4 Sistemul de fişiere Unix

4.1 Structura arborescentă şi legături suplimentare

4.1.1 Tipuri de fişiere şi sisteme de fişiere

In cadrul unui sistem de fişiere, apelurile sistem Unix gestionează opt tipuri de fişiere şi anume:

1. Normale (obişnuite)
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 entităţi, pe care apelurile sistem le văd, din punct de vedere sintactic, tot ca şi fişiere. Aceste entităţi sunt gestionate de nucleul Unix, au suportul fizic tot în nucleu şi folosite la comunicări între procese. Aceste entităţi sunt:

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

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

Fişierele obişnuite sunt privite ca şiruri de octeţi, accesul la un octet putându-se face fie secvenţial, fie direct prin numărul de ordine al octetului.

Fişierele directori. Un fişier director se deosebeşte de un fişier obişnuit numai prin informaţia conţinută în el. Un director conţine lista de nume şi adrese pentru fişierele subordonate lui. Uzual, fiecare utilizator are un director propriu care punctează la fişierele lui obişnuite, sau la alţi subdirectori definiţi de el.

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

Fiecare director are două intrări cu nume speciale şi anume:

`"."` (punct) denumeşte generic (punctează spre) însuşi directorul respectiv;
`".."` (două puncte succesive), denumeşte generic (punctează spre) directorul părinte.

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

In mod obişnuit, fiecare utilizator foloseşte un *director curent*, ataşat utilizatorului la intrarea în sistem. Utilizatorul poate să-şi schimbe acest director (`cd`), poate crea un nou director subordonat celui curent, (`mkdir`), să şteargă un director (`rmdir`), să afişeze *calea* de acces de la `root` la un director sau fişier (`pwd`) etc.

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

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

Actualele distribuţii de Unix permit utilizatea unor sisteme de fişiere proprii altor sisteme de operare. Printre cele mai importante amintim:

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

Din fericire, aceste extinderi sunt transparente pentru utilizatorii obişnuiţi. Totuşi, se recomandă prudenţă atunci când se efectuează altfel de operaţii decât citirea din fişierele create sub alte sisteme de operare decât sistemul curent. De exemplu, modificarea sub Unix a unui octet într-un fişier de tip `doc` creat cu Word sub Windows poate uşor să compromită fişierul aşa încât el să nu mai poată fi exploatat sub Windows!

Administratorii sistemelor Unix trebuie să ţină cont de sistemele de fişiere pe care le instalează şi de drepturile pe care le conferă acestora vis-a-vis de userii obişnuiţi.

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

4.1.2 Legături hard şi legături simbolice

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

Legăturile hard sunt identice cu legăturile naturale şi 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 fişiere 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 fişier primeşte efectiv două nume. Din această cauză, la parcurgerea unei structuri arborescente, fişierele punctate prin legături hard apar duplicate. Fiecare duplicat apare cu numărul de legături către el.

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

```
#ln  vechi  linknou
```

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

Legăturile hard pot fi făcute numai în interiorul aceluiaşi sistem de fişiere (detalii puţin mai târziu).

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

In 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 şi de către userii obişnuiţi. De asemenea, ele pot puncta şi înafara sistemului de fişiere (detalii puţin mai târziu).

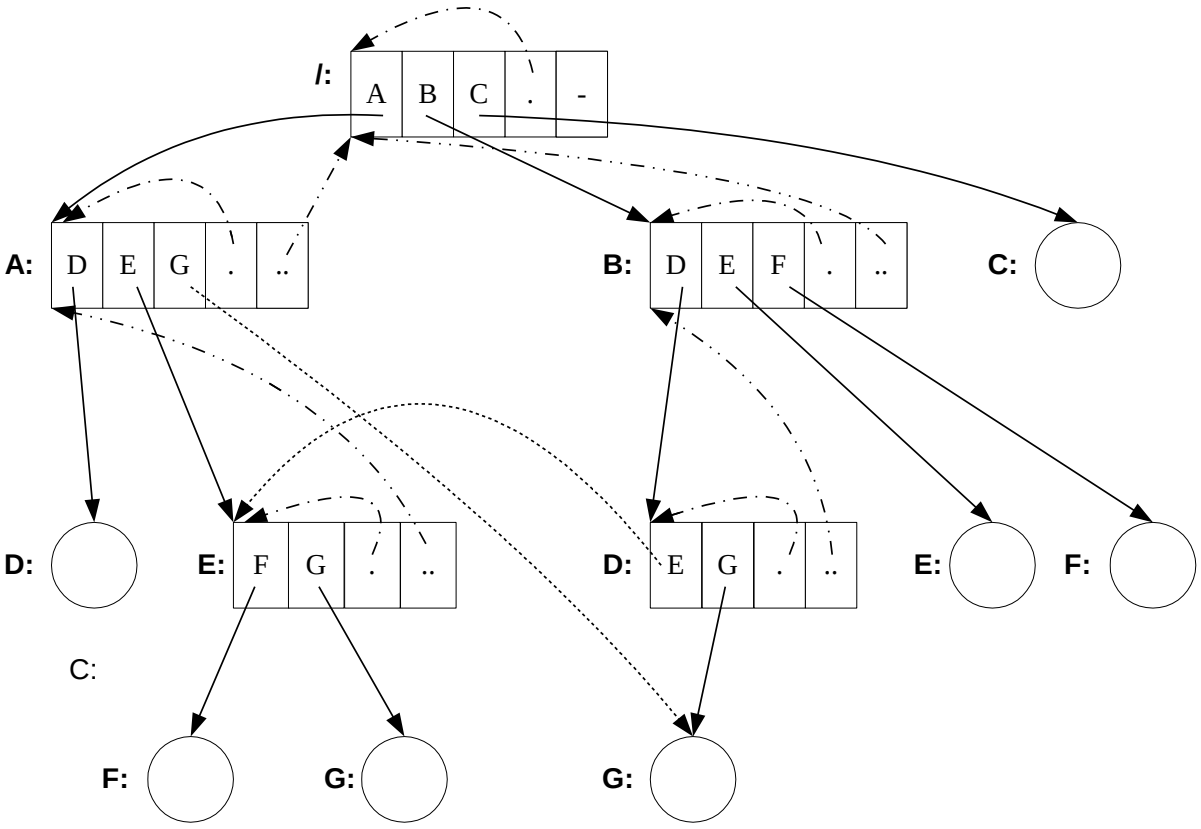

Figura 4.1 O structură arborescentă cu legături

Structura arborescentă împreună cu legăturile simbolice sau hard conferă sistemului de fişiere Unix o structură de graf aciclic. In fig. 4.1 este prezentat un exemplu simplu de structură de fişiere. Prin literele mari A, B, C, D, E, F, G am indicat nume de fişiere obişnuite, nume de directori şi nume de legături. Este evident posibil ca acelaşi nume să apară de mai multe ori în structura de directori, graţie structurii de directori care elimină ambiguităţile. Fişierele obişnuite sunt reprezentate prin cercuri, iar fişierele directori prin dreptunghiuri.

