise a unui proces, care punctează, așa cum am văzut în fig. 4.7 din 4.2.4, spre o intrare în tabela de fișiere deschise a sistemului. În urma apelului sistem, prin `dup` sau `dup2` se ocupă o nouă intrare `descrnou` din tabela de fișiere a procesului, care va puncta spre același fișier în tabela de fișiere deschise a sistemului. Această duplicare are loc în următoarele condiții:

- `descrvechi` și `descrnou` referă același fișier fizic
- ambele păstrează modul de acces la fișier stabilit la deschidere
- ambii descriptori partajează același pointer curent în fișier

Prototipurile celor două apeluri sistem sunt:

```
int dup (int descrvechi);
int dup2 (int descrvechi, int descrnou);
```

Apelul sistem `dup` face o copie a `descrvechi` în primul (cel mai mic număr) descriptor liber din tabela de fișiere a procesului.

Apelul sistem `dup2` face o copie a `descrvechi` în `descrnou`, închizând, dacă este cazul, fișierul către care puncta înainte `descrnou`.

Ambele apeluri întorc noul descriptor (`descrnou`) care duplică accesul la fișierul reperat prin `descrvechi`. În caz de eșec, ambele întorc -1 și poziționează corespunzător variabila `errno`.

De exemplu, dacă se dorește ca dintr-un program C mesajele de eroare să fie trecute într-un fișier de pe disc numit "ERORI", se poate proceda ca în programul din fig. 4.9.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

main () {
    int descrvechi, descrnou;
```



```

descrvechi = open("ERORI", O_CREAT|O_WRONLY, 0755);
if (descrvechi < 0) {
    fprintf(stderr, "Pe terminal prin stderr: open ERORI imposibil\n");
    fprintf(stdout, "Pe terminal prin stdout: open ERORI imposibil\n");
    exit(1);
}
descrnou = dup2(descrvechi, 2);
if (descrnou != 2) {
    fprintf(stderr, "Pe terminal prin stderr: dup2 imposibil\n");
    fprintf(stdout, "Pe terminal prin stdout: dup2 imposibil\n");
    exit(1);
}
fprintf(stderr, "Mesaj in ERORI prin stderr: dup2 REUSIT\n");
fprintf(stdout, "Mesaj pe terminal prin stdout: dup2 REUSIT\n");
} //main

```

Figura 4.9 Sursa testdup2.c

Presupunem că programul se lansează fără nici o redirectare pentru I/O standard. Partea esențială a programului este `descrnou = dup2(descrvechi, 2);` Prin aceasta, se închide automat fișierul cu descriptorul 2 care referă fișierul standard `stderr` și noul fișier standard de erori va fi fișierul `ERORI`. Dacă deschiderea sau `dup2` nu pot fi executate, atunci pe terminal va apare un mesaj în dublu exemplar: trimis prin `stderr` și prin `stdout`. În caz de succes, pe terminal va apare o linie trimisă prin `stdout` iar în fișierul `ERORI` linia trimisă prin `stderr`.

4.3.7 Apelul sistem *fcntl*

Această funcție sistem furnizează sau schimbă proprietăți ale unui fișier deschis indicat printr-un descriptor. Această funcție poate fi apelată prin unul dintre următoarele trei prototipuri:

