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).

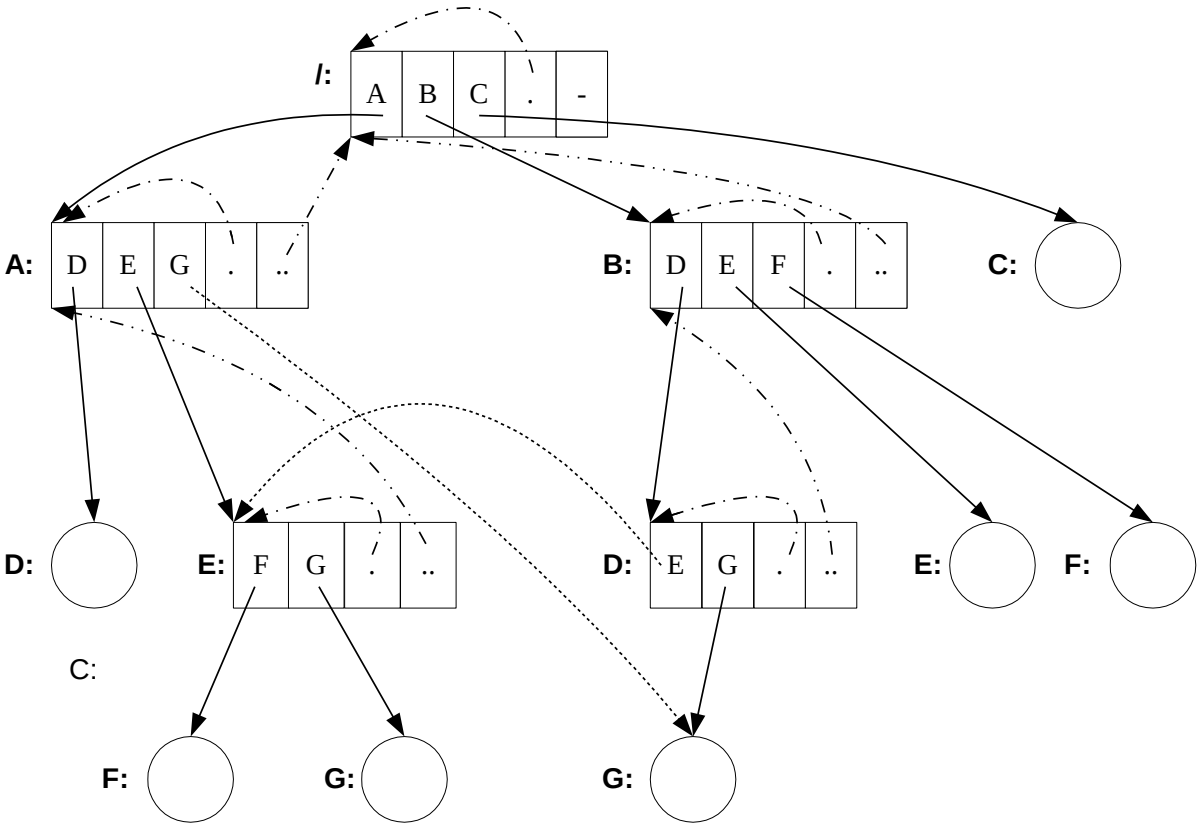

Figura 4.1 O structură arborescentă cu legături

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 :(

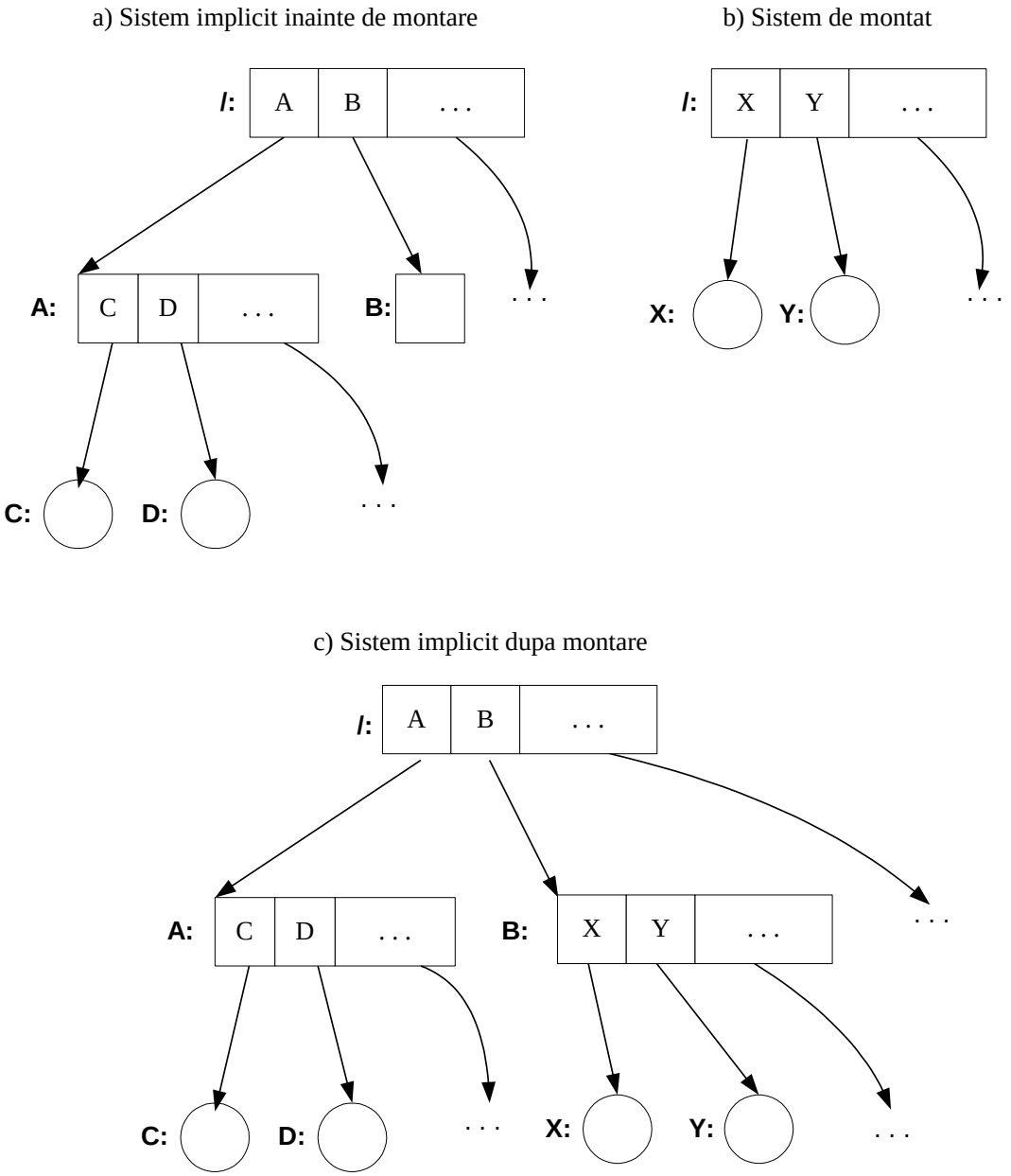

Figura 4.2 Operaţia de montare

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`, <u>în mod implicit</u> acţiunile pe care programul `P` le poate efectua asupra fişierului `F` <u>sunt cele permise de drepturile pe care `U` le are asupra lui `F`.</u> 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 cvasigeneralã 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`, <u>în mod setuid / setgid</u> acţiunile pe care programul `P` le poate efectua asupra fişierului `F` <u>sunt cele permise de drepturile proprietarului programului `P` şi ale grupului din care face parte acesta.</u>

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.

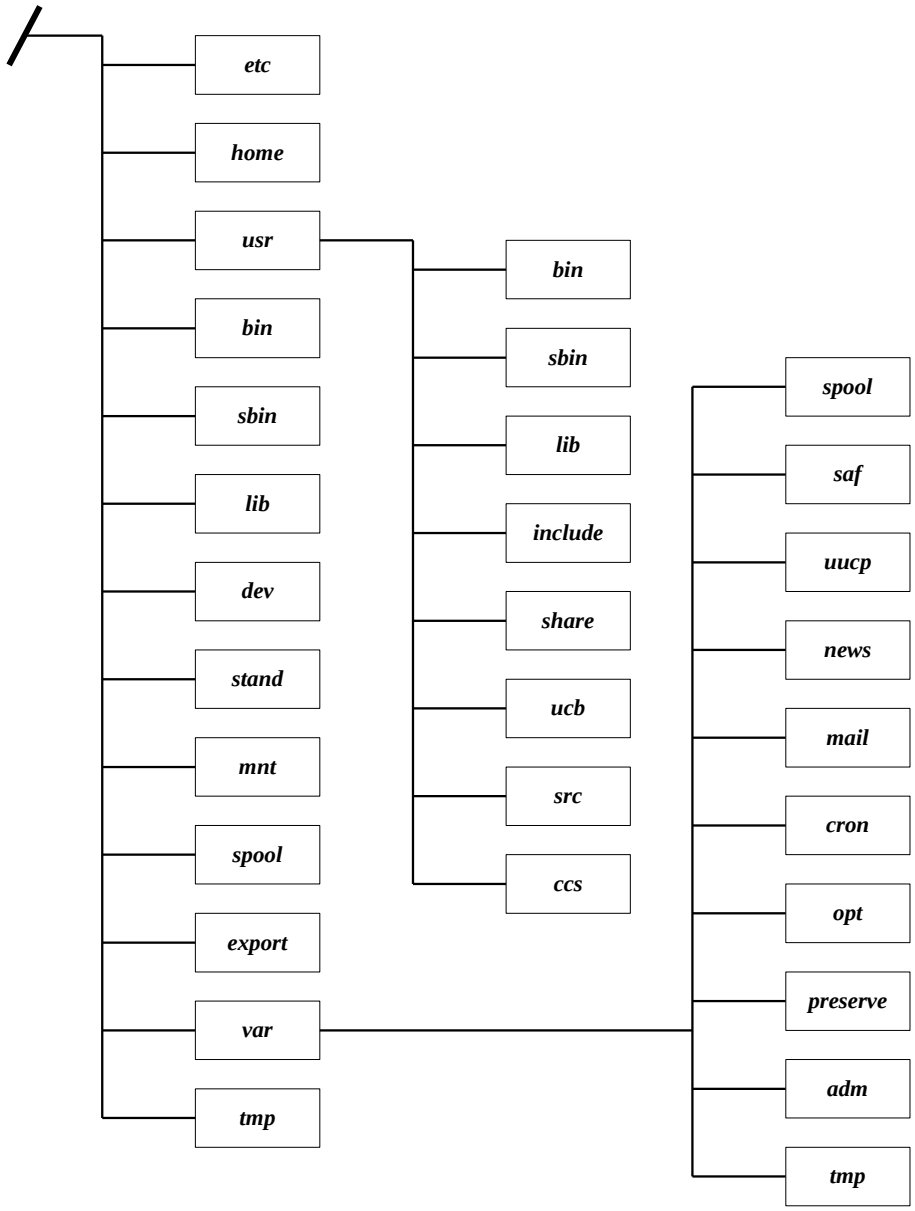

Figura 4.3 Structura superioară a unui sistem de fişiere Unix

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*.

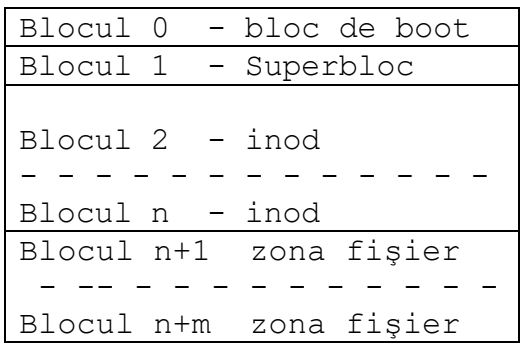

Figura 4.4 Structura unui disc Unix

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.

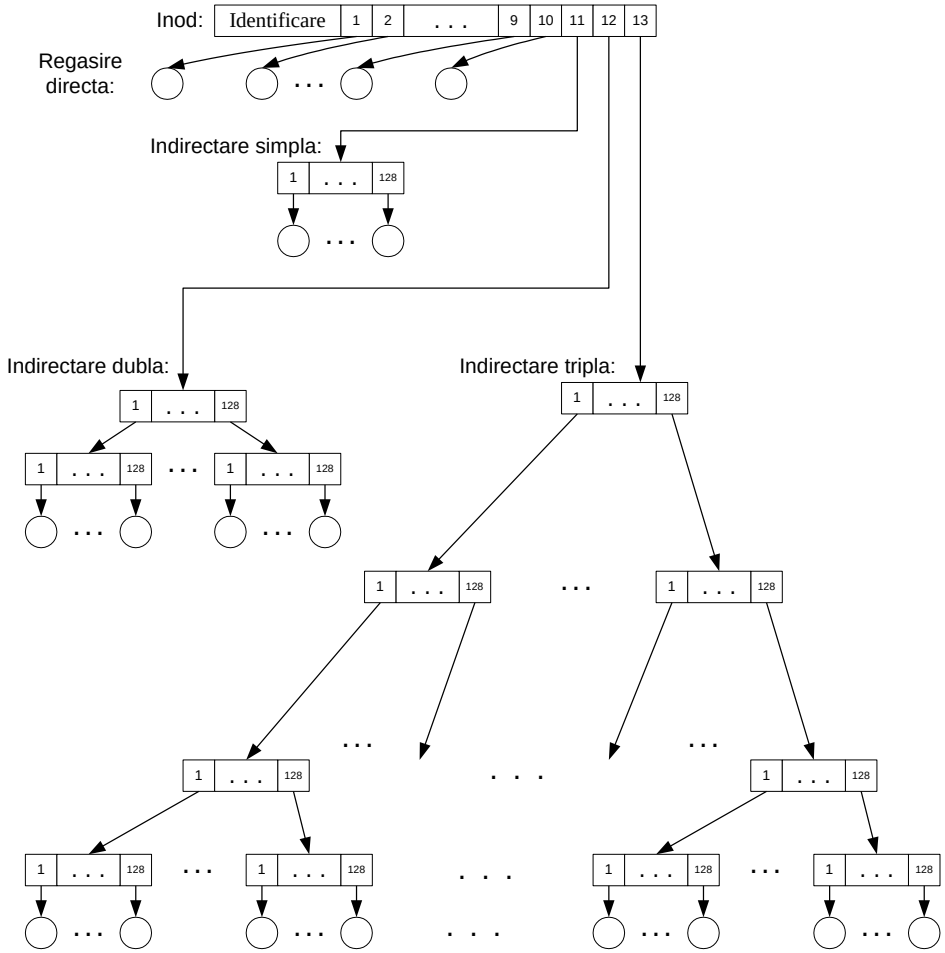

Figura 4.6 Structura unui inod şi accesul la blocurile unui fişier

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.