Legăturile sunt reprezentate prin săgeţi de trei tipuri:

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

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

Să presupunem că cele două legături (desenate cu linie punctata) 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 subordonaţii lui de la stânga spre dreapta. Următoarele 12 linii indică toate cele 12 noduri din structură. Pe aceeaşi linie apar, atunci când este posibil, mai multe specificări ale aceluiaşi 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 ştergerea 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 şters 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 şi creşte cu 1 la fiecare nouă legătură. La ştergere, se radiază legătura din directorul părinte care a cerut ştergerea, iar contorul de legare scade cu 1. Abia dacă acest contor a ajuns la zero, fişierul va fi efectiv şters de pe disc şi 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 fişierelor Unix *nu apare zona de periferic.* Acest fapt nu este întâmplător, ci este cauzat de filozofia generală de acces la fişierele Unix. Conceptul esenţial prin care se rezolvă această problemă este cunoscut în Unix prin termenii de *montare* şi *demontare* a unui sistem de fişiere.

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

```
# mount  [ optiuni ]  sistemDeFisiere directorDeMontare
```

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

Operaţia de *demontare* are efectul invers şi ea se face cu comanda:

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

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

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

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

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

- Nu se va cere accesul la un fişier de pe un disc decât dacă acesta este montat în structura de fişiere implicită.
- Nu se va cere demontarea unei substructuri decât dacă a fost montată în prealabil.
- *Scoaterea unui suport montat de pe o unitate şi 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 şi se introduce alta în loc, atunci este posibil să se piardă informaţii de pe noua dischetă, şi, 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 operaţia `unmount`.

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

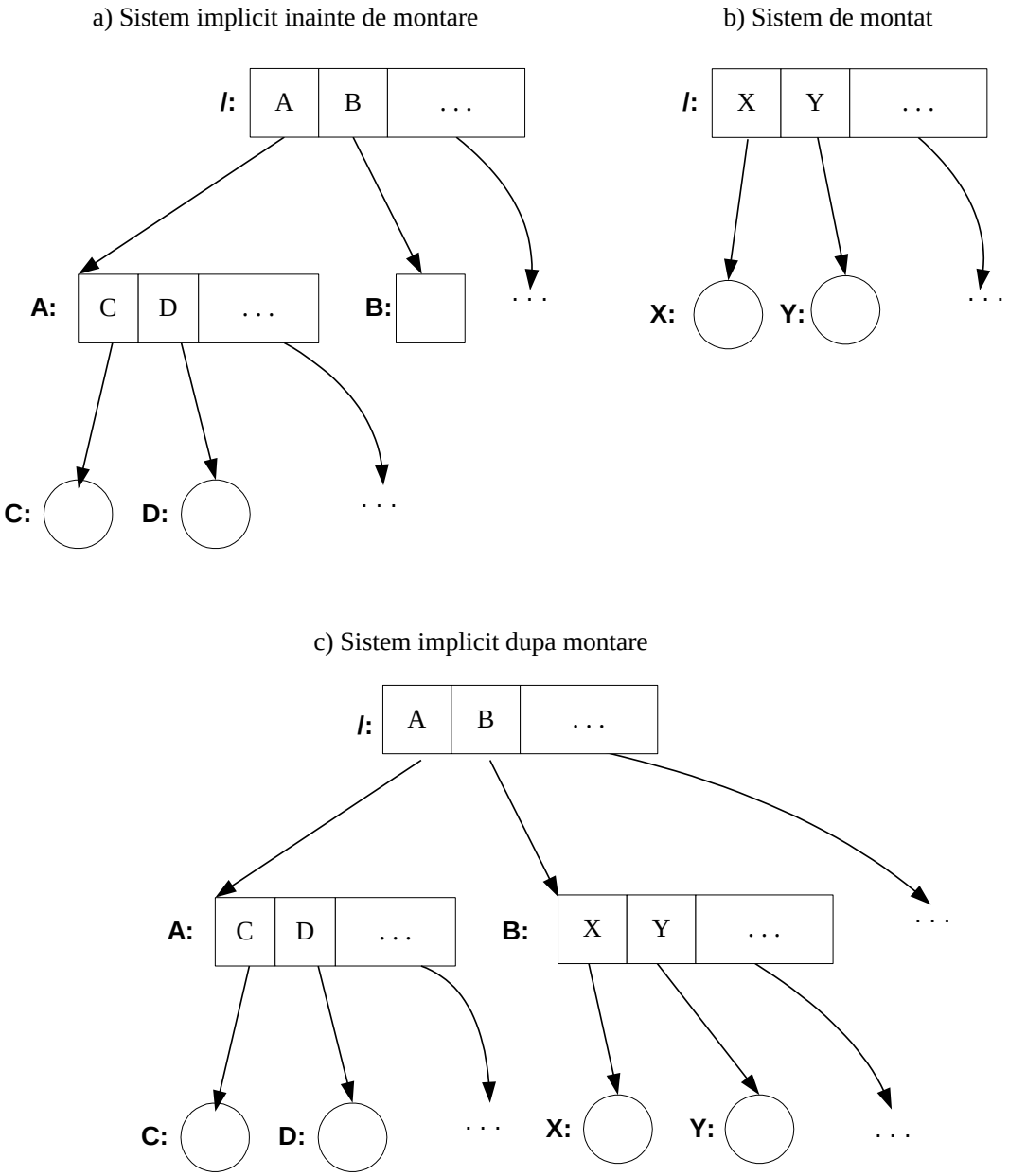

Figura 4.2 Operaţia de montare

In practica implementărilor Unix, la încărcarea **SO** se fac automat o serie de operaţii de montare, în conformitate cu configurarea sistemului. Indicaţiile 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:

- partiţia Unix (`periferic`, `sistemDeFisiere`) care se montează;
- `directorDeMontare` sub care este montată partiţia;
- tipul sistemului de fişiere conţinut;
- diverse opţiuni.