```

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);

```

Primul parametru, `fd`, referă fișierul deschis asupra căruia se dorește aplicarea controlului.

Parametrul `cmd` este o constantă care specifică operația dorită și în funcție de ea mai urmează sau nu încă un argument. Vom prezenta doar o parte dintre posibilitățile oferite de `fcntl`, urmând ca în secțiunile următoare să revenim cu prezentarea de noi facilități:

- `fcntl(fd, F_DUPFD, arg)` are aproximativ același efect ca și `dup(fd)`, doar că `fcntl` copiază descriptorul `fd` în cel mai mic descriptor, mai mare sau egal cu `arg`.
- `fcntl(fd, F_SETFL, arg)` modifică flagurile de tratare a fișierului față de cum au fost fixate prin `open` sau printr-un `fcntl` precedent. Parametrul `arg` are aceleași valori ca și parametrul `flag` de la `open`: una sau mai multe constante folosite pentru a specifica modul de deschidere a fișierului.
- `fcntl(fd, F_GETFL)` întoarce o configurație de biți reprezentând reuniunea valorilor curente ale flagurilor de tratare a fișierului, așa cum au fost ele fixate prin `open` sau printr-un `fcntl(... F_SETFL ...)`.

Ce-a de a treia formă va fi tratată în secțiunea 4.6.3.

4.4 Gestiunea fişierelor

Unix permite efectuarea din C a principalelor operaţii asupra sistemului de fişiere, oferind în acest sens câteva apeluri sistem. Ele sunt echivalente cu comenzile Shell care efectuează aceleaşi operaţii. Aceste operaţii sunt definite prin specificaţii POSIX corespunzătoare.

De regulă, aceste funcţii au prototipurile într-unul dintre fişierele header:

```
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
```

4.4.1 Manevrarea fişierelor în sistemul de fişiere

Iată prototipurile celor mai importante dintre aceste apeluri sistem:

```
int chdir (const char *nume);
char *getcwd(char *mem, int dimensiune);
int mkdir (const char *nume, unsigned int drepturi);
int rmdir (const char *nume);
int unlink(const char *nume);
int link(const char *numevechi, const char *numenou);
int symlink(const char *numevechi, const char *numenou);
int chmod (const char *nume, unsigned int drepturi);
int stat (const char *nume, struct stat *stare);
int mknod(const char *nume, unsigned int mod, dev_t dev);
int chown(const char *nume, unsigned int proprietar,
          unsigned int grup);
int access(const char *nume, int permisiuni);
int rename(const char *numevechi, const char *numenou);
```

- `chdir` schimbă directorul curent în cel specificat prin `nume`. Pe sistemele BSD şi Unix System V release 4 mai este prezent şi apelul `fchdir` care face acelaşi lucru ca şi `chdir`, doar că directorul este specificat printr-un descriptor de fişier deschis, nu prin `nume`.
- `getcwd` copiază în zona de memorie indicată de `mem`, de lungime `dimensiune` octeţi, calea absolută a directorului curent. Dacă calea absolută a directorului are lungimea mai mare decât `dimensiune`, se returnează `NULL` şi `errno` primeşte valoarea `ERANGE`.
- `mkdir` creează un nou director, având calea şi numele specificate prin `nume` şi drepturile indicate prin întregul `drepturi`, din care se reţin numai primii 9 biţi.
- `rmdir` şterge directorul specificat prin `nume` (acest director trebuie să fie gol).
- `unlink` şterge fişierul specificat prin `nume`.
- `link` creează o legătură hard cu numele `numenou` spre un fişier existent, `numevechi`. Numele nou se poate folosi în locul celui vechi în toate operaţiile şi nu se poate spune care dintre cele două nume este cel iniţial.
- `symlink` creează o legătură simbolică (soft) cu numele `numenou` spre un fişier existent, `numevechi`. O legătură soft este doar o referinţă la un fişier existent sau

inexistent. Dacă șterg o legătură soft se șterge doar legătura, nu și fișierul original, pe când dacă șterg o legătură hard, se șterge fișierul original.

- `chmod` atribuie drepturile de acces `drepturi` la fișierul specificat prin nume. Pe sistemele BSD și Unix System V release 4 mai este prezent și apelul `fchmod` care face același lucru ca și `chmod`, doar că fișierul este specificat printr-un descriptor de fișier deschis, nu prin nume.
- `stat` depune la adresa `stare` informații privind fișierul specificat prin nume (inod-ul fișierului, drepturile de acces, id-ul proprietarului, id-ul grupului, lungimea în octeți, numărul de blocuri ale fișierului, data ultimului acces la fișier, etc. – vezi exemplul de mai jos). Sistemele BSD și Unix System V release 4 au și apelul `fstat` care face același lucru ca și `stat`, doar că fișierul este specificat printr-un descriptor de fișier deschis, nu prin nume.
- `mknod` creează un fișier simplu sau un fișier special desemnat prin nume. Parametrul `mod` specifică printr-o combinație de constante simbolice legate prin simbolul ‘|’ atât drepturile de acces la fișierul nou creat, cât și tipul fișierului care poate fi unul dintre următoarele:
 - `S_IFREG` (fișier normal)
 - `S_IFCHR` (fișier special de tip caracter)
 - `S_IFBLK` (fișier special de tip bloc)
 - `S_IFIFO` (*pipe* cu nume sau FIFO – vezi capitolul 5)
 - `S_IFSOCK` (Unix domain socket – folosit pentru comunicarea locală între procese)

Dacă tipul fișierului este `S_IFCHR` sau `S_IFBLK`, atunci `dev` conține numărul minor și numărul major al fișierului special nou creat; altfel, acest parametru se ignoră.

- `chown` schimbă proprietarul și grupul din care face parte proprietarul unui fișier specificat prin nume. Noul proprietar al fișierului va fi cel indicat de parametrul `proprietar`, iar noul grup al fișierului va fi cel indicat de parametrul `grup`. Pe sistemele BSD și Unix System V release 4 mai este prezent și apelul `fchown` care face același lucru ca și `chown`, doar că fișierul este specificat printr-un descriptor de fișier deschis, nu prin nume.
- `access` verifică dacă procesul curent are dreptul specificat de `permisiuni` relativ la fișierul specificat prin nume. `permisiuni` va conține una sau mai multe valori legate prin ‘|’ dintre următoarele: `R_OK` - citire, `W_OK` - scriere, `X_OK` – execuție și `F_OK` – existență fișier. Verificarea se face cu UID-ul și GID-ul reale ale procesului, nu cele efective (vezi capitolul 5).
- `rename` redenumeste fișierul specificat de `numevechi` în `numenou`.

4.4.2 *Creat, truncate, readdir*

Următoarele trei apeluri nu au corespondent direct printre comenzile Shell:

```
int creat(const char *pathname, unsigned int mod);
int truncate(const char *nume, long int lung);

#include <dirent.h>
struct dirent *readdir(DIR *dir);
```

Apelul sistem `creat` este echivalent cu următorul apel sistem:

```
open (const char *nume, O_CREAT|O_WRONLY|O_TRUNC);
```

Apelul sistem `truncate` trunchiază fișierul specificat prin nume la exact lung octeți.

Funcția `readdir` nu este o funcție sistem, ci este funcția POSIX pentru parcurgerea subdirectoarelor și fișierelor dintr-un director. Ea întoarce într-o structură de tipul `dirent` următorul subdirector sau fișier din directorul dat de parametrul `dir`. Această funcție încapsulează de fapt apelul sistem `getdents`. Alte funcții în legătură cu `readdir` și specificate de standardul POSIX sunt: `opendir`, `closedir`, `rewinddir`, `scandir`, `seekdir` și `telldir`. Nu intrăm în detalierea acestor funcții deoarece ele nu sunt funcții sistem.

Majoritatea apelurilor de mai sus întorc valoarea 0 în caz de succes și -1 (plus setarea corespunzătoare a variabilei `errno`) în caz de eroare. Excepție de la această regulă fac apelurile: `getcwd` care returnează NULL în caz de eroare și setează corespunzător `errno` și `readdir` care întoarce NULL în caz de eroare.

4.4.3 Un exemplu: obținerea tipului de fișier prin apelul sistem `stat`

În această secțiune vom da un exemplu de utilizare a lui `stat`. Exemplul se referă la afișarea tipului de fișier pentru fișierele ale căror nume sunt date ca argumente la linia de comandă.

Programul `tipfis.c` din fig. 4.10 exemplifică folosirea apelului sistem `stat` pentru determinarea tipului de fișier recunoscut de către sistem. Tipurile de fișiere le-am prezentat într-o secțiune precedentă.