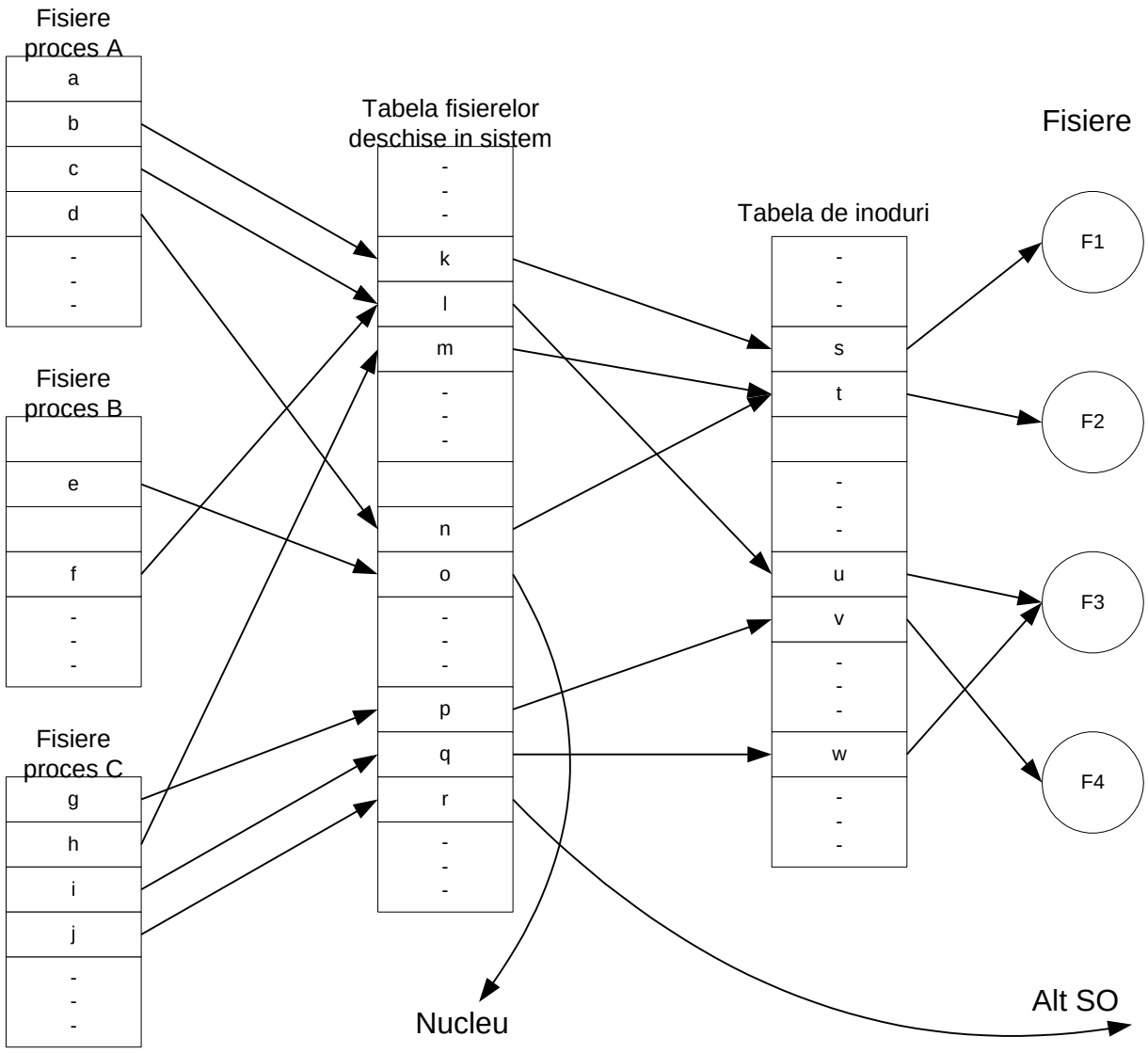

Figura 4.7 Corespondenţa Unix între procese şi fişiere

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

8 Sisteme de operare – prezentare generală

8.1 Tipuri de sisteme de operare (SO); clasificări

In prezent există un mare număr de SO în funcţiune şi numărul acestora este în continuă creştere. Nu există un criteriu unitar de comparare a acestora. Noi vom da mai multe clasificări, în funcţie de următoarele criterii:

- după gradul de partajabilitate a resurselor;
- după tipurile de interacţiuni permise;
- după organizarea internă a programelor ce compun SO.

8.1.1 Clasificare după gradul de partajabilitate a resurselor

După acest criteriu, distingem trei categorii de sisteme de operare [10]:

- Sisteme monouser şi monotasking;
- Sisteme monouser şi multitasking;
- Sisteme multiuser.

Sisteme de operare monouser sunt acelea care permit, la un moment dat, unui singur utilizator să folosească sistemul. Un sistem este monotasking dacă admite la un moment dat execuţia unui singur program.

In forma cea mai simplă, un sistem monouser şi monotasking execută un singur program la un moment dat şi acesta rămâne activ din momentul lansării lui şi până la terminarea lui completă. Un astfel de sistem este de exemplu cel care deserveşte un telefon mobil sau un terminal PDA (Personal Digital Assistant).

Un sistem monouser şi multitasking este acela în care un singur utilizator este conectat la sistem, dar el poate să îşi lanseze simultan mai multe procese în lucru. Tipice în acest sens sunt SO din familia Windows 9X, ca şi cele din familia NT care nu sunt servere. Tot în această categorie putem îngloba sistemele Linux instalate pe calculatoare neconectate în reţea. Pentru sistemele de acest fel care au memoria internă relativ mică, se aplică aşa-numita tehnică swapping, adică evacuarea temporară a unui program, din memoria internă pe discul magnetic. In timpul evacuării, în memoria internă este încărcat un alt program, care la rândul lui este executat parţial şi este apoi evacuat şi el. Cât timp un program este în memorie, el are acces la toate resursele sistemului. Programele sunt astfel executate pe porţiuni, momentele de execuţie alternând cu cele de evacuare. După unii autori ([4]), SO de acest tip se află la graniţa dintre sistemele monoutilizator şi cele multiutilizator.

Sisteme de operare multiutilizator permit accesul simultan al mai multor utilizatori la sistem. Ele trebuie să aibă în vedere partajarea procesorului, a memoriei, a timpului, a perifericelor şi a altor tipuri de resurse, între utilizatorii activi la un moment dat. Vom reveni cu detalii asupra metodelor de partajare a procesorului, timpului şi a memoriei. La astfel de sisteme sunt necesare tehnici mai sofisticate, de gestiune şi protecţie a utilizatorilor. La un moment dat aici există mai multe procese active care se execută concurent sub controlul SO.

8.1.2 Clasificare după tipurile de interacţiuni permise

In funcţie de resursele pe care le are un sistem şi în funcţie de destinaţia acestuia, este necesară stabilirea unor strategii de interacţiune cu utilizatorul. Conform acestui criteriu de diferenţiere, distingem [34]:

- Sisteme seriale (care prelucrează loturi de lucrări);
- Sisteme cu timp partajat (time – sharing);
- Calculatoare personale şi staţii de lucru (workstations);
- Sisteme autonome (embedded systems);
- Sisteme portabile, destinate comunicaţiilor (telefoane mobile, PDA-uri etc.);
- Sisteme conectate în reţea.

Sisteme seriale. In cadrul acestor sisteme lucrările se execută pe loturi pregătite în prealabil. Practic, din momentul predării unei lucrări la ghişeul centrului de calcul şi până la eliberarea ei, utilizatorul nu poate interveni spre a influenţa execuţia programului său. SO afectate acestor sisteme pot lucra atât monoutilizator, cât şi multiutilizator. Din punct de vedere istoric sunt printre primele sisteme. In prezent, în această categorie intră supercalculatoarele, precum şi o serie de sisteme medii – mari cum ar fi AS-400. Sistemul de operare trebuie să gestioneze loturi de lucrări pe care să le pregătească spre a fi introduse în sistem şi eventual să pregătească livrarea rezultatelor (listare etc.). In general gradul de concurenţă la acest tip de sisteme este relativ redus. In [10] sunt date mai multe detalii despre acest tip de sisteme.

Sistemele cu timp partajat trebuie să suporte, într-o manieră interactivă, mai mulţi utilizatori. Fiecare utilizator este în contact nemijlocit cu programul său. In funcţie de unele rezultate intermediare, el poate decide modul de continuare a activităţii programului său. SO trebuie să gestioneze printre altele şi terminalele de teletransmisie la capătul cărora se află utilizatori care-i pot transmite diverse comenzi. Sistemul trebuie să dispună de mecanisme mai sofisticate decât cele seriale pentru partajarea procesorului, a memoriei şi a timpului. De regulă, acest tip de sisteme practică partajarea timpului, astfel. Se fixează o cuantă de timp. Pe durata cuantei, un singur proces are acces la procesor (şi la celelalte resurse aferente). După epuizarea cuantei, procesul este suspendat, pus la sfârşitul cozii de procese şi un alt proces ocupă procesorul pentru următoarea cuantă ş.a.m.d. Dimensiunea cuantei şi strategiile de comutare sunt astfel alese încât să se reducă timpii de aşteptare şi să se realizeze o servire rezonabilă a proceselor. Tipice acestui tip de sisteme sunt sistemele Unix, dar şi Windows din familia NT / server.

Calculatoare personale şi staţii de lucru. La acest tip de sisteme, toate resursele maşinii sunt dedicate utilizatorului care este conectat. Aşa că nu se pune problema partajării resurselor între utilizatori, ci doar a datelor între procesele pe care userul le lansează simultan. Practic, sarcina principală a SO este aceea de a oferi userului o interfaţă prietenoasă de exploatare a sistemului.

Sistemele autonome sunt sisteme dedicate unui anumit proces industrial, cum ar fi: funcţionarea unui robot, coordonarea activităţilor dintr-un satelit geostaţionar, supravegherea unui baraj de apă, a unei staţii de radiolocaţie etc. Fiind conectate la diverse procese tehnologice, aceste sisteme trebuie să fie capabile să deservească în timp prestabilit fiecare serviciu care i se cere. Dacă sistemul nu este capabil să dea un răspuns, este posibilă oprirea procesului supravegheat. Astfel de sisteme sunt în prezent în plină dezvoltare, mai ales din cauza dezvoltării rapide a tehnologiilor multimedia.

Sisteme portabile, destinate comunicaţiilor. Acestea formează cea mai nouă categorie de sisteme. Exemplele tipice sunt telefoane mobile şi PDA-urile. Sunt maşini portabile, de dimensiuni mici şi au facilităţi puternice de comunicare. Facilitatea lor principală este conexiunea wireless printr-una dintre tehnologiile existente: radio, infraroşu, Bluetooth etc. Evident, sistemul de operare aferent trebuie să fie capabil să gestioneze eficient aceste comunicaţii. Din cauza formatului mic, memoria internă, dar mai ales memoria pe suport extern este foarte limitată. Sistemul de operare trebiue să facă faţă acestei penurii de resurse. Securizarea accesului la un astfel de sistem este esenţială. Fiind echipamente mici pot fi uşor pierdute / furate, iar această securizare trebuie să le facă de nefolosit de către persoanele neautorizate.

Sisteme conectate în reţea. Practic, astăzi marea majoritate a sistemelor nu mai sunt independente, ci sunt conectate la reţele de calculatoare, inclusiv la reţeaua publica Internet. Din această cauză, una dintre sarcinile fundamentale ale sistemelor de operare din această categorie trebuie să fie gestiunea accesului la reţea. De asemenea, accesul din exterior la resursele locale trebuie să fie bine protejat, spre a interzice accesele neautorizate.

In legătură cu acest criteriu de clasificare, trebuie arătat că el este relativ. Astfel, există sisteme seriale care permit interacţiunea cu programele, sistemele interactive au facilităţi puternice de lucru serial etc. Mai mult, acelaşi SO permite în acelaşi timp să lucreze, spre exemplu, în timp real, dar în acelaşi timp să servească, evident cu o prioritate mai mică, şi alte programe de tip serial sau interactiv, în regim de multiprogramare clasică.

*8.1.3 **Clasificare după organizarea internă a programelor ce compun SO.***

Să vedem cum arată un SO văzut din "interior". Vom vedea patru tipuri de structuri, în evoluţia lor istorică.

- Sisteme monolitice;
- Sisteme cu nucleu de interfaţă hardware;
- Sisteme cu structură stratificată;
- Sisteme organizate ca maşini virtuale

Sisteme monolitice. Un astfel de SO este o colecţie de proceduri, fiecare dintre acestea putând fi apelată după necesităţi. Execuţia unei astfel de proceduri nu poate fi întreruptă, ea trebuie să-şi execute complet sarcina pentru care a fost apelată. Un program utilizator rulat sub un sistem monolitic se comportă ca o procedură apelată de SO. La rândul lui, programul poate apela în maniera apelurilor sistem (vezi 4.5 pentru Unix) diverse rutine din SO. Singura structură posibilă aici este legată de cele două moduri de lucru: nucleu şi user (vezi pentru Unix 4.5 şi pentru Windows 7.1.3). Esenţial pentru aceste tipuri de sisteme este legătura de tip apel-revenire, fără posibilitatea de întrerupere a fluxului normal al execuţiei. In prezent, aceste tipuri de SO sunt pe cale de dispariţie.