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 practicată 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
```

In 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: useri, fişiere, procese şi mai ales reamintim faptul că între ele există multe interdependenţe. In 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 fişierelor este *root* sau superuserul.

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

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

In consecinţă, pentru specificarea drepturilor de mai sus asupra unui fişier sau director sunt necesari 9 (nouă) biţi. Reprezentarea externă a acestei configuraţii se face printr-un grup de 9 (nouă) caractere: `rwxrwxrwx` în care absenţa 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 $o_1o_2o_3$ sunt trei cifre octale. Biţii 1 ai primeia indică drepturile userului, biţii 1 ai celei de-a doua indică drepturile grupului, iar biţii 1 ai celei de-a treia cifră indică drepturile restului userilor. De exemplu, comanda:

```
chmod  754  A B
```

fixează la fişierele `A` şi `B` toate drepturile pentru user, drepturile de citire şi de execuţie pentru grup şi doar dreptul de citire pentru ceilalţi useri. In urma acestei comenzi, drepturile fişielor `A` şi `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

In momentul intrării unui user în sistem, acestuia i se vor acorda nişte drepturi implicite pentru fişiere nou create. Toate fişierele şi 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 măştii se lansează:

```
umask
```

Rezultatul este afişarea măştii, de regulă 022. Deci drepturile implicite vor fi: 777-022=755, adică userul are toate drepturile, iar grupul şi restul vor avea doar drepturi de citire şi de execuţie.

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

In 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--`.

In 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`. In 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. (Intr-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 fişier executabil bitul `setuid` este 1, atunci **un utilizator care lansează în execuţie acest fişier** (evident, dacă are dreptul să-l lanseze) **primeşte, pe timpul execuţiei, aceleaşi drepturi de acces la resurse** (fişiere, semafoare, zone de memorie etc.) **ca şi proprietarul fişierului executabil.**

Să vedem o situaţie concretă în care este utilă folosirea bitului `setuid`. Un utilizator cu numele **`profesor`** întreţine un fişier **`note`** al cărui proprietar este. Din raţiuni lesne de înţeles, drepturile fişierului `note` sunt fixate la **`rw-------`.** Utilizatorul **`profesor`** doreşte să permită utilizatorilor din grupul `studenti` să vadă unele informaţii din fişierul **`note`**.

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

Utilizatorii **`studenti`**, în momentul lansării programului `examen`, primesc aceleaşi drepturi de acces la fişiere ca şi **`profesor`**. In particular programul `examen` poate accesa fişierul **`note`** (vezi drepturile acestui fişier) chiar dacă el nu a fost lansat în execuţie de către **`profesor`**. In absenţa lui `setuid` pentru `examen`, acesta poate fi, totuşi lansat în execuţie, însă nu poate să acceseze fişierul **`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 execuţiei 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 fişierelor. In schimb, dacă superuserul gestionează prost acest mecanism, atunci potenţialii infractori au un câmp larg de acţiune. Să considerăm, din raţiuni evidente, doar un scenariu simplu: Presupunem ca `root` este proprietar al unui fişier executabil cu bitul `setuid` setat şi cu drept de scriere pentru alţii. In această situaţie, un răuvoitor poate să modifice acest fişier executabil aşa încât să aibă o acţiune malefică ce presupune acces la resurse ale superuserului!. Acţiunea se va putea executa deoarece EUID-ul este UID-ul lui `root`!

Modificarea drepturilor de acces la fişiere se poate face numai de către proprietarul fişierului (sau de către superuser), folosind comanda `chmod`. Modificarea proprietarului sau a grupului se poate face, în aceleaşi condiţii folosind comanda `chown`.

Un exemplu tipic în care se foloseşte `setuid` este comanda `/usr/bin/passwd`. Aceasta este lansată de către fiecare utilizator atunci când doreşte să-şi schimbe parola. Efectul ei se răsfrânge asupra fişierului `/etc/shadow`. In 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.

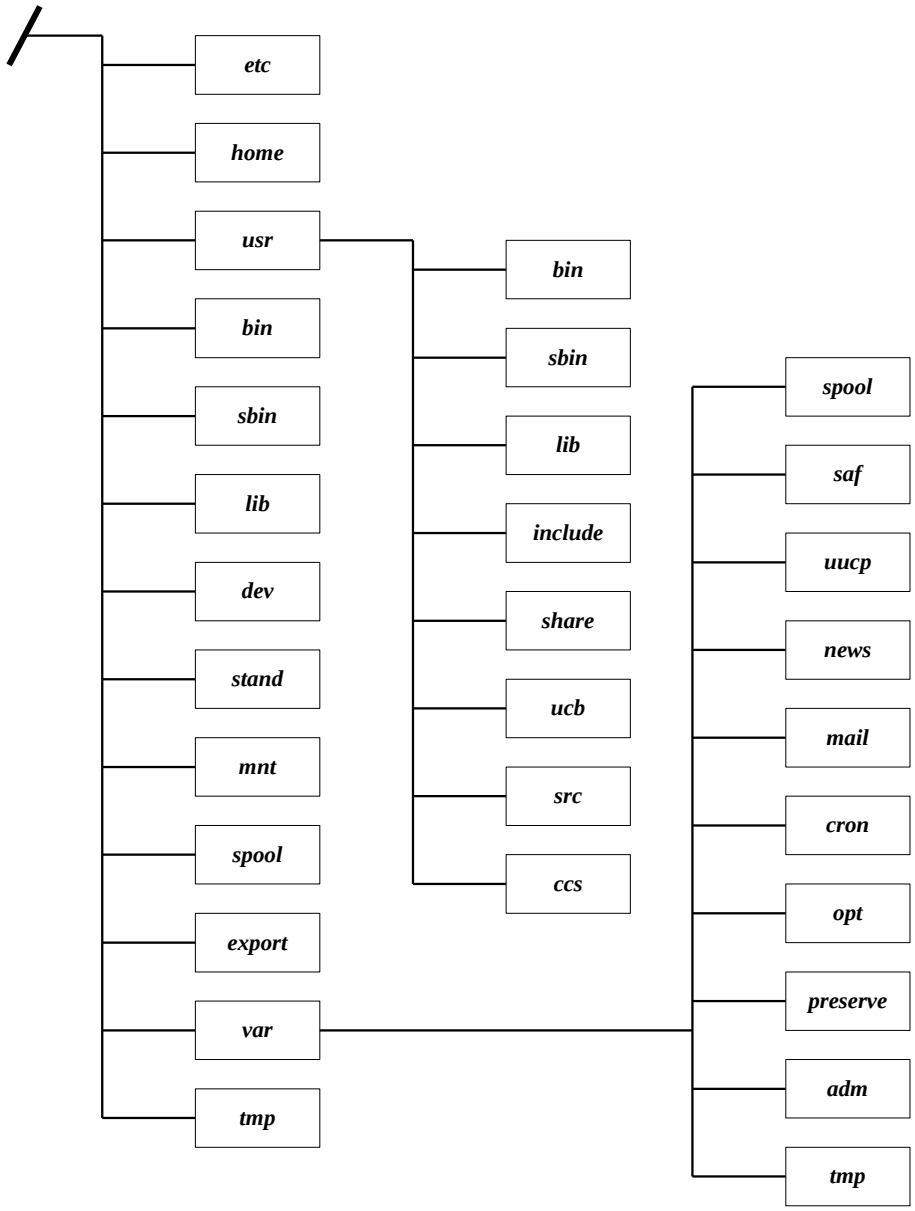

Figura 4.3 Structura superioară a unui sistem de fişiere Unix

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. In momentul intrării în sistem, fiecare utilizator indică directorul lui, care este fixat ca director curent (*home directory*) in /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. In prezent este punctul de montare pentru partiţia ce conţine usr. El contine 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, asambloare, 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 încărare 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. In 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 fişierelor care cresc dinamic. In particular, multe dintre versiunile de Unix ţin în acest director fişierele INBOX cu căsuţele poştale ale utilizatorilor. Structura de subdirectoare a lui /var o vom descrie ceva mai încolo.

Directorul /tmp este folosit pentru a memora fişiere temporare pentru aplicaţii. In mod normal, aceste fişiere nu sunt salvate şi sunt şterse după o perioadă de timp.

In continuare vom descrie pe scurt conţinutul directorului /usr. După cum se poate vedea, unele dintre subdirectoare sunt plasate (şi 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 conţine fişierele header (*.h) standard ale limbajului C de sub Unix.