```
/* tipfis.c:
 * Tipareste tipurile fisierelor date in linia de comanda
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

main (int argc, char *argv[]) {
    int i;
    struct stat statbuff;
    char tip[40];
    strcpy(tip, "");
    for (i = 1; i < argc; i++) {
        printf ("%s: ", argv[i]);
        if (stat (argv[i], &statbuff) < 0)
            fprintf (stderr, "Eroare stat");
        switch (statbuff.st_mode & S_IFMT) {
            case S_IFDIR:
                strcpy(tip, "Director");
                break;
            case S_IFCHR:
                strcpy(tip, "Special de tip caracter");
                break;
            case S_IFBLK:
                strcpy(tip, "Special de tip bloc");
                break;
            case S_IFREG:
                strcpy(tip, "Obisnuit");
                break;
            case S_IFLNK:

```

```

        /* acest test nu va fi adevărat niciodată deoarece stat
        * verifică în cazul unei legături simbolice, fişierul
        * pe care îl referă legătura şi nu legătura în sine.
        Il
        * scriem aici pentru completitudine. Ca să verificăm
        * tipul unei legături simbolice putem folosi funcţia
        * lstat asemănătoare cu stat şi disponibilă pe
        * versiunile BSD şi Unix System V.
        */
        strcpy(tip, "Legatura simbolica");
        break;
    case S_IFSOCK:
        strcpy(tip, "Socket");
        break;
    case S_IFIFO:
        strcpy(tip, "FIFO");
        break;
    default:
        strcpy(tip, "Tip necunoscut");
    } //switch
    printf ("%s\n", tip);
} //for
} //main

```

Figura 4.10 Sursa `tipfis.c`

4.5 Alte apeluri sistem

În acest subcapitol, vom prezenta câteva apeluri sistem care pot fi utile programatorului Unix şi care nu au mai fost prezentate până aici. Înainte de prezentarea fiecărui prototip vom scrie şi directiva `#include` necesară la începutul sursei programului C. În unele cazuri apar două astfel de directive, iar programatorul o va alege, încercând, pe cea care este potrivită pentru varianta lui de Unix.

4.5.1 Time

```

#include <time.h>
time_t time(time_t *t);

```

Apelul sistem `time` returnează timpul scurs de la 1 Ianuarie 1970 00:00 UTC, măsurat în număr de secunde. Dacă `t` este nenul, numărul de secunde va fi salvat la adresa referită de `t`.

4.5.2 Umask

```

#include <sys/types.h>
#include <sys/stat.h>
unsigned int umask(unsigned int masca);

```

Apelul sistem `umask` setează masca drepturilor pentru fişierele nou create la valoarea dată de `masca & 0777`. Această mască de biţi indică drepturile de acces care nu sunt setate implicit la crearea unui fişier. Valoarea anterioară a lui `umask` este returnată.

4.5.3 *Gethostname*

```
#include <unistd.h>
int gethostname(char *name, int lung);
```

Apelul sistem `gethostname` pune la adresa dată de `name`, pe lungime de maxim `lung` octeți, numele mașinii (calculatorului).

4.5.4 *Gettimeofday*

```
#include <sys/time.h>
int gettimeofday(struct timeval *tv, 0);
```

Apelul sistem `gettimeofday` pune în structura `tv`, de tip `timeval`, valoarea timpului curent. Structura `timeval` este definită în felul următor:

```
struct timeval {
    long    tv_sec;    /* numărul de secunde */
    long    tv_usec;  /* numărul de microsecunde */
};
```

4.5.5 *Mmap și munmap*