Sisteme cu nucleu de interfaţă hardware. Un astfel de sistem concentrează sarcinile vitale de nivelul cel mai apropiat de hardware într-o colecţie de rutine care formează nucleul SO. In acesta sunt înglobate inclusiv rutinele de întrerupere. Componentele nucleului se pot executa concurent. Toate acţiunile utilizatorului asupra echipamentului hard trec prin acest nucleu. Unele SO mai plasează între nucleu şi utilizator încă o interfaţă. Exemple de nuclee ale SO pot fi considerate componentele BIOS ale maşinilor PC deşi fac parte din hardware.

Sisteme cu structură stratificată este o generalizare a organizării cu nucleu. Un astfel de sistem este construit nivel după nivel, componentele fiecărui nivel folosind toate serviciile oferite de nivelul inferior. In prezent aceste sisteme sunt cele mai răspândite. "Părinţii" lor sunt SO THE (Dijkstra, 1968) şi SO MULTICS. Ultimul dintre ele oferă o organizare mai interesantă, şi unică în felul ei, aceea a inelelor de protecţie. Pentru detalii se pot consulta [48], [49]. In secţiunea următoare vom ilustra structura stratificată a unui SO ipotetic.

Sisteme organizate ca maşini virtuale. Echipamentul hard al acestor tipuri de sisteme serveşte, în regim de multiprogramare, eventual în time-sharing, un număr de procese. Fiecare proces dispune, în mod exclusiv, de o serie de resurse, cea mai importantă fiind memoria. Fiecare dintre procesele deservite este un sistem de operare, care are la dispoziţie toate resursele alocate procesului respectiv de către echipamentul hard. In acest mod, pe acelaşi echipament hard se poate lucra simultan sub mai multe SO. Primele sisteme de acest tip au fost VM/370 (Virtual Machines for IBM-370), care coordonează mai multe SO conversaţionale monoutilizator de tip CMS (Conversational Monitor System). Fiecare utilizator lucrează sub propriul CMS, pentru el fiind transparent faptul că lucrează sub CMS singur, pe un calculator mic, sau că este legat, împreună cu alţii la acelaşi echipament hard. Sistemele actuale Windows oferă maşini virtuale (ferestre) pentru lucru sub DOS. Implementările Linux, FreeBSD şi Solaris actuale oferă maşina virtuală Wine care emulează platforme Windows. Pentru detalii, se pot consulta [4], [49].

8.2 Structura si funcţiile unui sistem de operare

8.2.1 Stările unui proces şi fazele unui program

8.2.1.1 Stările unui proces

In secţiunile precedente am folosit destul de des termenul de proces, privit intuitiv ca un program în execuţie. Am văzut în capitolele precedente noţiunea de proces sub Unix şi sub Windows. In fig. 5.18 din 5.2.3 sunt prezentate stările unui proces Unix. Există [62] şi stări în care se poate afla un proces Windows.

Sintetizând stările proceselor sub diverse sisteme de operare, putem defini stările unui proces într- un sistem de operare. Aceste stări sunt:

- HOLD – proces pregătit pentru intrarea în sistem;
- READY – procesul este în memorie şi este gata de a fi servit de (un) procesor;
- RUN – un procesor execută efectiv instrucţiuni (maşină) ale procesului;
- WAIT – procesul aşteaptă terminarea unei operaţii de intrare / ieşire;
- SWAP – imaginea procesului este evacuată temporar pe disc;
- FINISH – procesul s-a terminat, trebuie livrate doar rezultatele.

HOLD este starea unui proces la un sistem serial în care procesului îi sunt citite datele de lansare din cadrul lotului din care face parte, se face o analiză preliminară a corectitudinii lor şi apoi procesul este depus într-o coadă pe disc (numită HOLD) spre a fi preluat spre prelucrare. De obicei, aceste sisteme au o componentă specializată pentru astfel de prelucrări – SPOOL-ing de intrare (Simultaneous Peripheral Operations OnLine – vezi [10]).

FINISH este, de asemenea, stare a unui proces la un sistem serial. Procesul se execută şi rezultatele lui sunt plasate pe disc într-o coadă FINISH. După terminare, acestea vor fi listate,

fie pe o imprimantă locală, fie pe una aflată la distanţă. Componenta din SO care face preia din coada FINISH şi listează se numeşte SPOOL-ing de ieşire (vezi [10]).

READY este starea în care un proces se află în memoria internă dar nu este servit de procesor. Este posibil ca el să fi fost servit parţial, dar pe moment să fie suspendat în defavoarea altui proces, urmând să i se continue execuţia mai târziu.

SWAP este starea în care un proces este evacuat temporar într-un spaţiu rezervat pe disc (SWAP) pentru a face temporar loc altui proces în memoria internă. Ulterior procesul va fi reîncărcat în memoria internă şi i se va continua execuţia.

WAIT este starea în care un proces a cerut executarea unei operaţii de intrare / ieşire. Cât timp procesul aşteaptă să se termine operaţia, cedă altor procese procesorul.

RUN este starea principală, aceea în care procesorul execută efectiv instrucţiuni maşină ale programului procesului. Intr-un sistem monoprocesor doar un singur proces se află în starea RUN. Dacă sistemul are n procesoare, atunci maximum n procese se vor afla în starea RUN.

Aceste stări nu se vor întâlni la toate tipurile de sisteme de operare. De exemplu, la un sistem monoutilizator (vezi 8.1)nu va fi starea SWAP, numai sistemele seriale (vezi [10]) au stările HOLD şi FINISH etc.

8.2.1.2 Fazele unui program

Fazele unui program reprezintă etapele prin care trece acesta din momentul în care s-a început proiectarea lui şi până când devine cod maşină executabil spre a fi integrat într-un proces. Aceste faze, bine cunoscute de către programatori, sunt:

1. Editarea textului sursă într-un limbaj de programare, executată de către programator cu ajutorul unui editor de texte.
2. Compilarea, executată de un compilator specializat în limbajul respectiv. Ca rezultat se obţine un fişier obiect.
3. Editarea de legături, fază în care se grupează mai multe module obiect rezultate din compilări şi se obţine un fişier executabil. Editarea se face cu un program specializat, numit editor de legături
4. Execuţia programului înseamnă mai întâi încărcarea fişierului executabil în memorie cu ajutorul unui program încărcător (loader) şi lansarea lui în execuţie.
5. Testarea şi depanarea programului, fază care alternează cu cele de mai sus până când proiectantul are certitudinea că programul lui este corect. Uneori această depanare poate fi asistată de către un program specializat numit depanator.

In secţiunile următoare vom reveni cu detalii, atât asupra stărilor unui proces cât şi asupra fazelor unui program.

Menirea unui SO, indiferent de tipul lui, este pe de o parte de a facilita accesul la sistem a unuia sau mai multor utilizatori, iar pe de altă parte de a asigura o exploatare eficientă a echipamentului de calcul. Vom încerca să arătăm mai în detaliu care sunt funcţiile posibile ale unui sistem, independent de tipul acestuia. Desigur, dependentă de tip va fi ponderea în cadrul sistemului a uneia sau alteia dintre funcţii. Din motive care se vor clarifica mai târziu, vom folosi în loc de termenul program termenul de proces sau task. Printr-un proces vom înţelege un calcul care poate fi efectuat concurent cu alte calcule (asupra acestei noţiuni vom reveni).

S-au proiectat şi se proiectează încă multe tipuri de SO. Fiecare dintre acestea trebuie să aibă ca obiective fundamentale:

- optimizarea utilizării resurselor;
- minimizarea efectului uman de programare şi exploatare;
- automatizarea operaţiilor manuale în toate etapele de pregătire şi exploatare a SC;
- creşterea eficienţei utilizării SC prin scăderea preţului de cost al prelucrării datelor.

Principalele funcţii ale SO sunt legate de acţiunile acestuia pentru tranziţia între diverse stări ale procesului. In fig. 8.1 [10], [47] sunt schematizate stările unui proces. In dreptul fiecărei tranziţii sunt trecute principalele acţiuni efectuate de către SO. Caracteristicile fiecăreia dintre cele şase stări posibile (HOLD, READY, RUN, SWAP, WAIT, FIHISH) le-am prezentat în secţiunea precedentă.

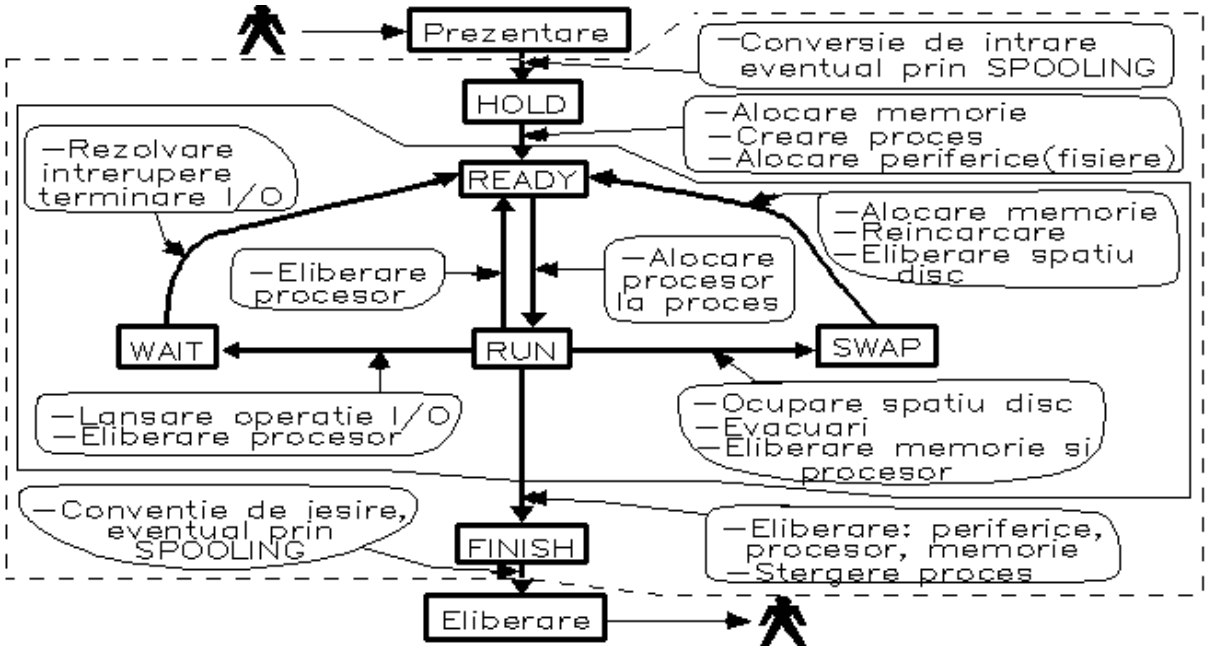

Figura 8.1 Stările unui proces şi acţiunile SO aferente

Acţiunile SO ilustrate în fig. 8.1 sunt executate de către componenta nucleu a sistemului de operare. La nivelul componentei user, o mare parte dintre acţiunile SO sunt focalizate mai ales în sprijinul utilizatorului pentru dezvoltarea de programe, adică de asistare a userului la trecerea programului prin diverse faze (vezi secţiunea precedentă 8.2.1.2).

Am exclus de aici, de fapt am amânat numai până la o secţiune viitoare, o funcţie primordială: punerea în lucru a SO. După cum vom vedea, în mod (aparent) paradoxal, această sarcină revine tot SO! La prima vedere pare a fi vorba de o antinomie, cum este posibil ca ceva (cineva) să-şi dea viaţă singur? Vom vedea.

8.2.2 Structura generală a unui sistem de operare

In această secţiune încercăm să oferim o imagine asupra structurii unui SO ipotetic. Un astfel de sistem nu va fi regăsit nicăieri în practică, însă structura fiecărui SO real va fi apropiată de a acestuia. Evident, în funcţie de tipul de SO real, unele dintre prezentele componente vor fi atrofiate sau vor lipsi cu desăvârşire, iar altele vor putea fi eventual mult mai dezvoltate decât

aici. Fără pretenţia de "gen-proxim" şi "diferenţă-specifică", încercăm să definim principalele noţiuni, structuri şi componente cu care operează orice SO.

Să trecem acum la structura propriu-zisă.

8.2.2.1 Structura unui SO

Schema generală a unui SO este ilustrată în fig. 8.2.

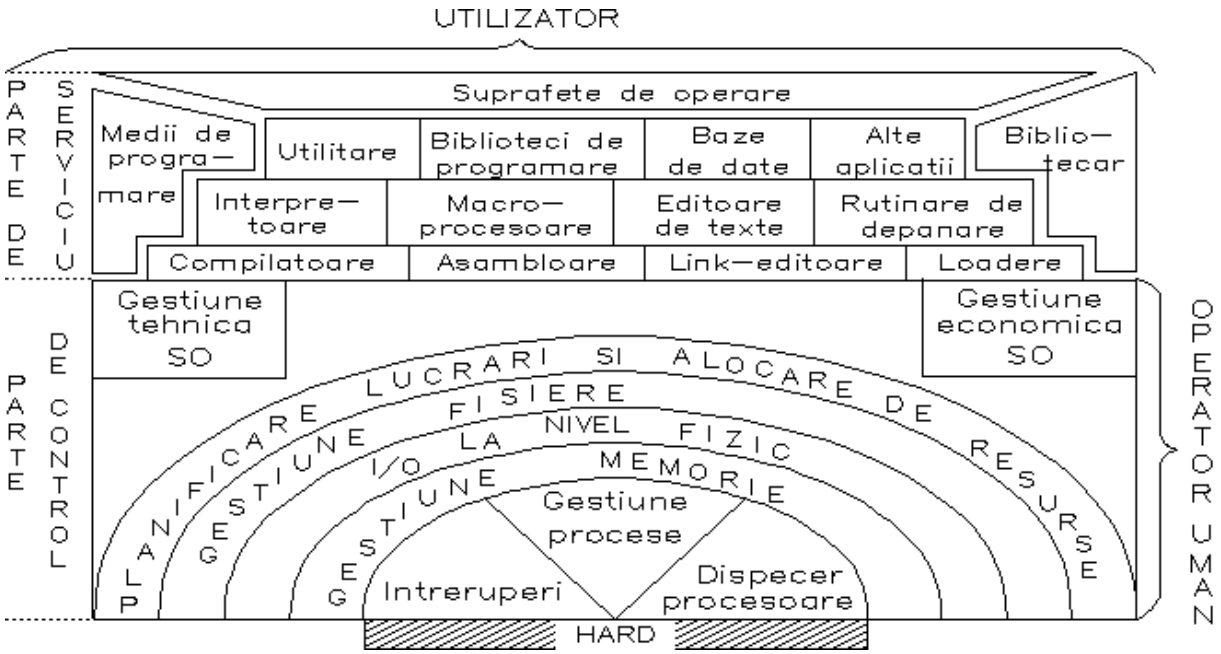

Figura 8.2 Structura generală a unui SO

Orice SO conţine două părţi mari: partea de control şi partea de servicii.