Directorul /usr/share conţine o serie de directoare partajabile în reţea. In multe dintre sistemele noi, în el se află directoarele: man cu manualele Unix, src cu sursele C ale nucleului Unix şi lib mai sus prezentat.

Directorul /usr/ucb conţine programele executabile compatibile Unix BSD.

Directorul /usr/src conţine textele sursă C ale nucleului Unix de pe maşina respectivă.

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

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

Directorul /var/saf conţine fişiere jurnal şi de contabilizare a serviciilor oferite.

Directorul /var/uucp conţine programele necesare efectuării de copieri de fişiere între sisteme Unix (Unix to Unix CoPy). Acest gen de servicii este primul pachet de comunicaţii instalat pe sisteme Unix, este operaţional încă din 1978 şi este utilizat şi astăzi atunci când nu există alt mijloc mai modern de comunicaţii. Un astfel de sistem permite, de exemplu, apelul telefonic între două sisteme Unix, iar după luarea contactului cele două sisteme îşi schimbă între ele o serie de fişiere, ca de exemplu mesajele de poştă electronică ce le sunt destinate.

Directorul /var/news conţine fişierele necesare serviciului de (ştiri) noutăţi (news) care poate fi instalat pe maşinile Unix.

Directorul /var/mail conţine căsuţele poştale implicite ale utilizatorilor (INBOX). Pe unele sisteme, ca de exemplu pe Linux, acestea se află în /var/spool/mail.

Directorul /var/cron conţine fişierele jurnal necesare serviciilor executate la termen.

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

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
`/bin`	`/usr/bin`
`/lib`	`/usr/lib`
`/usr/adm`	`/var/adm`
`/usr/spool`	`/vsr/spool`
`/usr/tmp`	`/var/tmp`
`/etc/termcap`	`/usr/share/lib/termcap`
`/usr/lib/terminfo`	`/usr/share/lib/terminfo`
`/usr/lib/cron`	`/etc/cron.d`
`/usr/man`	`/usr/share/man`
`/etc/<programe>`	`/usr/bin/<programe>`
`/etc/<programe>`	`/sbin/<programe>`

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*.

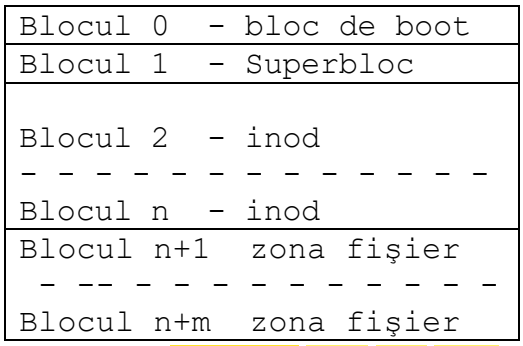

Figura 4.4 Structura unui disc Unix

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 şi memorie este *blocul*. La sistemele mai vechi acesta are 512 octeţi, iar la cele mai noi poate avea şi 1Ko sau 4Ko, pentru o mai eficientă gestiune a spaţiului.

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

Blocul 0 conţine programul de încărcare al **SO**. Acest program este dependent de maşina sub care se lucrează. Acţiunea lui se declanşează la pornirea sistemului - apăsarea butonului <***Reset***> sau <***Power***>. La acest moment, intră în lucru un mic program din memoria ROM, aşa-numitul *cod startup din BIOS.* Acesta ştie să citească primii 512 octeţi de pe disc, să îi depună undeva în memoria RAM şi să lanseze în execuţie secvenţa de octeţi citită. Evident, în primii 512 octeţi 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, execuţia programului poate continua cu părţi 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 staţiilor fără harddisk.

In 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 *partiţie* sau *segment de boot*. Această partiţie are o structură de date extrem de simplă (mult mai simplă decât a sistemului de fişiere) şi poate fi accesată foarte uşor.

Blocul 1 este numit şi *superbloc.* In el sunt trecute o serie de informaţii prin care se defineşte sistemul de fişiere de pe disc. Printre aceste informaţii amintim:

- numărul *n* de *inoduri* (detaliem imediat);
- numărul de zone definite pe disc;
- pointeri spre harta de biţi a alocării inodurilor;
- pointeri spre harta de biţi a spaţiului 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 fişier. 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 octeţi şi se numeşte *i-număr*. Acest i-număr constituie legătura dintre fişier şi programele utilizator.

Blocurile *n+1 la n+m* reprezintă *zona fişierelor*. Este partea cea mai mare din cadrul discului. Alocarea spaţiului pentru fişiere se face printr-o variantă elegantă de indexare. Informaţiile 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 fişiere.

4.2.2 Directori şi inoduri

Structura unei intrări într-un fişier 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.

In 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 conţine adresa unui bloc, numit *bloc de indirectare triplă*. In acest bloc sunt conţinute adresele a 128 blocuri de indirectare dublă, fiecare dintre acestea conţinând adresele a câte 128 blocuri de indirectare simplă, iar fiecare dintre acestea conţine adresele a câte 128 blocuri, de câte 512 octeţi, cu informaţii ale fişierului.

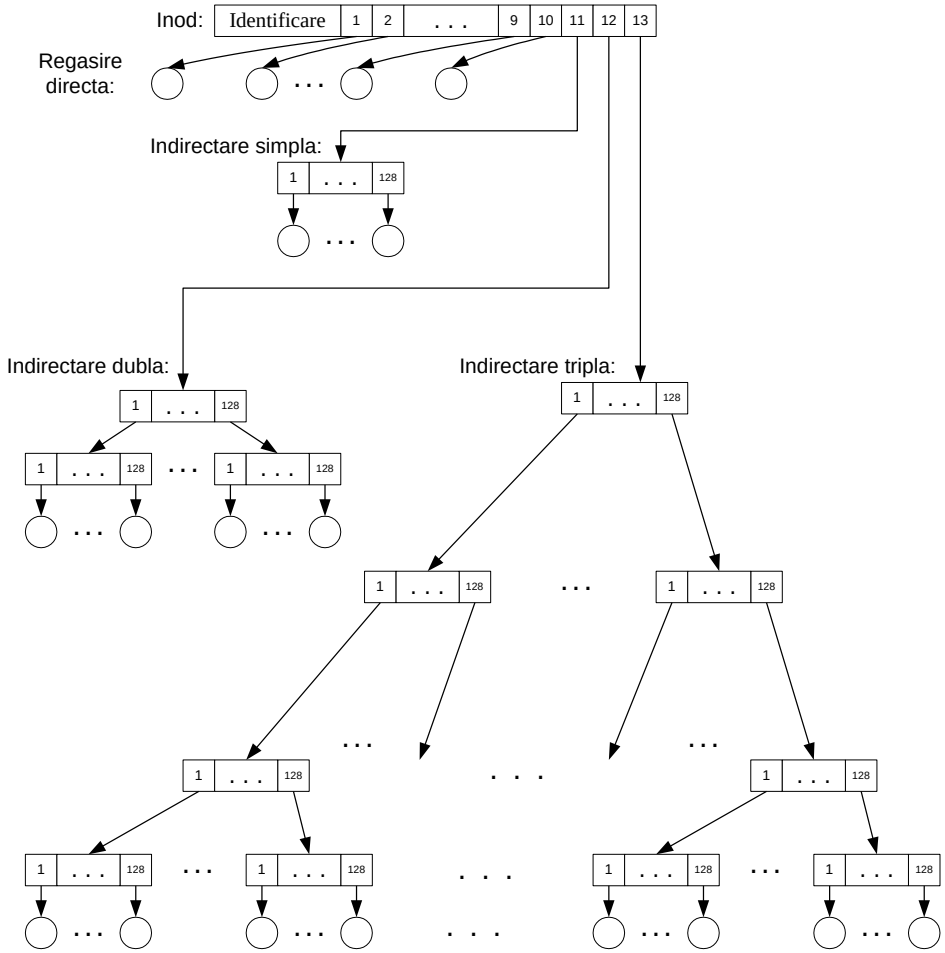

Figura 4.6 Structura unui inod şi accesul la blocurile unui fişier

In fig. 4.6 am ilustrat prin cercuri blocurile de informaţie care aparţin fişierului, iar prin dreptunghiuri blocurile de referinţe, în interiorul acestora marcând referinţele.

Numărul de accese necesare pentru a obţine direct un octet oarecare este cel mult 4. Pentru fişiere mici acest număr este şi mai mic. Atât timp cât fişierul este deschis, inodul lui lui este prezent în memoria internă. Tabelul următor dă numărul maxim de accese la disc pentru a obţine, în acces direct orice octet dintr-un fişier, în funcţie de lungimea fişierului.

Lungime maximă (blocuri)	Lungime maximă (octeţi)	Accese indirecte	Accese la informaţie	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 octeţi care poate înregistra 1024 adrese, iar în inod se înregistrează direct adresele primelor 12 blocuri. In aceste condiţii, tabelul de mai sus se transformă în:

Lungime maximă (blocuri)	Lungime maximă (octeţi)	Accese indirecte	Accese la informaţie	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 fişier actual, indiferent de mărimea lui, poate fi reprezentat printr-o astfel de schemă.

4.2.4 Accesul proceselor la fişiere

Unix priveşte conceptul de fişier într-un sens ceva mai larg decât o fac alte sisteme de operare. Aşa cum am mai arătat mai sus, există opt tipuri de fişiere:

- 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 fişierelor, următoarele patru tipuri de comunicaţii 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 fişiere sunt, în ultimă instanţă:

- *Zona fişierelor pe disc*, pentru fişierele normale, directori, legături hard şi simbolice, FIFO şi socket din familia Unix.
- *Perifericul respectiv* pentru perifericele caracter şi bloc.
- *Zone rezervate de nucleu în memoria internă*, pentru pipe, memorie partajată, cozi de mesaje şi semafoare.
- *Interfaţa de comunicaţie prin reţea*, pentru socket din familia Internet.

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

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

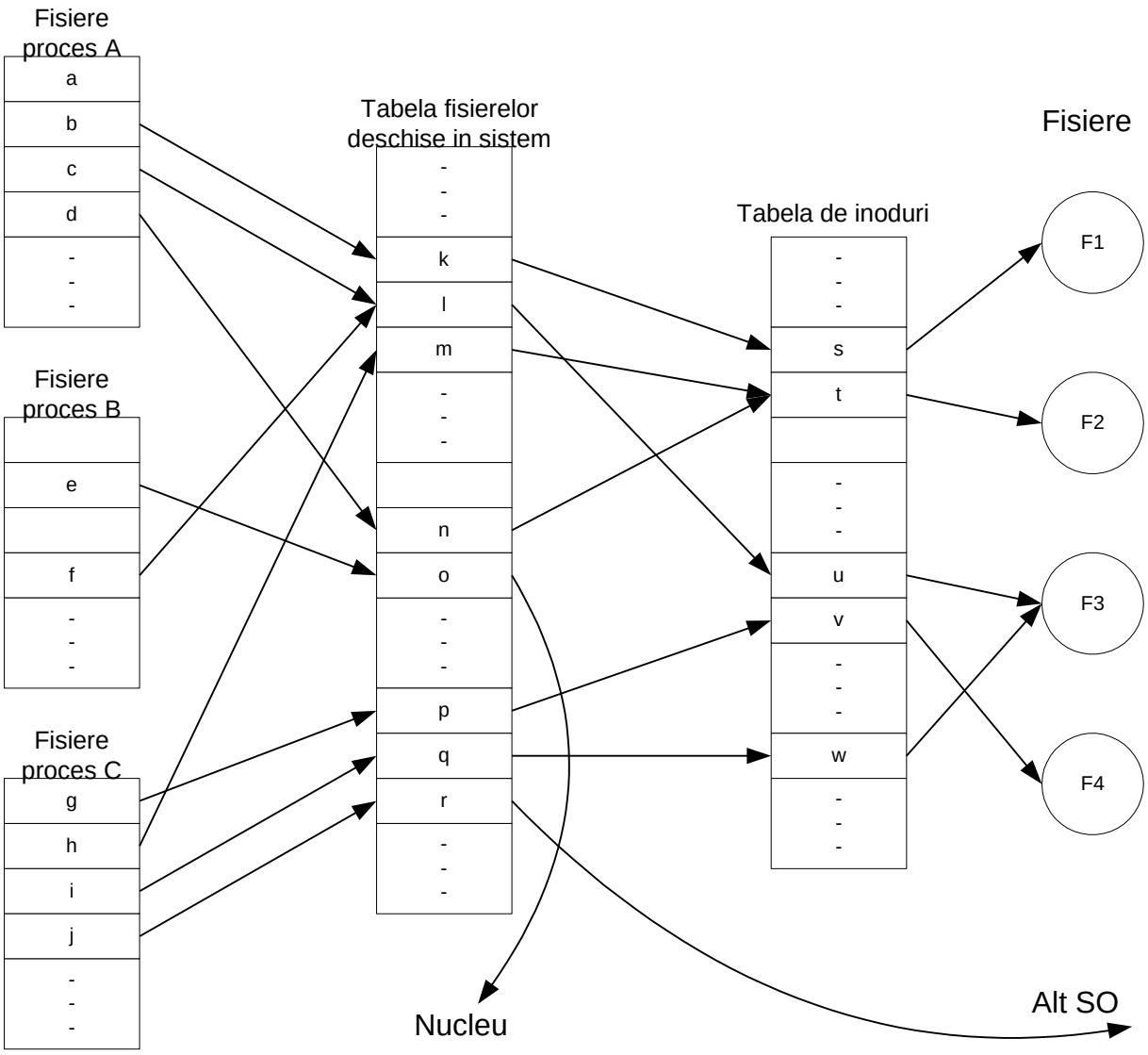

Figura 4.7 Corespondenţa Unix între procese şi fişiere

Nivelul proces. Fiecare proces îşi întreţine o tabelă proprie în care înregistrează toate fişierele lui deschise. In fig. 4.7 am notat cu **a-j** câteva intrări de pe acest nivel.

Nivelul sistem întreţine o tabelă unică cu toate fişierele deschise de către toate procesele din sistem. In 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. In fig. 4.7 am notat prin **s-w** câteva astfel de intrări.

Nivelul fişier este reprezentat de blocurile disc ce aparţin fişierului. In fig. 4.7 am notat **F1-F4** astfel de fişiere.

Tabela de fişiere 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** şi **g** din procesul **C**);
- intrarea 1 este rezervată ieşirii standard (vezi intrările **b, e, h** din fig. 4.7);
- intrarea 2 este rezervată fişierului standard de erori (unde sistemul afişează mesajele de eroare pentru proces - vezi intrările **c, i** din fig. 4.7).

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

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

In Unix este posibil ca mai multe procese să deschidă, în acelaşi timp, un acelaşi fişier - *multiacces la fişiere*. Acest lucru este posibil prin faptul că două intrări de fişiere din două procese pot puncta spre aceeaşi intrare din tabela sistem. In fig. 4.7 intrarea **c** de la procesul **A** şi intrarea **f** de la procesul **B** folosesc în comun acelaşi fişier, 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. Aşa 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** şi **n-t**. Această corespondenţă este posibilă în cazul în care cel puţin una dintre intrările **m** sau **n** se referă la o *legătură simbolică*.

Legăturile din **o** şi 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 reţea cu un alt sistem de operare.

In fine, în marea majoritate a cazurilor există o corespondenţă biunivocă între inoduri şi fişierele corespunzătoare. In fig. 4.7 avem corespondenţele **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 acelaşi fişier. In fig. 4.7 avem corespondenţele **u-F3** şi **w-F3**, cel puţin una dintre **u** şi **w** fiind o legătură hard.

4.3 Apeluri sistem pentru lucrul cu fişiere

4.3.1 Operaţii I/O

Există două posibilităţi de efectuare a operaţiilor 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. In caz de eşec `open` întoarce valoarea -1 şi poziţionează corespunzător variabila `errno`. In 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 sfîrşitul 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.

In 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`. In 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 neintreruptibile 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 maximul dintre `noct` şi ceea ce permite suportul fişierului. Astfel, dacă în fişier există doar `n < noct` 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 < noct` octeţi disponibili, atunci se depun din memorie în fişier doar `n` octeţi şi `write` va întoarce valoarea `n`. In 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 operaţia de citire sau de scriere nu se poate termina - a apărut o eroare în timpul transferului - funcţia întoarce valoarea -1 şi se poziţionează corespunzător variablia `errno`.

Ca exemplu de folosire a apelurilor sistem `open, close, read` şi `write` vom scrie un program care face acelaşi lucru ca şi comanda shell `cp`, adică copiază conţinutul unui fişier într-altul. Cele două fişiere se vor da ca parametrii în linia de comandă. Dacă al doilea fişier (fişierul destinaţie) nu există, el se va creea, iar dacă există deja, el se va suprascrie. 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 fişier 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 fişier în modul write-only. Dacă el nu
       * există, se va creea, sau dacă există va fi trunchiat. 0755
       * specifică drepturile de acces ale fişierului nou creat
       * (citire+scriere+executie pentru proprietar, citire+executie
       * pentru grup şi alţii).
       */
     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 fişierul fd_sursa bucăţi de maxim 100 octeţi şi le
     //scriem în fişierul 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 fişierul
         p = buf;
         for( ; n > 0 ; ) { // Poate nu se poat scrie odata n octeti
             i = write(fd_dest, p, n);
             if (i == n) break;
             p += i;
             n -= i;
         }
     }
     //închidem cele două fişiere
     close(fd_sursa);
     close(fd_dest);
}
```