```
#include <sys/mman.h>
void* mmap(void *start, size_t lung, int prot,
           int atribut, int fd, off_t offset);
int munmap(void *start, size_t lung);
```

Apelul sistem `mmap` mapează `lung` octeți începând de la deplasamentul `offset` din fișierul (sau alt obiect) indicat de descriptorul de fișier `fd` în memorie, preferabil la adresa `start`. În general, `start` are valoarea 0 și adresa unde are loc maparea este returnată de `mmap`. Apelul sistem `munmap` șterge maparea de la adresa `start`, pe lungime `lung`. Parametrul `prot` specifică protecția de memorie dorită și poate avea valorile:

- `PROT_EXEC` - paginile de memorie pot fi executate
- `PROT_READ` - paginile de memorie pot fi citite
- `PROT_WRITE` - paginile de memorie pot fi modificate
- `PROT_NONE` - paginile de memorie nu pot fi accesate

Parametrul `atribut` specifică tipul de obiect mapat, opțiuni de mapare și dacă modificări ale conținutului memoriei unde a avut loc maparea sunt vizibile numai procesului curent sau și altor procese. Cele mai importante valori ale sale sunt: `MAP_FIXED`, `MAP_SHARED`, `MAP_PRIVATE`.

4.5.6 *Fsync și fdatasync*

```
#include <unistd.h>
int fsync(int fd);
int fdatasync(int fd);
```

Apelul sistem `fsync` scrie pe disc datele fişierului `fd` modificate în buffer-ele nucleului, dar nesalvate încă pe disc. Diferența dintre `fsync` și `fdatasync` este că `fdatasync` nu salvează și date de control despre fişier ca timpul ultimului acces la fişier, ci doar conținutul fişierului.

4.5.7 *Uname*

```
#include <sys/utsname.h>
int uname(struct utsname *buf);
```

Apelul sistem `uname` returnează în structura `buf` de tip `utsname` date despre nucleul sistemului de operare. Tipul `utsname` are următoarele câmpuri:

```
struct utsname {
    char sysname[];
    char nodename[];
    char release[];
    char version[];
    char machine[];
#ifdef _GNU_SOURCE
    char domainname[];
#endif
};
```

Aceste câmpuri au aceleași sensuri ca și câmpurile afișare de comanda shell `uname`.

4.5.8 *Select și seturi de descriptori*

```
#include <sys/select.h>
int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Apelul sistem `select` monitorizează într-un interval de timp dat de `timeout` starea descriptorilor de fişiere din seturile `readfds`, `writefds` și `exceptfds` și raportează (returnează) câte dintre ele și-au modificat starea. Descriptorii din `readfds` vor fi monitorizați pentru a vedea dacă au apărut caractere noi disponibile pentru citire, cei din `writefds` pentru a vedea dacă o nouă operație de scriere nu se va bloca, iar cei din `exceptfds` vor fi monitorizați pentru excepții. Parametrul `n` trebuie să aibă ca valoare cel mai mare număr de descriptor din cele trei seturi plus 1.

Standardul POSIX specifică și patru macrouri pentru a manipula cele trei seturi de descriptori. Prototipurile lor sunt date mai jos, unde prin **fd** am notat un descriptor de fişiere, iar prin **set** unul dintre seturile de descriptori din **select**:

```
FD_CLR(int fd, fd_set *set); //sterge fd din set
FD_ISSET(int fd, fd_set *set); //este fd in set?
FD_SET(int fd, fd_set *set); //adauga fd la set
FD_ZERO(fd_set *set); //goleste set
```

Apelul sistem **select** multiplexează de fapt mai multe canale de intrare-ieșire sincrone.

4.6 Blocarea fișierelor

4.6.1 Un (contra)exemplu

Una din problemele ce apar frecvent în medii concurente este partajarea resurselor. Să considerăm următoarea situație: se dă un fișier cu numele `secv` care conține pe prima linie un număr întreg de 5 cifre, reprezentate în ASCII. Să considerăm că fiecare proces (deocamdată prin proces vom înțelege program aflat în execuție) face următoarele acțiuni:

- Citește acest număr din `secv`.
- Mărește numărul cu o unitate.
- Rescrie numărul rezultat în `secv`.