PARTEA DE CONTROL este în legătură directă cu operatorul uman şi indirectă cu utilizatorul. SO execută în mod master componentele acestei părţi. Principalul ei rol este de a realiza legătura cu SC, deci cu echipamentul hard.

PARTEA DE SERVICIU lucrează în mod slave, folosind facilităţile părţii de control. Este în legătură directă cu utilizatorul şi indirectă cu operatorul uman. Sarcina de bază a ei este de a ţine legătura cu utilizatorul, facilitându-i acestuia accesul la resursele SC şi SO.

Ideea schemei din fig. 8.2 este preluată din [10]. In continuare vom descrie succint rolul fiecărei componente.

8.2.2.2 Partea de control

INTRERUPERI. Un ansamblu de rutine, fiecare dintre ele fiind activată la apariţia unui anumit semnal fizic de întrerupere. Corespunzător, SO efectuează acţiunile aferente întreruperii respective. Semnificaţia conceptului de întrerupere este descrisă în fiecare manual de arhitectura calculatoarelor. Pentru întreruperile 8086, se poate consulta [52].

Funcţionarea concretă a componentei (sistemului) de întreruperi depinde foarte mult de particularităţile SC, deci nu se pretează prea bine la abordări cu caracter general. Pe de altă parte, detalii despre aceste componente interesează în primul rând pe programatorii şi inginerii de sistem, mai puţin pe utilizatorii obişnuiţi., cărora ne adresăm noi. Din aceste motive vom schiţa numai mecanismul de funcţionare a întreruperilor.
Fiecare tip de întrerupere are asociată o locaţie fixă de memorie. In această locaţie se află o adresă. Această adresă indică locul din memorie la care se află o secvenţă de instrucţiuni, numită handler, secvenţă care deserveşte întreruperea respectivă.

La apariţia semnalului de întrerupere, după ce instrucţiunea maşină în curs s-a executat, se execută, în această ordine, următoarele activităţi:

- se salvează într-o zonă de memorie starea programului în curs de desfăşurare;
- se activează handlerul asociat întreruperii respective;
- handlerul execută acţiunile necesare servirii întreruperii;
- se restaurează starea programului care a fost întrerupt.

GESTIUNE PROCESE. Creează procese şi rezolvă problemele privind cooperarea şi concurenţa acestora. La terminarea unui proces, acesta este şters din sistem. Asupra conceptului de proces vom mai reveni.

DISPECER PROCESOARE. La sistemele multiprocesor, repartizează sarcinile de calcul solicitate de procese, procesoarelor care sunt libere şi care pot executa sarcinile cerute.

GESTIUNEA MEMORIEI. Alocă necesarul de memorie internă solicitat de procese şi asigură protecţia memoriei interprocese. Componenta este parţial realizată prin hard şi parţial este realizată soft, de către o componentă a SO. Vom reveni şi asupra problemelor legate de gestiunea memoriei.

I/O LA NIVEL FIZIC. Asigură efectuarea operaţiilor elementare de I/O cu toate tipurile de periferice din sistem realizând, acolo unde este posibil, desfăşurarea simultană a uneia sau mai multor operaţii I/O, cu activitatea procesorului central. Această componentă este, de asemenea, puternic dependentă hard, în special de multitudinea de echipamente periferice conectate la sistem. Aşa după cum am văzut, există nişte rutine de interfaţă numite drivere, care realizează compatibilizarea convenţiilor de I/O între SC şi echipamentele periferice. Driverele printre altele, fac parte din această componentă. Vom reveni într-o secţiune viitoare cu detalii.

GESTIUNEA FISIERELOR. O colecţie de module prin intermediul cărora se asigură: deschiderea, închiderea şi accesul utilizatorului la fişierele rezidente pe diferite suporturi de informaţii. Componentă de bază a SO, este cel mai des invocată de către utilizatori şi de către operator. Vom reveni şi asupra ei într-un capitol special.

PLANIFICAREA LUCRARILOR SI ALOCAREA RESURSELOR. Resursele unui SC se împart în: resurse fizice (procesoare, memorie internă, periferice etc.) şi resurse logice (proceduri comune, tabele ale sistemului etc.). Fiecare proces solicită la un moment dat anumite resurse. Prin această componentă a SO se asigură planificarea proceselor astfel, încât ele să poată obţine de fiecare dată resursele necesare, în mod individual sau partajate cu alte procese. Elemente ale planificării se răsfrâng, după cum vom vedea mai târziu, asupra gestiunii proceselor, gestiunii memoriei, intrărilor si ieşirilor etc. Atunci când vom discuta despre procese, memorie şi I/O la nivel fizic vom aborda şi problematica planificărilor şi a alocărilor de resurse.

GESTIUNEA TEHNICA a SO. Tine evidenţa erorilor hard apărute la echipamentele SC. La cerere, furnizează informaţii statistice asupra gradului de utilizare a componentelor SC.

GESTIUNEA ECONOMICA a SO. Tine evidenţa utilizatorilor care folosesc sistemul, lucrările executate de către aceş- tia şi resursele consumate de aceste lucrări (timpi de rulare, memorie internă ocupată, echipamente şi suporţi folosiţi etc). Periodic (zilnic, lunar, anual) editează liste cuprinzând lucrările rulate şi facturi de plată către beneficiari.

8.2.2.3 Partea de servicii

COMPILATOARELE sunt programe care traduc texte sursă scrise în limbaje de nivel înalt (C, C++, Java, FORTRAN, COBOL, Pascal, Ada etc.) în limbaj maşină. La această oră există o mare varietate de tehnici de compilare, având ca nucleu teoria limbajelor formale. Există de asemenea sisteme automate de elaborare a diverselor componente ale compilatorului. Rezultatul unei compilări este un modul obiect memorat într-un fişier obiect. Detalii privind compilatoarele se pot obţine consultând [33].

ASAMBLORUL este un program care traduce texte scrise în limbajul de asamblare propriu SC într-un modul obiect. Din punct de vedere al SO problemele legate de asamblor sunt incluse în problemele compilatoarelor. De exemplu, programarea în limbaj de asamblare 8086 este descrisă în [52].

LINK-EDITORUL sau EDITORUL DE LEGATURI. Grupează unul sau mai multe module obiect, împreună cu subprograme de serviciu din diverse biblioteci, într-un program executabil. De cele mai multe ori această componentă oferă posibilitatea segmentării programelor mari, adică posibilitatea ca două componente diferite a programului să poată ocupa în execuţie, la momente diferite de timp, aceeaşi zonă de memorie. Rezultatul editării de legături este un fişier executabil. In 5.1 este descris formatul executabil de sub Unix, iar în [52] fişierele executabile COM şi EXE de pe platformele Microsoft.

LOADER (PROGRAM DE INCARCARE) este un program al SO care are trei sarcini:

- citirea unui program executabil de pe un anumit suport;
- încărcarea acestuia în memorie;
- lansarea lui în execuţie.

Dacă programul executabil este segmentat, atunci loaderul încarcă la început numai segmentul rădăcină al programului, după care, la cerere, în timpul execuţiei, încarcă de fiecare dată segmentul solicitat în memorie. In [52] este descris mecanismul de încărcare de sub MS-DOS.

INTERPRETOR este un program care execută pas cu pas instrucţiunile descrise într-un anumit limbaj. Tehnica interpretării este aplicată la diferite limbaje de programare, cum ar fi Java, BASIC, LISP, PROLOG etc. Pentru SO, cea mai importantă categorie de interpretoare sunt "interpretoarele de comenzi". In 2.1.1 am descris interpretoarele de comenzi Shell de sub Unix, iar în [10] [52] interpretorul de comenzi COMMAND.COM de pe platformele Microsoft.

MACROPROCESOR (PREPROCESOR) este un program cu ajutorul căruia se transformă o machetă de program, pe baza unor parametri, într-un program sursă compilabil. Spre exemplu, fie macheta:

```
MACRO     ADUNA     &1, &2, &3
LOAD      &1
ADD       &2
STORE     &3
ENDMACRO
```

Dacă într-un program (dintr-un limbaj de asamblare) se scrie:

```
ADUNA  A, B, C
```

atunci prin macroprocesoare se generează:

```
LOAD      A
ADD       B
STORE     C
```

Practic, macroprocesarea este tehnica prin care se realizează o scriere mai compactă a programelor şi o posibilă parametrizare a acestora pentru realizarea comodă a unor implementări înrudite.

Astfel de macroprocesoare există la toate SO moderne. Remarcăm în mod deosebit interpretorul de comenzi "shell" de sub Unix, descris în 2.2. Acesta dispune, printre altele, de cele mai puternice facilităţi de macroprocesare dintre toate interpretoarele de comenzi cunoscute.

Unele limbaje au prevăzute în standardele lor specificaţii de macroprocesare. In acest sens amintim preprocesorul "C" care tratează în această manieră liniile care încep cu "#" şi construcţiile "typedef", precum şi facilităţile generice ale limbajului ADA .

EDITORUL DE TEXTE este un program interactiv destinat introducerii şi corectării textelor sursă sau a unor texte destinate tipăririi. Orice SO interactiv dispune cel puţin de câte un astfel de editor de texte. Până vom ajunge la capitolul dedicat editoarelor de texte, să enumerăm câteva editoare "celebre" unele prin vechimea lor, iar altele prin performanţele lor:

- ed şi vi sub Unix;
- Notepad şi EDIT sub Windows;
- xedit şi emacs de sub mediul X WINDOWS.

RUTINELE DE DEPANARE (DEPANATOARELE) asistă execuţia unui program utilizator şi-l ajută să depisteze în program cauzele erorilor apărute în timpul execuţiei lui. Intreaga execuţie a programului utilizator se face sub controlul depanatorului. La prima apariţie a unui eveniment deosebit în execuţie (depăşire de indici, adresă inexistentă, cod de operaţie necunoscut etc.) depanatorul primeşte controlul. De asemenea, el poate primi controlul în locuri marcate în prealabil de către utilizator. Un astfel de loc poartă numele de breakpoint. La primirea controlului, depanatorul poate afişa valorile unor variabile sau locaţii de memorie, poate reda, total sau parţial istoria de calcul, poate modifica valorile unor variabile sau locaţii de memorie etc. Unele depanatoare mai performante permit execuţia reversibilă, concept tratat teoretic dar mai puţin aplicat practic. La această oră sunt cunoscute două tipuri de depanatoare:

- depanatoare maşină;
- depanatoare simbolice.

Depanatoarele maşină operează cu elemente cum ar fi: adrese fizice, configuraţii de biţi, locaţii de memorie etc. Fiecare SO dispune de câte un astfel de instrument de depanare. Pentru

exemplificări se pot consulta documentaţiile MS_DOS pentru DEBUG, precum şi documentaţia Unix despre depanatoarele adb şi gdb [47].

Depanatoarele simbolice operează cu elemente ale textelor sursă ale programelor. Intâlnim aici termeni ca linie sursă, nume de variabilă, nume de procedură, stivă de apel a procedurilor etc. Evident, depanatoarele simbolice sunt mult mai tentante pentru utilizator, şi este bine că este aşa. De obicei, un astfel de depanator însoţeşte implementarea unui limbaj de programare de nivel înalt. Primul şi cel mai răspândit limbaj la care au fost ataşate depanatoare simbolice este limbajul Pascal. Amintim aici depanatorul Pascal de pe lângă implementarea TURBO Pascal, un depanator simbolic complet interactiv, cu o grafică destul de bună pentru momentul implementării lui şi cu facilităţi puternice. Firma BORLAND, un adevărat vârf de lance în acest domeniu, a realizat printre altele produsul TURBO DEBUGER, care recunoaşte în vederea depanării toate limbajele pentru care s-au realizat medii TURBO: Pascal, "C", limbajul de asamblare TASM, Basic, PROLOG etc. Produsul lucrează atât simbolic, cât şi la nivel maşină.

BIBLIOTECARUL este un program cu ajutorul căruia utilizatorul poate comanda păstrarea programelor proprii în fişiere speciale de tip bibliotecă, poate copia programe dintr-o bibliotecă în alta, şterge, inserează şi modifică programe în aceste biblioteci. Exemple de programe biblioteci sunt LBR sub MS-DOS, ar (archive) sub Unix etc.

MEDIILE DE PROGRAMARE sunt componente complexe destinate în principal activităţii de dezvoltare de programe. O astfel de componentă înglobează, relativ la un limbaj, cel puţin următoarele subcomponente:

- editor de texte;
- compilator interactiv;
- compilator serial (clasic);
- editor de legături sau un mecanism echivalent acestuia destinat creerii de programe executabile;
- interpretor pentru execuţia rezultatului compilării;
- depanator, de regulă simbolic;
- o componentă de legătură cu mediul exterior, de regulă cu SO sub care se lucrează.