Figura 4.8 Copierea unui fişier cu apelurile sistem read şi write

4.3.5 Apelul sistem lseek

Funcţia sistem `lseek` facilitează accesul direct la orice octet din fişier. Evident, pentru aceasta trebuie ca suportul fişierului să fie unul adresabil. Prototipul acestei funcţii sistem este:

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

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

- de la începutul fişierului, dacă are valoarea SEEK_SET (valoarea 0)
- de la poziţia curentă dacă are valoarea SEEK_CUR (valoarea 1)
- de la sfârşitul fişierului dacă are valoarea SEEK_END (valoarea 2)

4.3.6 Apelurile sistem dup şi dup2

Aceste două funcţii sistem permit ca un acelaşi fişier să fie accesibil prin doi descriptori diferiţi. Presupunem că `descrvechi` este o intrare în tabela de fişiere deschise a unui proces, care punctează, aşa cum am văzut în fig. 4.7 din 4.2.4, spre o intrare în tabela de fişiere deschise a sistemului. In urma apelului sistem, prin `dup` sau `dup2` se ocupă o nouă intrare `descrnou` din tabela de fişiere a procesului, care va puncta spre acelaşi fişier în tabela de fişiere deschise a sistemului. Această duplicare are loc în următoarele condiţii:

- `descrvechi` şi `descrnou` referă acelaşi fişier fizic
- ambele păstrează modul de acces la fişier stabilit la deschidere
- ambii descriptori partajează acelaşi pointer curent în fişier

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 fişiere a procesului.