Programul `lockfile.c` din fig. 4.11 realizează aceste acțiuni. Funcțiile `my_lock(fd)` și `my_unlock(fd)` au sarcina de a asigura accesul exclusiv la fișierul `secv` pe durata acțiunii. Într-o primă variantă, comentăm apelurile celor două funcții.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

main () {
    int  fd, i, n, nrsecv;
    char buf[6];
    fd = open ("secv", O_RDWR);
    for (i = 0; i < 10000; i++) {

        // my_lock(fd);          Blocheaza fisierul

        lseek (fd, 0L, 0);      //pozitionare la inceput
        n = read (fd, buf, 5);  //citim numarul
        buf[n] = '\0';          //zeroul terminal
        sscanf (buf, "%d\n", &nrsecv);
                                //convertim stringul citit in numar
        printf ("pid = %d, secventa = %d\n", getpid(), nrsecv);
        nrsecv++;               //incrementeaza secventa
        sprintf (buf, "%5d\n", nrsecv);
                                //convertim numarul in string
        lseek (fd, 0L, 0);      //revenire inainte de scriere
        write (fd, buf, strlen(buf)); //scriem numarul inapoi

        // my_unlock(fd);      Deblocheaza fisierul
    }
}
```

Figura 4.11 Sursa `lockfile.c`

Înainte de rularea programului, se va crea cu un editor de texte fișierul `secv` în care se va introduce o singură linie conținând numărul 1. Se compilează programul și să presupunem că fișierul executabil are numele `a.out`. Se rulează apoi programul, simultan în două procese, folosind comanda:

```
$ a.out & a.out &
```


Cele două procese vor afişa, intercalat, linii pe ieşirea standard. Ultima linie afişată va fi ceva de forma:

```
pid = 1265 secventa = 11953
```

Rezultatul pare ciudat, cel puţin la prima vedere. În mod normal, valoarea numărului de secvenţă ar trebui să fie 20000! De ce nu se întâmplă așa? Deoarece prin comentarea apelului funcţiilor `my_lock` și `my_unlock` secvenţa dintre ele poate fi executată, total sau parţial, în acelaşi moment de către cele două procese! Frecvent se va întâmpla că ambele vor citi din fişier aceeaşi valoare, o vor incrementa și o vor scrie. În consecinţă valoarea va fi mărită doar cu o unitate, deşi au acţionat asupra ei două procese!

4.6.2 Tipuri de blocare

Exemplul din secţiunea precedentă ilustrează faptul că în anumite condiţii se impune ca asupra unui fişier sau asupra unei porţiuni din fişier, să aibă dreptul să acţioneze doar un singur proces. Această restricţie este cunoscută sub numele de *blocarea* unui fişier. Sub Unix sunt utilizate două feluri de blocare: conciliantă și obligatorie.

Blocare conciliantă (*advisory*) este atunci când sistemul ştie care fişiere au fost blocate și de către cine, iar procesele cooperează, prin funcţii de tip `my_lock` și `my_unlock`, la accesarea fişierului. Nucleul sistemului de operare nu previne situaţia în care un proces indisciplinat scrie în fişierul blocat.

Blocarea obligatorie (*mandatory*) are loc atunci când sistemul verifică la fiecare scriere și citire dacă fişierul este blocat sau nu. Pentru a permite blocarea obligatorie, trebuie invalidat dreptul de execuţie al grupului și bitul `set-group-ID` să fie 1 pentru fişierul respectiv, iar la montarea sistemului de fişiere (prin `mount`) să se permită blocare obligatorie.

Blocarea unui fişier înseamnă blocarea accesului la orice octet din fişier.

Blocarea unui articol înseamnă blocarea accesului la un număr de octeți consecutivi din fişier.

Mecanismele de blocare a fişierelor sub Unix specificate de standardul POSIX (și implementate de biblioteca C de la GNU) permit, în esenţă, două tipuri de blocări, fiecare dintre ele putând fi conciliante sau obligatorii (implicit conciliante) în funcţie de îndeplinirea sau nu a condițiilor de mai sus:

- blocare *exclusivă*, atunci când un singur proces are acces la fişier sau porţiunea din fişier. De obicei, este vorba aici de o operaţie de scriere în fişier, iar pe durata pregătirii operaţiei și a scrierii propriu-zise, nici un alt proces nu poate nici scrie, nici citi porţiunea rezervată.
- blocare *partajată*, atunci când porţiunea rezervată poate fi citită, simultan, de către mai multe procese, dar nici un proces nu scrie.

Aceste tipuri de blocare corespund rezolvării unei probleme celebre de programare concurentă - problema cititorilor și a scriitorilor.

4.6.3 Blocarea conciliantă prin fcntl

Nucleele Unix oferă mai multe apeluri sistem destinate blocării. De exemplu, sub Linux și FreeBSD există o funcție sistem `flock` pentru blocare. De asemenea, biblioteca C de la GNU oferă funcțiile `flock` și `lockf`. Toate fac uz de serviciile apelului sistem `fcntl` pentru realizarea blocării. Reamintim cea de-a treia formă a prototipului `fcntl`, folosită pentru blocare:

```
int fcntl(int fd, int cmd, struct flock *lock);
```

Parametrul `fd` este descriptorul fișierului supus blocării.

Parametrul `lock` este un pointer la o structură de tip `flock`, prin care sunt indicați parametrii operației de blocare. Această structură ce conține cel puțin următoarele câmpuri:

```
struct flock {
    - - -
    short l_type;
    short l_whence; /* cum se interpretează l_start */
    off_t l_start; /* offset-ul de început */
    off_t l_len; /* numărul de octeți */
    pid_t l_pid; /* PID-ul procesului care blochează blocajul
                  curent */
    - - -
};
```

Câmpul `l_type` indică tipul operației de blocare:

- `F_WRLCK` indică faptul că este vorba de o blocare exclusivă - blocare la scriere. Un singur proces poate efectua scriere în porțiunea rezervată, iar toate celelalte procese solicitante sunt în așteptare.
- `F_RDLCK` indică faptul că este vorba de o blocare partajată, procesele ce solicită porțiunea rezervată pentru citire își pot efectua operațiile, în timp ce procesele scriitori stau în așteptare.
- `F_UNLCK` indică ridicarea stării de blocare, lăsând eventualele alte procese ce doresc blocarea să o facă.

Câmpul `l_whence` indică locul începând de unde se determină localizarea porțiunii rezervate. Valorile lui sunt aceleași cu cele de la funcția `lseek`:

- `SEEK_SET` indică reperarea porțiunii rezervate față de începutul fișierului.
- `SEEK_CUR` indică reperarea porțiunii rezervate față de poziția curentă a pointerului fișierului.
- `SEEK_END` indică reperarea porțiunii rezervate față de sfârșitul fișierului.

Câmpul `l_start` indică începutul porțiunii rezervate, numărat în octeți față de locul indicat prin `l_whence`. Acest câmp poate avea atât valori pozitive, cât și valori negative. În consecință, începutul zonei se specifică prin combinația parametrilor `l_whence` și `l_start`.

Câmpul `l_len` indică lungimea în octeți a zonei rezervate, un număr nenegativ. Dacă valoarea este strict pozitivă atunci sunt rezervați pentru blocare un număr de `l_len` octeți de la începutul zonei rezervate. Dacă `l_len` are valoarea zero, atunci sunt rezervați toți octeții de la începutul zonei până la sfârșitul fișierului.

Câmpul `l_pid` conține PID-ul procesului care ține blocată regiunea (are sens numai în cazul comenzii `F_GETLK`).

Parametrul `cmd`, în acțiunile de blocare, poate avea următoarele valori:

- `F_SETLKW` indică faptul că procesul cooperează la blocarea fișierului. Dacă fișierul este blocat exclusiv (la scriere) de către un alt proces, atunci procesul curent intră în așteptare până la deblocarea fișierului. Acțiunea de blocare pe care o execută, după eventuala ieșire din starea de așteptare, este cea dictată de parametrul `lock`.
- `F_SETLK` este similar cu `F_SETLKW`, numai că în cazul unei blocări exclusive procesul nu mai intră în așteptare, ci întoarce `-1` și setează `errno` la `EACCES` sau `EAGAIN`.
- `F_GETLK` întoarce la adresa `lock` starea de blocare în care se află fișierul.

În fig. 4.12 sunt prezentate sursele funcțiilor `my_lock` și `my_unlock` în care se folosește blocarea și deblocarea prin `fcntl`.