De regulă, filozofia de manevrare a mediului are un caracter unitar, este simplă şi puternică. Facilităţile grafice ale acestor medii sunt destul de mult utilizate, făcând deosebit de agreabil lucrul cu ele.

SUPRAFETELE DE OPERARE au apărut tot la calculatoarele IBM-PC şi a celor compatibile cu ele. Ele au rolul principal de a "îmbrăca" sistemul de operare (Windows sau Unix), pentru a-i face mai uşoară operarea. Iniţial au fost destinate utilizatorilor neinformaticieni. La această oră, nici profesioniştii nu se mai pot "dezlipi" de o suprafaţă de operare cu care s-a obişnuit să lucreze. De regulă, suprafeţele de operare înlocuiesc limbajul de control (de comandă) al SO respectiv. Utilizatorul, în loc să tasteze o comandă a SO respectiv, poartă un dialog cu suprafaţa de operare, iar la finele dialogului suprafaţa generează sau execută comanda respectivă. Ponderea dialogului o deţine suprafaţa însăşi, utilizatorul răspunde de regulă prin apăsarea uneia sau maximum a două taste, sau unul sau două "clicuri" cu un mouse. Suprafaţa deţine şi un mecanism deosebit de puternic de HELP, conducând utilizatorul din aproape în aproape spre scopul dorit. Deşi se adresează în primul rând neinformaticienilor, ele sunt din ce în ce mai mult folosite şi de către specialişti.

O remarcă specială trebuie făcută pentru WINDOWS, care deţine un mecanism propriu de dezvoltare a unor aplicaţii.

Din raţiuni impuse de standarde, sub Unix se foloseşte aproape peste tot suprafaţa X-WINDOWS [43].

Din scurta prezentare a componentelor principale ale unui SO rezultă caracterul modular al SO şi organizarea lui ierarhică. Modularitatea este termenul prin care se descrie partiţionarea unui program mare în module mai mici, cu sarcini bine precizate. Caracterul ierarhic este reflectat în faptul că orice componentă acţionează sub controlul componentelor interioare acesteia, apelând primitivele oferite de acestea. Aceste facilităţi asigură o implementare mai uşoară a SO, o înţelegere mai facilă a SO de către diverse categorii de utilizatori, o întreţinere şi depanare mai uşoară a sistemului şi creează premisele simulării lui pe alt sistem.

8.3 Incărcarea (lansarea în execuţie) a unui sistem de operare

Una dintre sarcinile de bază ale unui SO este aceea de a se putea autoîncărca de pe disc în memoria internă şi de a se autolansa în execuţie. Această acţiune se desfăşoară la fiecare punere sub tensiune a SC, precum şi ori de câte ori utilizatorul (operatorul uman) crede de cuviinţă să reîncarce SO. Acţiunea este cunoscută sub numele de încărcare a SO sau lansare în execuţie a SO.

In cele ce urmează dorim să punem în evidenţă mecanismul de lansare. O vom face însă pentru un SO ipotetic, fără să ne legăm de un anume sistem de operare concret. simplificând la maximum expunerea.

Pentru lansare se utilizează un mecanism combinat hard şi soft, numit bootstrap. Pentru a putea înţelege mai bine modul lui de funcţionare, să considerăm un exemplu.

Să notăm cu M[0], M[1], M[2], . . . locaţiile de memorie ale unui calculator ipotetic. Presupunem că în fiecare locaţie M[i] încape un număr întreg. Presupunem, de asemenea că sistemul nostru dispune de o memorie ROM şi un disc (dischetă, CD, DVD, hard disc, partiţie disc etc.) numit disc de boot, de la care se citesc, secvenţial, începând din primul sector, numere întregi.

Prin read(x) vom înţelege că se citeşte următorul număr de la intrarea standard şi valoarea lui se depune în locaţia x.

Prin transferla(y) înţelegem că următoarea instrucţiune maşină de executat este cea din (al cărei cod se află în) locaţia y.

Mecanismul bootstrap intră în lucru la apăsarea butonului de pornire <START>. Ca prim efect, sistemul extrage din ROM un şir de octeţi reprezentând echivalentul în limbaj maşină al programului din fig. 8.3.

```
APASASTART:
  for (i = 0; i < 100 ; i++) {
    read(M[i]);
  }
  transferla(M[0]);
```

Figura 8.3 Programul APASASTART

Sirul de octeţi citit este depus în memorie la o adresă pe care o vom nota APASASTART. După depunerea la adresa respectivă, sistemul execută instrucţiunea:

transferla(APASASTART);

Cu aceasta, sistemul transferă controlul programului din fig. 8.3. Execuţia acestui program înseamnă că sistemul citeşte de pe discul de boot primele 100 de numere, le depune în primele 100 de locaţii de memorie, după care trece la executarea instrucţiunii al cărei cod se află în prima locaţie.

Este limpede că în 100 de locaţii nu se poate scrie decât un program simplu. Ori pentru lansarea în execuţie a unui SO, probabil că este necesară punerea în lucru a unui program mai sofisticat. Un astfel de program poate fi scris pe mai multe sectoare consecutive, folosind, de exemplu reprezentarea din fig. 8.4. Să presupunem că într-un sector se pot înregistra S numere întregi.

AM1	n1	n1 întregi de program sau date	S – n1 – 2 întregi zero
AM2	n2	n2 întregi de program sau date	S – n2 – 2 întregi zero
- - -	- - -	- - -	- - -
AMk	Nk	nk întregi de program sau date	S – nk – 2 întregi zero
ADL	-1		S – 2 întregi zero

Figura 8.4 Structura unui fişier ce conţine un program executabil

Deci fişierul executabil din fig. 8.4 poate avea un număr oarecare k de înregistrări a câte un sector, conţinând numere întregi. Semnificaţiile câmpurilor sunt:

- AMi adresa din memorie de unde începe introducerea porţiunii de cod de program sau date din înregistrarea i;
- ni este numărul efectiv de locaţii ocupat de porţiunea de program sau date a înregistrării i. Valoarea -1 indică ultimul sector al a programului.
- ADL se află în ultimul sector al programului şi este adresa de lansare în execuţie a programului reprezentat: un număr între AM1 şi AMk+nk-1.

Să considerăm acum programul din fig. 8.5, în care presupunem că începând de la locaţia M[r] există S locaţii libere.

```
LOADERABSOLUT:
  do {
    for ( i=0; i < S; i++) {
      read(M[r+i]);
    }
    A = M[r];
    n = M[r+1];
    if (n== -1)
      break;
    for ( i=0; i< n; i++) {
      M[A+i] = M[r+2+i]
    }
  while (FALSE);
LANSARE:
  transferla(A);
```

Figura 8.5 Incărcător (loader) de programe cu structura din fig. 8.4

Avem acum posibilitatea să citim de pe disc şi să lansăm în execuţie, cu programul din fig. 8.5, programe oricât de mari cu formatul din fig. 8.4. De fapt, în fig. 8.5 avem de-a face cu un exemplu simplu de program de încărcare (loader).

Cheia (saltul calitativ al) încărcării: In cele 100 de numere care vor fi citite în momentul startului cu programul APASASTART (din fig. 8.3), se trece codul maşină al programului LOADERABSOLUT (din fig. 8.5). Să vedem acum, pe etape, cum are loc încărcarea şi lansarea în execuţie a SO. In fig. 8.6 sunt ilustrate zonele de memorie folosite în faza de încărcare şi lansare în execuţie a SO.

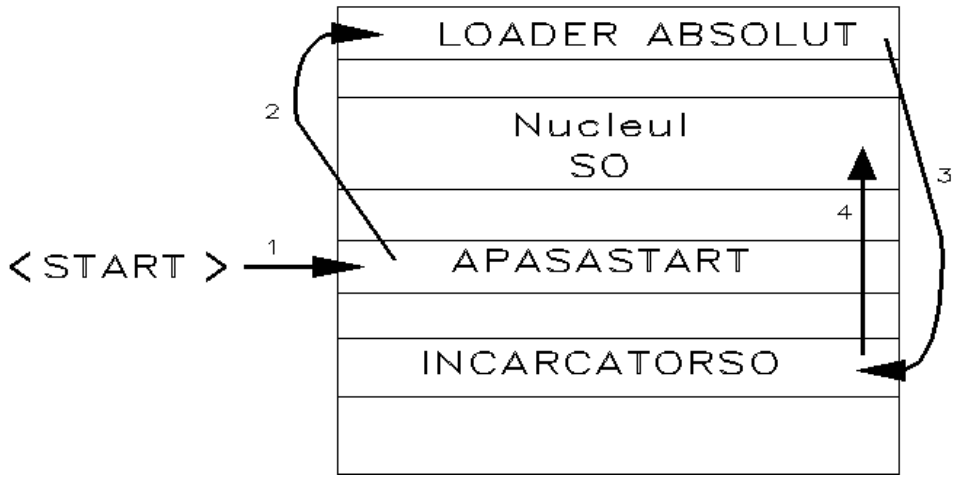

Figura 8.6 Incărcarea unui SO

Etapele încărcării sunt:

1) Se apasă <START>. Ca efect, programul APASASTART citeşte, în primele 100 de locaţii de memorie, codul maşină al programului LOADERABSOLUT.
2) După transferla(M[0]) intră în lucru programul LOADERABSOLUT.
3) Acesta citeşte la rândul lui un program, care de această dată poate fi oricât de sofisticat, cu formatul din fig. 8.4. Acest program îl vom numi INCARCATORSO.
4) După ce se execută şi ultima instrucţiune transferla(A) a programului LOADERABSOLUT, intră în lucru programul INCARCATORSO. Dat fiind faptul că formatul de reprezentare a codului lui maşină este cel din fig. 8.4, acesta poate fi practic oricât de mare. Aşa cum îi spune şi numele, acest program va încărca de pe disc programele şi rutinele (nucleului) sistemului de operare.

La încheierea încărcării SO, INCARCATORSO execută următoarele acţiuni:

- şterge din memorie codul programului APASASTART;
- şterge din memorie codul programului LOADERABSOLUT;
- eliberează spaţiul de memorie ocupat de el, cu excepţia unei scurte porţiuni, despre care vorbim mai jos;
- execută o instrucţiune transferla(ASSO), unde ASSO este adresa de start a sistemului de operare. Codul necesar acestei instrucţiuni este cel pe care nu-l poate, în mod firesc, elibera. Insă dacă acest cod este plasat într-o zonă folosită ca buffer, practic nu mai rămâne memorie ocupată.
- după aceasta, încărcătorul îşi încheie activitatea.

In sfârşit, toate SO moderne lansează la terminarea încărcării un fişier căruia o să-i spunem fişier de comenzi iniţiale. Acesta este un fişier de comenzi, care la rândul lui poate lansa alte fişiere de comenzi ş.a.m.d. text în care sunt trecute toate comenzile pe care administratorul le doreşte a fi executate la pornirea sistemului. Printre ultimele acţiuni desfăşurate la lansarea So amintim:

- montarea partiţiilor locale;
- efectuarea unor operaţii de întreţinere asupra sistemului de fişiere (filesystem cleanup);
- pornirea diferitelor servicii (de exemplu serviciile de printare);
- setare nume sistem şi iniţializare ceasuri în funcţie de zona geografică;
- configurează reţeaua;
- montează sisteme de fişiere aflate la distanţă (aşa-numitele sisteme de fişiere remote)
- permite conectarea utilizatorilor;

Proiectantul SO va avea grijă ca programul încărcător să fie depus într-o zonă de memorie care să nu fie ocupată de către programele SO pe care el le încarcă.

Faţă de modelul de încărcare exemplificat de noi mai sus, realitatea, conceptual, este foarte aproape. Primele calculatoare executau APASASTART citind o cartelă, pe care era înregistrat programul LOADERABSOLUT. Acesta realiza citirea programului INCARCATORSO de pe bandă sau de pe disc. Multe calculatoare foloseau în acest scop banda perforată (care uneori trebuia "ajutată" cu mâna să defileze prin cititor ☺. Imaginea "arhaică" a unei astfel de porniri este similară -cei în vârstă probabil că îşi aduc aminte- cu pornirea în anii '50 a celebrului tractor KD, sau a unei drujbe, pornire care se face trăgând de o sfoară. ☺)

In prezent suportul magnetic de încărcare este primul sector (sectorul zero) al unui disc magnetic. Programele care realizează încărcarea se vor deosebi de cele de mai sus, numai prin înlocuirea lui 100 cu 128, 256, 512 sau 1024, deci cu lungimea unui sector.

Unele SO pretind o modalitate prin care i se indică sistemului de pe ce disc să încarce programele LOADERABSOLUT şi INCARCATORSO: specificarea se face în memoria CMOS, la cheile pupitrului calculatorului etc. Alte sisteme stabilesc singure, în mod automat, de pe ce disc să facă încărcarea.

9 Teoria generală a sistemelor de operare

9.1 Procese

9.1.1 Conceptul de proces

Facilitatea de a a rula mai multe programe simultan în cadrul unui sistem de operare este considerată astăzi normală de către toţi utilizatorii. Rularea unui navigator web (Mozilla Firefox, Internet Explorer, Safari, Konqueror etc.) simultan cu rularea unui program pentru citirea poştei electronice (Mozilla Thunderbird, Outlook Express, Eudora, Pine etc.) este o practică de zi cu zi a majorităţii utilizatorilor. Din punctul de vedere al unui proces de operare toate aceste programe rulând pe un sistem de operare sunt considerate procese sau task-uri.

Formal, un *proces* sau *task,* este un calcul care poate fi executat concurent (în paralel) cu alte calcule. Este o abstractizare a activităţii procesorului care pur şi simplu execută instrucţiunile care i se transmit fără a face diferenţiere între procesele cărora le aparţin. Revenind la exemplul iniţial, un procesor nu "ştie" dacă instrucţiunea executată aparţine de Mozilla Firefox sau Outlook Express.

In terminologia actuală există o tendinţă de confuzie între noţiunea de proces şi cea de program. Deşi folosirea noţiunii de program pentru a ne referi la un proces nu creează în general confuzii, considerăm totuşi necesară clarificarea situaţiei. *Procesul are un caracter dinamic*, el precizează o secvenţă de activităţi în curs de execuţie, iar *programul are un caracter static*, el numai descrie această secvenţă de activităţi. Cu alte cuvinte procesul este un program în execuţie.