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

Ambele apeluri întorc noul descriptor (`descrnou`) care duplică accesul la fişierul reperat prin `descrvechi`. In caz de eşec, ambele întorc -1 şi poziţionează corespunzător variabila `errno`.

De exemplu, dacă se doreşte ca dintr-un program C mesajele de eroare să fie trecute într-un fişier 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 esenţială a programului este `descrnou = dup2(descrvechi, 2);` Prin aceasta, se închide automat fişierul cu descriptorul 2 care referă fişierul standard `stderr` şi noul fişier standard de erori va fi fişierul ERORI. Dacă deschiderea sau `dup2` nu pot fi executate, atunci pe terminal va apare un mesaj în dublu exemplar: trimis prin `stderr` şi prin `stdout`. In caz de succes, pe terminal va apare o linie trimisă prin `stdout` iar în fişierul ERORI linia trimisă prin `stderr`.

4.3.7 Apelul sistem fcntl

Această funcţie sistem furnizează sau schimbă proprietăţi ale unui fişier deschis indicat printr-un descriptor. Această funcţie 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 fişierul deschis asupra căruia se doreşte aplicarea controlului.

Parametrul `cmd` este o constantă care specifică operaţia dorită şi în funcţie de ea mai urmează sau nu încă un argument. Vom prezenta doar o parte dintre posibilităţile oferite de `fcntl`, urmând ca în secţiunile următoare să revenim cu prezentarea de noi facilităţi:

- `fcntl(fd, F_DUPFD, arg)` are aproximativ acelaşi efect ca şi `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 fişierului faţă de cum au fost fixate prin `open` sau printr-un `fcntl` precedent. Parametrul `arg` are aceleaşi valoari ca şi parametrul `flag` de la `open`: una sau mai multe constante folosite pentru a specifica modul de deschidere a fişierului.
- `fcntl(fd, F_GETFL)` întoarce o configuraţie de biţi reprezentând reuniunea valorilor curente ale flagurilor de tratare a fişierului, aşa cum au fost ele fixate prin `open` sau printr-un `fcntl( ... F_SETFL ...)`.

Ce-a de a treia formă va fi tratată în secţiunea 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 specificări 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 şterg o legătură soft se şterge doar legătura, nu şi fişierul original, pe când dacă şterg o legătură hard, se şterge fişierul original.

- `chmod` atribuie drepturile de acces `drepturi` la fişierul specificat prin `nume`. Pe sistemele BSD şi Unix System V release 4 mai este prezent şi apelul `fchmod` care face acelaşi lucru ca şi `chmod`, doar că fişierul este specificat printr-un descriptor de fişier deschis, nu prin nume.
- `stat` depune la adresa `stare` informaţii privind fişierul specificat prin `nume` (inod-ul fişierului, drepturile de acces, id-ul proprietarului, id-ul grupului, lungimea în octeţi, numărul de blocuri ale fişierului, data ultimului acces la fişier, etc. – vezi exemplul de mai jos). Sistemele BSD şi Unix System V release 4 au şi apelul `fstat` care face acelaşi lucru ca şi `stat`, doar că fişerul este specificat printr-un descriptor de fişier deschis, nu prin nume.
- `mknod` creează un fişier simplu sau un fişier special desemnat prin `nume`. Parametrul `mod` specifică printr-o combinaţie de constante simbolice legate prin simbolul '|' atât drepturile de acces la fişierul nou creat, cât şi tipul fişierului care poate fi unul dintre următoarele:
  - S_IFREG (fişier normal)
  - S_IFCHR (fişier special de tip caracter)
  - S_IFBLK (fişier 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 fişierului este S_IFCHR sau S_IFBLK, atunci `dev` conţine numărul minor şi numărul major al fişierului special nou creat; altfel, acest parametru se ignoră.
- `chown` schimbă proprietarul şi grupul din care face parte proprietarul unui fişier specificat prin `nume`. Noul proprietar al fişierului va fi cel indicat de parametrul `proprietar`, iar noul grup al fişierului va fi cel indicat de parametrul `grup`. Pe sistemele BSD şi Unix System V release 4 mai este prezent şi apelul `fchown` care face acelaşi lucru ca şi `chown`, doar că fişierul este specificat printr-un descriptor de fişier deschis, nu prin nume.
- `access` verifică dacă procesul curent are dreptul specificat de `permisiuni` relativ la fişierul specificat prin `nume`. `permisiuni` va conţine una sau mai multe valori legate prin '|' dintre următoarele: R_OK - citire, W_OK - scriere, X_OK – execuţie şi F_OK – existenţă fişier. Verificarea se face cu UID-ul şi GID-ul reale ale procesului, nu cele efective (vezi capitolul 5).
- `rename` redenumeşte fişierul 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 fişierul specificat prin `nume` la exact `lung` octeţi.

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

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

4.4.3 Un exemplu: obţinerea tipului de fişier prin apelul sistem stat

In această secţiune vom da un exemplu de utilizare a lui `stat`. Exemplul se referă la afişarea tipului de fişier pentru fişierele 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 fişier recunoscut de către sistem. Tipurile de fişiere le-am prezentat într-o secţiune 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

In acest subcapitol, vom prezenta câteva apeluri sistem care pot fi utile programatorului Unix şi care nu au mai fost prezentate până aici. Inainte de prezentarea fiecărui prototip vom scrie şi directiva `#include` necesară la începutul sursei programului C. In 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` octeţi, numele maşinii (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 şi 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` octeţi începând de la deplasamentul `offset` din fişierul (sau alt obiect) indicat de descriptorul de fişier `fd` în memorie, preferabil la adresa `start`. In general, `start` are valoarea 0 şi adresa unde are loc maparea este returnată de `mmap`. Apelul sistem `munmap` şterge maparea de la adresa `start`, pe lungime `lung`. Parametrul `prot` specifică protecţia de memorie dorită şi 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, opţiuni de mapare şi dacă modificări ale conţinutului memoriei unde a avut loc maparea sunt vizibile numai procesului curent sau şi altor procese. Cele mai importante valori ale sale sunt: MAP_FIXED, MAP_SHARED, MAP_PRIVATE.

4.5.6 Fsync şi 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. Diferenţa dintre `fsync` şi `fdatasync` este că `fdatasync` nu salvează şi date de control despre fişier ca timpul ultimului acces la fişier, ci doar conţinutul 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 aceleaşi sensuri ca şi câmpurile afişare de comanda shell `uname`.