```
void my_lock(int fd) {
    struct flock lock;
    lock.l_type = F_WRLCK; //blocat exclusiv (la scriere)
    lock.l_whence = SEEK_SET; //baza blocarii (inceputul fisierului)
    lock.l_start = 0; //offset blocare fata de baza
    lock.l_len = 0; //lungimea blocarii, tot fisierul
    fcntl(fd, F_SETLKW, &lock); //comanda blocarea
} //my_lock

void my_unlock(int fd) {
    struct flock lock;
    lock.l_type = F_UNLCK; //deblocheaza
    lock.l_whence = SEEK_SET; //incepand de la inceputul fisierului
    lock.l_start = 0; // pozitia 0 fata de whence
    lock.l_len = 0; //tot fisierul
    fcntl(fd, F_SETLKW, &lock); //comanda deblocarea
} //my_unlock
```

Figura 4.12 funcțiile `my_lock` și `my_unlock` folosind `fcntl`

4.6.4 Blocare prin `lockf` și `flock`

Prototipurile celor două funcții sunt:

```
int lockf(int fd, int actiune, long lungime);
int flock(int fd, int actiune);
```

Apelul `lockf` realizează, în funcție de condițiile specificate mai sus, fie blocare conciliantă, fie obligatorie (implicit, blocare conciliantă). De asemenea, poate bloca un fișier întreg sau numai o parte din el. Apelul `flock` blochează numai conciliant și numai întregul fișier.

actiune pentru `lockf` este una dintre următoarele:

- `F_ULOCK` deblocarea unei regiuni blocate
- `F_LOCK` blocarea unei regiuni
- `F_TLOCK` testarea și blocare dacă nu este deja blocată
- `F_TEST` testarea unei regiuni dacă este blocată sau nu

actiune pentru `flock` este una dintre următoarele:

- `LOCK_SH` face blocare partajată (acces din mai multe procese)

- LOCK_EX face blocare exclusivă
- LOCK_UN face deblocare
- LOCK_NB în general, dacă se cere blocarea și fișierul este deja blocat, atunci programul așteaptă la apelul flock până când poate realiza blocarea cerută. Legarea acestui atribut, cu sau (), de una dintre acțiunile anterioare, face ca în această situație să nu se mai aștepte.

fd este descriptorul de fișier.

lungime spune câți octeți vor fi blocați în fișier începând cu poziția curentă. Dacă lungime=0 atunci se blochează întregul fișier.

În caz de erori, funcția întoarce -1 și poziționează variabila globală sistem errno.

Reamintim că dintre cele două funcții, numai flock reprezintă o funcție sistem implementată de nucleu (pe care o apelăm prin intermediul funcției POSIX cu același nume implementată de biblioteca C de la GNU – menționăm din nou că funcția flock a bibliotecii C de la GNU este o funcție wrapper pentru funcția sistem flock implementată de nucleu), apelul funcției lockf traducându-se de fapt într-un apel sistemfcntl.

Folosirea lui F_TEST este relativ nesigură, din cauză că secvența:

```
if (lockf (fd, F_TEST, size) == 0)
    rc = lockf (fd, F_LOCK, size)
```

nu este echivalentă cu secvența:

```
rc = lockf (fd, F_TLOCK, size)
```

deoarece între F_TEST și F_LOCK ar putea interveni un alt proces care să blocheze.

În fig. 4.13 sunt prezentate sursele my_lock și my_unlock realizate folosind lockf.

```
void my_lock (int fd) {
    lseek (fd, 0L, 0);
    lockf (fd, F_LOCK, 0L);
} //my_lock

void my_unlock (int fd) {
    lseek (fd, 0L, 0);
    lockf (fd, F_UNLOCK, 0L);
} //my_unlock
```

Figura 4.13 funcțiile my_lock și my_unlock folosind lockf