Existenţa unui proces este condiţionată de existenţa a trei factori:

1. *procedură* (un set de instrucţiuni) care trebuie executată;
2. un *procesor* care să poată executa aceste instrucţiuni;
3. un *mediu* (memorie, periferice) asupra căruia să acţioneze procesorul conform celor precizate în procedură.

Execuţia unui program (şi implict transformarea acestuia în proces) implică execuţia de către procesor a fiecărei instrucţiuni din programul (procedura) respectiv(ă). In mod inerent, aceste instrucţiuni vor avea efect asupra mediului în care procesul rulează, cum ar fi: alocarea de memorie, folosirea perifericelor, tipărirea unui mesaj pe terminal, consumul de timp etc.

Trebuie menţionat că în arhitecturile actuale monoprocesor (vezi [52]), mai multe procese care rulează concomitent sunt de fapt deservite alternativ de către procesor. El (procesorul) execută alternativ grupuri de instrucţiuni din fiecare program de-a lungul unei cuante de timp, după care comută la alt grup de la alt program. Dimensiunea cuantei de timp este astfel aleasă (şi aşa de mică) încât momentele de staţionare ale unui program nu sunt sesizabile de către utilizator. Astfel se creează impresia de simultaneitate în execuţie.

Paralelismul este efectiv doar în cadrul sistemelor multiprocesor. Practic, două procese pot fi executate simultant în sens propriu doar dacă rulează pe o maşină bi-procesor.

9.1.2 Concurenţa între procese

Existenţa simultană a mai multor procese într-un sistem de operare ridică probleme în ceea ce priveşte accesul la resursele sistemului. Prin resursă înţelegem orice este necesar unui proces pentru a-şi continua activitatea. Câteva resurse mai populare sunt:

- memoria (cereri de alocare de memorie)
- discul magnetic (acces la fişiere)
- terminalul (pentru interacţiunea cu utilizatorul)
- interfaţa de reţea (pentru interacţiunea cu alte maşini legate prin reţea)

9.1.2.1 Secţiune critică; resursă critică; excludere mutuală.

O problemă legată de accesul la resurse este asigurarea corectitudinii operaţiilor executate in regim de concurenţă. Pentru a vedea cum operaţii corecte pot da rezultate greşite în regim de concurenţă, vom analiza un exemplu clasic. Să considerăm un fişier care stochează numărul de locuri libere într-un sistem de vânzare de bilete. Vânzarea unui bilet va avea ca şi consecinţă decrementarea numarului de locuri libere. De asemenea, să considerăm că există două procese care efectuează simultan operaţii asupra acestui fişier. Fragmentul de cod necesar pentru decrementarea numărului de locuri este prezentat in fig. 9.1.

```
int n;
int fd = open(”locuri.db”, ”ORDWR”);
read(fd, &n, sizeof(int));
lseek(fd, 0, SEEK_SET);
n--;
write(fd, &n, sizeof(int));
close(f);
```

Figura 9.1 O secvenţă de decrementare

In situaţia în care codul din fig. 9.1 este rulat simultan de două sau mai multe procese, şi considerând faptul că procesorul execută alternativ grupuri de instrucţiuni din procese diferite, este posibil ca instrucţiunile de mai sus să fie executate în procese separate în ordinea din figura 9.2.

Procesul A	**n = (în A)**	**Procesul B**	**n = (în B)**
`read(fd,&n,sizeof(int))`	10		
	10	`read(fd,&n,sizeof(int))`	10
`n--`	9		10
`lseek(fd,0,SEEK_SET)`	9		10
`write(fd,&n,sizeof(int))`	9		10
		`n--`	9
		`lseek(fd,0,SEEK_SET))`	9
		`write(fd,&n,sizeof(int))`	9

Figura 9.2 Programul din fig. 9.1 rulat "defavorabil" în două procese

In mod evident, ambele procese citesc *aceeaşi valoare* din fişier, o decrementează şi apoi o scriu la loc. Ca urmare, în final, în loc ca numărul de locuri să scadă cu două poziţii, va scădea doar cu una şi ca urmare operaţiile executate asupra fişierului deşi sunt greşite. In viaţa de zi cu zi o asemenea eroare ar conduce la vânzarea aceluiaşi loc în sală de două ori (o “bucurie” cu care mulţi dintre citiori au avut proabil de-a face).

Trebuie remarcat că eroarea poate să apară doar la o execuţie concurentă, altfel secvenţa de cod fiind perfect corectă. Eroarea apare din lipsa de sincronizare între accesele proceselor la fişier. Procesul B, în loc să opereze asupra numărului de locuri decrementat de procesul A, reuşeşte să citească numarul de locuri prea devreme şi va opera asupra *aceluiaşi număr de locuri* ca şi procesul A. Pentru a obţine rezultate corecte, secvenţa de execuţie dorită este cea din fig. 9.3. Adică secvenţa de cod care decrementează numărul de locuri să fie executată la un moment dat doar de un signur proces.

Procesul A	**n = (în A)**	**Procesul B**	**n = (în B)**
`read(fd,&n,sizeof(int))`	10		
`n--`	9		
`lseek(fd,0,SEEK_SET)`	9		
`write(fd,&n,sizeof(int))`	9		
		`read(fd,&n,sizeof(int))`	9
		`n--`	8
		`lseek(fd,0,SEEK_SET)`	8
		`write(fd,&n,sizeof(int))`	8

Figura 9.3 Programul din fig. 9.1 rulat ca secţiune critică

Acelaşi gen de situaţie se creează şi în momentul în care procesele scriu date pe disc. Dacă sistemul de operare ar servi aceste cereri în stilul din figura 9.2 două procese ar putea primi acelaşi bloc pe disc pentru a scrie în el date aparţinând la fişiere diferite. Este uşor de imaginat ce s-ar întâmpla dacă într-un script shell editat cu un editor de text ar apărea părţi din conţinutul unui fişier `".o"` generat de `gcc` la compilare …

Revenind la problema noastră, vom spune că porţiunea de program din fig. 9.1 începând cu `read` şi terminând cu `write` este o *secţiune critică*, deoarece nu este permis ca ea să fie executată simultan de către două procese. Analog, vom spune că poziţia pe care se stochează numărul de locuri în fişier este o *resursă critică*, deoarece nu poate fi accesată simultan de mai multe procese. In sfârşit, pentru a evita situaţia eronată de mai sus vom spune că procesele A şi B trebuie să se *excludă reciproc*, deoarece trebuie să aibă acces exclusiv la secţiunea şi la resursa critică [22].

Problema secţiunii critice a suscitat un interes deosebit în istoria **SO**. Vom expune şi noi câteva aspecte ale ei, pentru a se vedea câteva dintre "subtilităţile" programării concurente. Să vedem mai exact cerinţele problemei:

- la un moment dat, numai un singur proces este în secţiunea critică; orice alt proces care solicită accesul la ea, îl va primi după ce procesul care o ocupă a terminat de executat instrucţiunile secţiunii critice;
- vitezele relative ale proceselor sunt necunoscute;
- oprirea oricărui proces are loc numai în afara secţiunii critice;
- nici un proces nu va aştepta indefinit pentru a intra în secţiunea critică.

Pentru a fixa ideile, presupunem că fiecare proces execută în mod repetat, de un număr nedefinit de ori, întâi secţiunea critică şi apoi restul programului. Descrierea problemei astfel puse este dată în limbaj pseudo-cod în fig. 9.4.

```
do {
  <secţiune critică>
  <rest program>
} while(false);
```

Figura 9.4 Un program cu secţiune critică

Au existat multe încercări de rezolvare a problemei secţiunii critice. In [4] sunt date şi comentate trei încercări de descriere, fiecare dintre ele încălcând unele dintre condiţiile puse mai sus. Matematicianul Dekker a fost primul (1965) care a dat o soluţie software (rezonabilă, care însă este destul de complicată şi încâlcită). In 1981 Peterson a dat o soluţie ceva mai simplă. Aceste soluţii se referă la rezolvarea secţiunilor critice apărute la concurenţa în cadrul aceluiaşi proces (un subiect pe care nu l-am abordat încă). Totuşi, pentru a evidenţia complexitatea problemei, prezentăm soluţia lui Peterson în fig 9.5 şi lăsăm cititorului "plăcerea" de a demonstra că soluţia este corectă.

Cod iniţial	int c1 = 1;
	int c2 = 1;
	int schimb;	

Cod executat în paralel	do {
	  c1 = 0;
	  schimb = 1;
	  while((c2 == 0) &&
	  (schimb == 1)){
	  <aşteaptă>;
	  }
	  <secţiune critică>;
	  c1 = 1;
	  <rest program 1>
	} while(false)
		do {
		  c2 = 0;
		  schimb = 2;
		  while((c1 == 0) &&
		  (schimb == 2)){
		  <aşteaptă>;
		  }
		  <secţiune critică>;
		  c2 = 1;
		  <rest program 2>
		} while(false)

Figura 9.5 Soluţia Peterson pentru secţiune critică

Evident, rezolvarea nu este simplă. Dacă se cere ca secţiunea critică să fie accesibilă la mai multe procese, soluţia din fig. 9.5 nu poate fi generalizată prea uşor. Mai mult, soluţiile software presupun o *aşteptare activă*, adică toate procesoarele, în faza de aşteptare execută în mod repetat o aceeaşi instrucţiune (ciclează încontinuu). Ar fi de dorit ca pe timpul cât un proces aşteaptă să primească dreptul de execuţie a secţiunii critice, procesorul lui să devină disponibil spre a servi alte procese. O asemenea soluţie poate fi realizată folosind conceptul de *semafor* descris în secţiunea următoare.

9.1.2.2 Conceptul de semafor.

Pentru a evita situaţia de mai sus şi pentru a permite o serie de operaţii cu procese (pe care le vom prezenta mai târziu), Dijkstra a introdus conceptul de semafor.

Un *semafor s* este o pereche

`(v(s), c(s))`

unde **`v(s)`** este valoarea semaforului, iar **`c(s)`** o coadă de aşteptare. Valoarea **`v(s)`** este un număr întreg, care primeşte o valoare iniţială **`v0(s)`.** Coada de aşteptare conţine (pointeri la) procesele care aşteaptă la semaforul **`s`**. Iniţial coada este vidă, iar disciplina cozii depinde de sistemul de operare (FIFO, LIFO, priorităţi etc.).

Pentru gestiunea semafoarelor se definesc două operaţii indivizibile **`P(s)`** şi **`V(s)`** ale căror roluri, exprimate laconic, sunt "a trece de resursă" şi "a anunţa eliberarea resursei". (Denumirile operaţiilor au fost date de Dijkstra după primele litere din limba olandeză: P de la *prolaag* prescurtare de la *probeer te verlagen* – încercare de a descreşte, respectiv V de la *verhoog* – a creşte). In unele lucrări [4], operaţia **`P`** se mai numeşte **WAIT**, iar operaţia **`V`** se

mai numeşte **SIGNAL**. Indivizibilitatea operaţiilor înseamnă că ele nu pot fi întrerupte în cursul execuţiei lor, deci nu pot exista două procese care să execute simultan **P**(s) sau **V**(s), sau simultan două operaţii **P**, sau simultan două operaţii **V**.

Definiţiile exacte ale operaţiilor **P** şi **V**, apelate de un proces A şi aplicate asupra unui semafor sunt date în fig. 9.6.

```
// P(s) apelat de procesul A:

v(s) = v(s) - 1;
if(v(s) < 0) {
  STARE(A) = WAIT;
  c(s) ← A;
    //Procesul A intra în aşteptare
  <Trece controlul la DISPECER>;
} else {
  <Trece controlul la procesul A>;
}
```

```
// V(s) apelat de procesul A

v(s) = v(s) + 1;
if(v(s) <= 0) {
  c(s) → B;
    // Extrage din coadă alt proces B
  STARE(B) = READY;
  <Trece controlul la DISPECER>;
} else {
  <Trece controlul la procesul A>;
}
```

Figura 9.6 Operaţiile P(s) şi V(s) apelate de procesul A

Pentru a sugera funcţionarea (şi denumirea) semaforului vom considera un exemplu concret. Fie G1 şi G2 două gări legate prin *n* linii paralele, pe care se circulă numai de la G1 spre G2. Presupunem că în G1 intră mai mult de *n* linii. Procesul trecerii trenurilor va fi dirijat astfel:

Valoarea iniţială: `v0(s) = n`

Procesul de trecere:

```
P(s);     <Trenul trece pe una din cele n linii>;    V(s);
```

Să notăm cu *np*(*s*), *nv*(*s*), numărul de primitive **P**(s), respectiv **V**(*s*) efectuate până la un moment dat, *v0*(*s*) valoarea iniţială a semaforului *s*, iar *nt*(*s*) numărul proceselor care au trecut de semaforul *s*. Următoarele proprietăţi au loc:

1. `v(s) = v0(s) - np(s) + nv(s) ;`
2. `dacă v(s)<0` atunci `în c(s)` există `-v(s)` procese;
3. `dacă v(s)>0` atunci `v(s)` procese pot trece succesiv de semaforul s fără să fie blocate ;
4. `nt(s) = min(v0(s) + nv(s), np(s))`

Demonstraţia acestor proprietăţi este simplă, prin inducţie după numărul operaţiilor **P** şi **V** efectuate asupra semaforului. Aspectul critic al semaforului este că operaţiile asupra lui (**P** şi **V**) sunt atomice. Astfel două procese nu le pot executa în acelaşi timp asupra aceluiaşi semafor *s*.

Folosirea semafoarelor rezolvă complet şi elegant problema secţiunii critice prezentată în secţiunile anterioare. Rezolvarea presupune folosirea unui singur semafor *s* şi este corectă indiferent de numărul de procese care folosesc în comun secţiunea critică. Descrierea soluţiei este dată în fig. 9.7.