4.5.8 Select şi 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` şi `exceptfds` şi raportează (returnează) câte dintre ele şi-au modificat starea. Descriptorii din `readfds` vor fi monitorizaţi pentru a vedea dacă au apărut caractere noi disponibile pentru citire, cei din `writefds` pentru a vedea dacă o nouă operaţie de scriere nu se va bloca, iar cei din `exceptfds` vor fi monitorizaţi pentru excepţii. Parametrul `n` trebuie să aibă ca valoare cel mai mare număr de descriptor din cele trei seturi plus 1.

Standardul POSIX specifică şi 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-ieşire sincrone.

4.6 Blocarea fişierelor

4.6.1 Un (contra)exemplu

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

- Citeşte acest număr din `secv`.
- Măreşte numărul cu o unitate.
- Rescrie numărul rezultat în `secv`.

Programul `lockfile.c` din fig. 4.11 realizează aceste acţiuni. Funcţiile `my_lock(fd)` şi `my_unlock(fd)` au sarcina de a asigura accesul exclusiv la fişierul `secv` pe durata acţiunii. Intr-o primă variantă, comentăm apelurile celor două funcţii.

```
#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`

Inaintea rulării programului, se va crea cu un editor de texte fişierul `secv` în care se va introduce o singură linie conţinând numărul 1. Se compilează programul şi să presupunem că fişierul 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. In mod normal, valoarea numărului de secvenţă ar trebui să fie 20000! De ce nu se întâmplă aşa? Deoarece prin comentarea apelului funcţiilor `my_lock` şi `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 şi o vor scrie. In 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 şi obligatorie.

Blocare *conciliantă* (*advisory*) este atunci când sistemul ştie care fişiere au fost blocate şi de către cine, iar procesele cooperează, prin funcţii de tip `my_lock` şi `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 şi citire dacă fişierul este blocat sau nu. Pentru a permite blocarea obligatorie, trebuie invalidat dreptul de execuţie al grupului şi 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 octeţi consecutivi din fişier.

Mecanismele de blocare a fişierelor sub Unix specificate de standardul POSIX (şi 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 condiţiilor 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 şi 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 şi a scriitorilor.

4.6.3 Blocarea conciliantă prin fcntl

Nucleele Unix oferă mai multe apeluri sistem destinate blocării. De exemplu, sub Linux şi FreeBSD există o funcţie sistem `flock` pentru blocare. De asemenea, biblioteca C de la GNU oferă funcţiile `flock` şi `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 fişierului supus blocării.

Parametrul `lock` este un pointer la o structură de tip `flock`, prin care sunt indicaţi parametrii operaţiei de blocare. Această structură ce conţine cel puţin 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 octeţi */
   pid_t l_pid;    /* PID-ul procesului care blocheaza blocajul
                      curent */
 - - -
};
```

Câmpul `l_type` indică tipul operaţiei de blocare:

- `F_WRLCK` indică faptul că este vorba de o blocare exclusivă - blocare la scriere. Un singur proces poate efectua scriere în porţiunea rezervată, iar toate celelalte procese solicitante sunt în aşteptare.
- `F_RDLCK` indică faptul că este vorba de o blocare partajată, procesele ce solicită porţiunea rezervată pentru citire îşi pot efectua operaţiile, în timp ce procesele scriitori stau în aşteptare.
- `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 porţiunii rezervate. Valorile lui sunt aceleaşi cu cele de la funcţia `lseek`:

- `SEEK_SET` indică reperarea porţiunii rezervate faţă de începutul fişierului.
- `SEEK_CUR` indică reperarea porţiunii rezervate faţă de poziţia curentă a pointerului fişierului.
- `SEEK_END` indică reperarea porţiunii rezervate faţă de sfârşitul fişierului.

Câmpul `l_start` indică începutul porţiunii rezervate, numărat în octeţi faţă de locul indicat prin `l_whence`. Acest câmp poate avea atât valori pozitive, cât şi valori negative. In consecinţă, începutul zonei se specifică prin combinaţia parametrilor `l_whence` şi `l_start`.

Câmpul `l_len` indică lungimea în octeţi a zonei rezervate, un număr nenegativ. Dacă valoarea este strict pozitivă atunci sunt rezervaţi pentru blocare un număr de `l_len` octeţi de la începutul zonei rezervate. Dacă `l_len` are valoarea zero, atunci sunt rezervaţi toţi octeţii de la începutul zonei până la sfârşitul fişierului.

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

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

- `F_SETLKW` indică faptul că procesul cooperează la blocarea fişierului. Dacă fişierul este blocat exclusiv (la scriere) de către un alt proces, atunci procesul curent intră în aşteptare până la deblocarea fişierului. Acţiunea de blocare pe care o execută, după eventuala ieşire din starea de aşteptare, 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 aşteptare, ci întoarce -1 şi setează `errno` la EACCES sau EAGAIN.
- `F_GETLK` întoarce la adresa `lock` starea de blocare în care se află fişierul.

In fig. 4.12 sunt prezentate sursele funcţiilor `my_lock` şi `my_unlock` în care se foloseşte blocarea şi 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 bază
  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 funcţiile my_lock şi my_unlock folosind fcntl

4.6.4 Blocare prin lockf şi flock

Prototipurile celor două funcţii sunt:

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

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

`actiune` pentru `lockf` este una dintre următoarele:

- `F_ULOCK` deblocarea unei regiuni blocate
- `F_LOCK` blocarea unei regiuni
- `F_TLOCK` testarea şi 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 şi fişierul este deja blocat, atunci programul aşteaptă la apelul `flock` până când poate realiza blocarea cerută. Legarea acestui atribut, cu *sau* (|), de una dintre acţiunile anterioare, face ca în această situaţie să nu se mai aştepte.

`fd` este descriptorul de fişier.

`lungime` spune câţi octeţi vor fi blocaţi în fişier începând cu poziţia curentă. Dacă `lungime`=0 atunci se blochează întregul fişier.

In caz de erori, funcţia întoarce -1 şi poziţionează variabila globală sistem `errno`.
Reamintim că dintre cele două funcţii, numai `flock` reprezintă o funcţie sistem implementată de nucleu (pe care o apelăm prin intermediul funcţiei POSIX cu acelaşi nume implementată de biblioteca C de la GNU – menţionăm din nou că funcţia `flock` a bibliotecii C de la GNU este o funcţie *wrapper* pentru funcţia sistem `flock` implementată de nucleu), apelul funcţiei `lockf` traducându-se de fapt într-un apel sistem `fcntl`.

Folosirea lui `F_TEST` este relativ nesigură, din cauză că secvenţa:

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

nu este echivalentă cu secvenţa:

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

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

In fig. 4.13 sunt prezentate sursele `my_lock` şi `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_ULOCK, 0L);
}//my_unlock
```

Figura 4.13 funcţiile my_lock şi my_unlock folosind lockf