Cheia este tocmai semaforul *s*. Dacă `v0(s)=1` şi toate procesele care folosesc secţiunea critică sunt de forma:

```
...; P(s);  <secţiune critică>; V(s); ...
```

atunci se demonstrează uşor că $v(s)<1$, indiferent de numărul proceselor care operează asupra lui *s* cu operaţii **P** sau **V**. Drept urmare, în baza proprietăţilor menţîonate mai sus, un singur proces poate trece de acest semafor, deci sînt îndeplinite condiţiile cerute pentru a proteja o secţiune critică. Un astfel de semafor mai poartă numele de *semafor de excludere mutuală.* Sincronizarea este o operaţie fundamentală în programarea concurentă. In secţiunile care urmează vom întâlni mai multe aplicaţii ale ei.

Cod iniţial	sempahore s; v0(s) = 1;	
Cod executat în paralel	do { P(s); <secţiune critică>; V(s); <rest program 1> } while(false)	do { P(s); <secţiune critică>; V(s); <rest program 2> } while(false)

Figura 9.7 Soluţia cu semafor pentru secţiune critică

9.1.2.3 Problema producătorului şi a consumatorului.

Să presupunem că există unul sau mai multe procese numite *producătoare*, şi unul sau mai multe procese numite *consumatoare*. De exemplu, conceptele de pipe şi FIFO sunt de această natură (vezi 5.3.2 şi 5.3.4). Transmiterea informaţiilor de la producători spre consumatori se realizează prin intermediul unui buffer cu *n* intrări pentru *n* articole. Fiecare producător depune câte un articol în buffer, iar fiecare consumator scoate câte un articol din buffer. Problema constă în a dirija cele două tipuri de procese astfel încât:

- să existe acces exclusiv la buffer;
- consumatorii să aştepte când bufferul este gol;
- producătorii să aştepte când bufferul este plin.

Rezolvarea prin semafoare este foarte simplă. Se vor folosi trei semafoare: *gol, plin* şi *exclus.* Rezolvarea este descrisă în fig. 9.8. Semaforul *exclus* asigură accesul exclusiv la buffer în momentul modificării lui, pentru a nu permite depuneri şi extrageri de date simultane. Semaforul *plin* indică numărul de poziţii ocupate din buffer, iar semaforul *gol* numărul de poziţii libere din buffer. Depunerea unui articol în buffer necesită decrementarea (**P**) semaforului *gol* şi incrementarea (**V**) semaforului *plin* (practic o poziţie este "mutată" din semaforul *gol* in semaforul *plin*). Extragerea unui articol din buffer este operaţia inversă depunerii.

Producător	**Consumator**

semaphore plin, gol, exclus;
v0(plin) = 0;
v0(gol) = n;
v(exclus) = 1;	

do { <produce articol>; P(gol); P(exclus); <depune articol în buffer>; V(exclus); V(plin); } while(false);	do { P(plin); P(exclus); <extrage articol din buffer>; V(exclus); V(gol) <consuma articol>; } while(false);

Figura 9.8 Problema producătorului şi a consumatorului rezolvată cu semafoare

9.1.2.4 Regiuni critice condiţionate.

Prin intermediul operaţiilor de excludere şi sincronizare pot fi rezolvate o serie de probleme clasice de concurenţă. Dintre acestea au fost prezentate problema secţiunii critice şi problema producătorului şi a consumatorului. Am văzut că descrierea lor cu ajutorul semafoarelor este destul de comodă. Din păcate, operaţiile **P** şi **V**, lăsate libere la îndemâna programatorului, pot provoca necazuri mari. Incercaţi să vedeţi ce se întâmplă dacă la descrierea din fig. 9.8 se inversează între ele operaţiile **P**(*gol*) şi **P**(*exclus*) din procesul producător. Acesta este motivul pentru care au fost introduse construcţii structurate de concurenţă. Prin intermediul lor se poate exercita un control riguros asupra încălcării regulilor de concurenţă.

Vom descrie o astfel de construcţie structurată, numită *regiune critică condiţionată*. Ea înglobează într-un mod elegant conceptele de secţiune critică şi resursă critică despre care am vorbit mai sus, cu posibilitatea (eventuală) de a executa regiunea numai dacă este îndeplinită o anumită condiţie.

Fiecărei regiuni critice i se asociază o *resursă* constând din toate variabilele care trebuie protejate în regiune. Declararea ei se face astfel:

```
resource r :: v1, v2, ..., vn
```

unde `r` este numele resursei, iar `v1 , ..., vn` sunt numele variabilelor de protejat.

O *regiune critică condiţionată* se defineşte astfel:

```
region r [ when B ] do S
```

unde `r` este numele unei resurse declarate ca mai sus, `B` este o expresie booleană, iar `S` este secvenţa de instrucţiuni corespunzătoare regiunii critice. Dacă este prezentă opţiunea `when`, atunci `S` este executată numai dacă `B` este adevărată. Variabilele din resursa `r` pot fi folosite numai de către instrucţiunile din secvenţa de instrucţiuni `S`.

Descrierea prin semafoare a unei regiuni critice condiţionate este dată în fig. 9.9. Sunt necesare două semafoare şi un număr întreg. Semaforul `sir` reţine în coada lui toate procesele care solicită acces la regiune. Semaforul `exclus` asigură accesul exclusiv la anumite secţiuni ale implementării (în special la modificarea numărului întreg `nr`). Intregul `nr` reţine câte procese au cerut acces la regiune, la un moment dat. In secţiunea următoare vom da o aplicaţie interesantă a regiunii critice condiţionate.

In fig 9.9 algoritmul propriu-zis începe de la linia 9. Dacă regiunea critică este folosită fără o condiţie asociată, se va trece direct la execuţia lui `S`. Altfel, incrementând variabila `nr`, se marchează faptul că încă un proces accesează regiunea critică cu o condiţie asociată. Cum mai multe procese pot face acest lucru simultan, incrementarea trebuie protejată. Deşi la prima vedere este doar o singură operaţie, în realitate incrementarea se traduce într-o secvenţă de instrucţiuni binare care pot fi executate într-o ordine similară cu cea din fig 9.2 şi poate conduce la erori. Ca urmare, înainte ca incrementarea să aibă loc, se execută **P**(`exclus`) care dacă condiţia `B` este satisfăcută va proteja şi execuţia lui `S`. Altfel, dacă condiţia `B` nu este îndeplinită procesul trebuie să intre în aşteptare. Aceasta se face prin **P**(`sir`), dar nu înainte de a elibera semaforul `exclus` prin **V** (`exclus`). Dacă **V** (`exclus`) este omis, practic orice alt proces care ar accesa regiunea critică se va opri la linia 9. In acest fel concurenţa pentru

evaluarea condiţiei *B* ar fi. Scopul nostru însă nu este serializarea evaluării condiţiei *B* şi a secvenţei de instrucţiuni *S*, ci serializarea accesului la *S* în momentul în care *B* este satisfăcută. Ca urmare trebuie să lăsăm fiecare proces să evalueze *B* fără restricţii. Dacă condiţia *B* este îndeplinită, se decrementează *nr* (marcând astfel intrarea în regiunea critică) şi se execută instrucţiunile din *S*. Apoi (liniile 20-21), semaforul *sir* este incrementat până la zero în aşa fel încât controlorul va alege unul din procesele din coadă pentru a fi executat. In final semaforul *exclus* este eliberat la linia 22. Procesul proaspăt reactivat va reevalua condiţia B şi dacă este îndeplinită va continua şi va executa instrucţiunile din *S*.

```
 1 // Valorile de pornire
 2 semaphore sir, exclus;
 3 int nr, i, cuConditie;
 4 v0(sir) = 0;
 5 v0(exclus) = 1;
 6 nr = 0;
 7
 8 //Codul corespunzător regiunii
 9 P(exclus);
10 if(cuConditie) {
11   nr++;
12   while(!B) {
13     V(exclus);
14     P(sir);
15     P(exclus);
16   }
17   nr--;
18 }
19 S;
20 for(i=0; i<nr; i++)
21   V(sir);
22 V(exclus);
```

Figura 9.9 Implementarea unei regiuni critice condiţionate

Până în acest moment lucrurile ar părea clare, dar există o situaţie în care implementarea de mai jos nu funcţionează! Este vorba de cazul în care toate procesele accesează regiunea critică condiţionată în momentul în care *B* nu este satisfacută şi ca urmare vor intra în aşteptare în coada semaforului *sir*. Intrebarea este: cum vor ieşi aceste procese din caoda semaforului? Este necesar ca cineva (un proces) să modifice starea condiţiei B şi apoi să anunţe schimbarea proceseleor care aşteaptă. Aceasta poate avea loc în două moduri:

1. După câtva timp un alt proces schimbă starea condiţiei *B*, intră în regiunea critică şi constată că *B* este satisfăcută. Ca urmare, nu va intra în coada semaforului *sir* şi după execuţie lui *S* îi va incrementa valoarea până la zero (liniile 20-21). In acest moment unul dintre procesele blocate în coada semaforului va fi rectivat şi va trece prin scţiunea critică. La rândul lui, acesta va reincrementa semaforul până la zero astfel reactivand un nou proces.
2. Unul dintre procesele care folosesc regiunea critică nu o asociază cu condiţia *B* şi ca urmare va omite aşteptarea la semaforul *sir*. La fel ca şi în cazul anterior, după execuţia lui S, acest proces va incrementa semaforul activând procesele din coada lui.

Un program care se bazează pe varianta 1 trebuie să se asigure ca întotdeuna va exista un proces care va găsi condiţia *B* satisfăcută, altfel totul se blochează. Varianta a doua este mai puţin riscantă pentru că un proces apelând regiunea critică fără condiţie nu va rămâne blocat. Pentru a cuprinde ambele într-un singur program, s-a adăugat condiţia din linia 10 cu închiderea ei din linia 18.

9.1.2.5 Problema citirilor şi a scrierilor.

Problema a fost formulată în 1971 de către Courtois, Heymans şi Parnas [4] [50]. Se presupune că există două tipuri de procese: *cititor* şi *scriitor*. Ele partajează împreună o aceeaşi resursă, să zicem un fişier. Un proces scriitor modifică conţinutul fişierului, iar unul cititor consultă informaţiile din el. Spre deosebire de problema producătorului şi a consumatorului (unde toate procesele aveau acces exclusiv la resursă), la acest tip de problemă *orice proces scriitor are acces exclusiv la resursă, în timp ce mai multe procese cititor pot avea acces simultan la ea.*

Intre cerinţele problemei trebuie să se aibă în vedere ca nici un proces să nu fie nevoit să aştepte indefinit. De asemenea, trebuie să se precizeze, pentru cazurile de acces simultan, care tip de proces este mai prioritar. In prima versiune era stabilit ca procesele cititor să fie mai prioritare. Aceasta putea conduce la situaţia ca procesele scriitor să aştepte indefinit dacă o infinitate de procese cititoare cer accesul. Versiunile ulterioare au dat prioritate proceselor scriitor. Drept urmare, este posibil ca procesele cititor să aştepte indefinit.

In secţiunile precedente am văzut rezolvarea diferitelor probleme de programare concurentă, folosind cel mai des operaţiile **P** şi **V**. Ele sunt suficiente pentru descrierea acestei probleme şi invităm cititorul să o facă. In secţiunea precedentă am definit o construcţie mai elegantă şi mai uşor de urmărit în programe, regiunea critică condiţionată.

O posibilă rezolvare, folosind regiuni critice condiţionate, este cea din fig. 9.10. Soluţia aparţine lui P. B. Hansen, 1973, şi ea dă prioritate proceselor scriitor faţă de cele cititor. Un proces scriitor intră în lucru imediat după ce procesele cititor active şi-au încheiat citirile curente. Resursa `f` este zona la care doresc accesul cele două tipuri de procese. Presupunem că ea este un fişier, iar operaţiile de bază asupra lui sunt `read` şi `write`. Resursa `c` conţine două contoare care indică câte procese de fiecare tip au cerut acces la resursa `f`. Este de remarcat folosirea regiunii `c` cu şi fără o condiţie ataşată.

Cititor	Scriitor

`int nw, nr;`
`resource f;`
`resource c :: nr, nw;`
`nr = 0;`
`nw = 0;`	

`region c`
`  when nw == 0`
`  do nr++;`
`<citeşte date>;`
`region c`
`  do nr--;`
	`region c`
	`  do nw++;`
	`region c`
	`  when nr == 0`
	`  do <aşteaptă să se termine toţi cititorii activi>;`
	`region f`
	`  do <scrie date>;`
	`region c`
	`  do nw--;`

Figura 9.10 Problema citirilor şi a scrierilor

9.1.3 Problema impasului

9.1.3.1 Conceptul de impas.

Noţiunea de impas nu este specifică sistemelor de operare sau informaticii, ci este un fenomen întâlnit în multe situaţii din viaţa de zi cu zi. In general, impasul poate fi definit ca o blocare a actvităţii normale a două sau mai multe entităţi ca urmare a efectului lor coroborat asupra mediului.

Un exemplu de impas uşor de vizualizat este cel al traficului în jurul unui cvartal. După cum se vede în fig 9.11, cele 8 maşini sunt blocate în mod circular, şi traficul nu poate continua fără ca una dintre ele să dea înapoi.

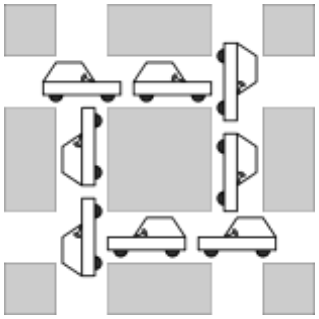

Figura 9.11 Un impas într-o intersecţie

Trecând la domeniul informatic, o modalitate uşoară de a crea o situaţie de impas este invesarea operaţiilor **P** şi **V** în procese concurente. Fig 9.12 prezintă codul celor două procese.

Proces A	Proces B

semaphore x, y;
v0(x) = 1;
v0(y) = 1;	

P(x); <instrucţiuni A>; P(y); V(y); V(x);	P(y); <instrucţiuni B>; P(x); V(x); V(y);

Figura 9.12 Impas provocat de două semafoare

Dacă instrucţiunile celor două procese sunt executate întâmplător în secvenţa din figura 9.13, cele două procese ***A*** şi ***B*** se vor afla într-o stare de impas. Cauza este oarecum similară cu situaţia traficului prezentată anterior. Procesul ***A*** va deţine semaforul binar ***x***, procesul ***B*** va deţine semaforul binar ***y***, apoi amble vor încerca să obţină semaforul deţinut de celălalt, fără a elibera semforul pe care îl deţine inainte de a-l obţine pe cel deţinut de celălat proces. Situaţia este ilustrată în fig. 9.13.

Proces A	Proces B
P(x);	
	P(y);
<instrucţiuni A>;	
	<instrucţiuni B>;
P(y);	
wait ...	P(x) wait...

Figura 9.13 O situaţie de impas

Cauza principală care a generat această situaţie este faptul că procesele îşi ocupă resursele numai atunci când au nevoie efectiv de ele şi ca urmare apare situaţii de aşteptare circulară.

Apropiindu-ne mai mult de funcţionarea unui sistem de operare, să presupunem că în sistem sunt numai două benzi magnetice ***MM0*** şi ***MM1***. Dacă un proces cere mai întâi ***MM0*** şi apoi

MM1 fără a elibera ***MM0***, iar al doilea cere ***MM1*** şi apoi ***MM0*** fără a elibera ***MM0***, poate apare o astfel de aşteptare circulară. Acest fenomen este cunoscut în literatură sub mai multe denumiri, şi anume: *impas, interblocare, deadlock, deadly embrace* etc. [4], [10], [51].

Ultimele două exemple de *impas* au avut ca protagonişti câte două procese şi două resurse. Impasul poate apărea însă în situaţii mai complexe care implică nu doar două procese, ci mai multe şi nu neaparat resurse specifice, ci clase de resurse. Să presupunem că într-un anume moment din funcţionare, un sistem de operare dispune de 50MB de memorie liberă şi de 10 poziţii libere pnetru crearea de conexiuni în reţea. In acest moment un proces ***A*** cere alocarea a 30MB de memorie şi apoi a 7 conexiuni în reţea, iar un proces ***B*** cere întâi alocarea a 5 conexiuni de reţea şi apoi alocarea a 40MB de memorie. Similar cu exemplul anterior, este posibil ca ambelor procese să li se satisfacă primele cereri, însă după aceea din lipsă de resurse va apare impas. Acest tip de impas este mai greu de detectat pentru ca nu implică resurse punctuale ci clase de resurse, în cazul nostru memoria RAM şi conexiunile la reţea.

Impasul rezultat din conflictul asupra câtorva resurse punctuale poate fi modelat teoretic pe baza impasului pe clase, considerând că fiecare clasă de elemente conţine un singur element. Altfel spus, impasul pe clase este o generalizare a impasului pe resurse punctuale.

Impasul este o stare gravă care poate duce la un blocaj al sistemului de operare sau la distrugerea unor procese. In cazul în care două procese se blochează unul pe altul, doar ele vor fi în impas şi situaţia se poate salva terminând forţat unul dintre ele. Dacă însă impasul se manifestă între un proces şi sistemul de operare sau direct în interiorul sistemului de operare, parctic sistemul "îngheaţă" şi işirea cea mai frecventă din asemenea situaţii este restartarea calculatorului!

Astfel de situaţii trebuie avute în vedere şi prevenite de către proiectanţii de sisteme de operare. Dar, după cum vom vedea, tratarea impasului este costisitoare şi ca urmare proiectanţii o tratează doar în situaţii izolate pentru a nu afecta performanţa sistemului.

In legătură cu impasul, fiecare sistem de operare trebuie să dea rezolvări măcar la una dintre următoarele probleme:

- ieşirea din impas,
- detectarea unui impas,
- evitarea (prevenirea apariţiei) impasului.

Cele două rezultate de mai sus dau condiţii prin care este indicată eventuala apariţie a unui impas, însă nu furnizează metode practice de rezolvare a problemelor de mai sus. In continuare vom da metode şi tenhici de rezolvare a problemelor generate de impas.

9.1.3.2 Ieşirea din impas.

Soluţiile ieşirii din impas pot fi clasificate în manuale şi automate. Soluţiile manuale sunt cele care implică acţiuni ale operatorului uman. Cele automate sunt executate de către sistemul de operare.

Ieşirea din starea de impas automată (de către sistemul de operare) presupune în primul rând detectarea acestei stări. La rândul ei, detectarea stării de impas implică existenţa unei

componente a sistemului de operare care să întreţină şi să analizeze starea alocărilor de resurse din sistem (vom discuta acest aspect mai jos). Pe baza acestei analize, sistemul de operare recunoaşte starea de impas şi trebuie să aleagă o soluţie pentru a-l rezolva.

Indiferent dacă este automată sau manuală, soluţia ieşirii din impas necesită acţiuni drastice. In cele ce urmează prezentăm trei astfel de soluţii.

O primă variantă este *reîncărcarea sistemului de operare*. Soluţia nu prea este de dorit, dar este cea mai la îndemâna operatorului. Dacă pierderile nu sunt prea mari se poate adopta şi o astfel de strategie.

O a doua varintă este *alegerea un proces "victimă"*, care este fie cel care a provocat impasul, fie un altul de importanţă mai mică, astfel încât să fie înlăturat impasul. Se distruge acest proces şi odată cu el toţi descendenţii lui. Această alegere este o parte importantă a algortimilor automaţi de rezolvarea a impasului. In cazul abordării manuale, din păcate, criteriul de alegere a "victimei" este de multe ori departe de a fi obiectiv.

O a treia soluţie este *cre*erea *unui "punct de reluare", care este o fotografie a memoriei pentru procesul victimă* şi pentru procesele cu care el colaborează. Apoi se distruge procesul victimă şi subordonaţii lui. Procesul victimă se reia ulterior. Crearea punctelor de reluare reclamă consum de timp şi complicaţii, motiv pentru care de multe ori nu se execută.

9.1.3.3 Detectarea impasului

Detectarea unui impas se face atunci când **SO** nu are un mecanism de prevenire a impasului. Pentru a se reuşi detecţia este necesar ca **SO** să aibă o evidenţă clară, pentru fiecare proces, privind resursele ocupate, precum şi cele solicitate dar neprimite. Pentru aceasta, cel mai potrivit model este *graful alocării resurselor*.

Facem ipoteza, că fiecare resursă este într-un singur exemplar şi vom nota `R1, R2, ... , Rm` aceste resurse. Fie acum `P1, P2, ... , Pn` procesele din sistem. Se construieşte un graf bipartit (**X**, **U**) astfel:

- **X** = {`P1, P2, ..., Pn, R1, R2, ..., Rm`}
- `(Rj, Pi)` este arc în **U** dacă procesul `Pi` a ocupat resursa `Rj`.
- `(Pi, Rj)` este arc în **U** dacă procesul `Pi` aşteaptă să ocupe resursa `Rj`.

Dacă graful (**X**, **U**) definit mai sus este ciclic, atunci sistemul se află în stare de impas. Demonstraţia este un exerciţiu simplu de inducţie. Detectarea ciclicităţii unui graf se face folosind algoritmii clasici din teoria grafelor.

In fig 9.14 prezentăm un posibil graf de alocare. Nodurile pătrate reprezintă procesele iar nodurile rotunde reprezintă resursele. Pentru o mai uşoară diferenţiere arcele de aşteptare pentru o resursă au fost desenate cu linie întreruptă.

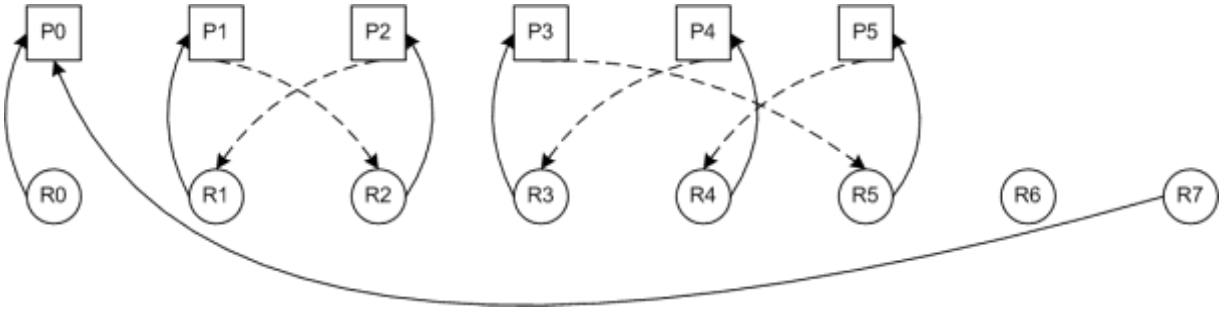

Figura 9.14 Un graf de alocare a resurselor

Dacă selectăm numai subgraful cu nodurile {P1, P2, R1, R2}, iar pentru resursele R1 şi R2 considerăm unităţile de bandă MM0 şi MM1, atunci se obţine graful de alocare pentru unul dintre exemplele prezentate în 9.1.3.1.

Deşi nu sunt evidente la prima vedere, graful din figura de mai sus conţine două cicluri. Primul ciclu implică procesele P1 şi P2, şi resursele R1 şi R2. Al doilea ciclu implică procesele P3, P4 şi P5, şi resursele R3, R4, şi R5. In ambele cazuri aveam de-a face cu stări de impas.

9.1.3.4 Evitarea (prevenirea apariţiei) impasului.

In 1971, Coffman, Elphic şi Shoshani [4], [19] au indicat patru condiţii *necesare* pentru apariţia impasului:

1. procesele solicită controlul exclusiv asupra resurselor pe care le cer (condiţia de *excludere mutuală*);
2. procesele păstrează resursele deja ocupate atunci când aşteaptă alocarea altor resurse (condiţia de *ocupă şi aşteaptă*);
3. resursele nu pot fi şterse din procesele care le ţin ocupate, până când ele nu sunt utilizate complet (condiţia de *nepreempţie*);
4. există un lanţ de procese în care fiecare dintre ele aşteaptă după o resursă ocupată de altul din lanţ (condiţia de *aşteptare circulară*).

Evitarea impasului presupune împiedicarea apariţiei uneia dintre aceste patru condiţii. Vom lua fiecare condiţie pe rând şi vom analiza modalităţile în care ea poate fi împiedicată şi efectele asupra funcţionării sistemului

Condiţia 1 – excluderea mutuală. In absenţa excluderii mutuale între procese, dispare noţiunea de aşteptare după o resursă critică blocată de alt proces. Dispărând noţiunea de aşteptare, evident dispare şi noţiunea de impas din moment ce nici un proces nu se va opri din execuţie. Eliminarea acestei condiţii însă nu este o alegere fericită. După cum am demonstrat anterior, excluderea mutuală este esenţială pentru pastrarea corectidudinii datelor. Practic imasul a apărut ca şi un efect secundar al aplicării acestei condiţii indiscutabil necesare.

Condiţia 2 – ocupă şi aşteaptă. Această condiţie spune că un proces poate aştepta după o resursă în timp ce deţine (blochează) alte resurse. Eliminarea acestei condiţii poate fi făcută în două moduri:

- Procesul trebuie să elibereze toate resursele blocate înainte de a solicita o nouă resursă.
- Procesul trebuie să blocheze toate resursele de care are nevoie la pornire.

Prima variantă este destul de forţată pentru că cere procesului să concureze pentru toate resursele ori de câte ori ar avea nevoie de una nouă.

A doua variantă nu este aplicabilă sistemelor de operare interactive (Windows, Unix etc.) pentru că ar restrânge posibilităţile de uitilizare a sistemului. Să luăm spre exemplu cazul comun al unui editor de text care suportă deschiderea simultană a mai multor fişiere. Dacă sistemul de operare ar impune blocarea tuturor resurselor la încărcarea procesului, editorul nostru ar fi obligat să solicite o cantitate finită de memorie pe care să o folosească la încărcarea fişierelor editate. Efectul acestui comportament are două tăişuri:

- Dacă editorul deschide un singur fişier mic, practic majoritatea memoriei solicitate va fi ţinută blocată fără motiv. Evident ar fi mult mai util ca sistemul de operare să dispună de această memorie pentru a servi alte procese.
- Dacă editorul trebie să dechidă multe fişiere de dimensiuni mari, va ajunge să ocupe toată memoria alocată şi să fie nevoit să refuze deschiderea altor fişiere.

Condiţia 3 – nepreempţie. Eliminarea condiţiei de nepreempţie permite sistemului de operare să aleagă procesele care ar trebui oprite pentru a ieşi din impas şi chiar să le oprească. Evident simpla terminare a unui proces nu este acceptabilă şi ca urmare sistemul trebuie să şi repornească acel proces. Deşi această abordare rezolvă situaţia de impas, nu garantează că procesele vor funcţiona conform aşteptărilor. Teoretic este posibil ca după restartarea unui proces dintr-un grup implicat într-un impas, procesele rămase să nu progreseze suficient pentru a ieşi din regiunea critică înainte ca programul repornit să reajungă acolo. In acel moment impasul va reapare şi din nou unul dintre procese va fi repornit. Această secvenţa se poate repeta la nesfârşit şi deşi nici unul dintre procese nu este blocat, totuşi niciunul nu progrseză cu execuţia. Acest fenomen este cunoscut sub numele de *impas activ* (livelock).

Condiţia 4 – aşteptarea circulară. O soluţie simplă pentru împiedicarea aşteptării circulare este impunerea unei ordini de blocare a resurselor. Sistemul de operare decide o ordine absolută a tuturor resurselor şi nici un proces nu are voie să blocheze o resursă cu număr de ordine mai mare înaintea uneia cu un număr de ordine mai mic. Pentru a vedea efectiv efectul blocării ordonate, vom prezenta o *problema filozofilor*.

Să considerăm o masă circulară la care sunt aşezaţi opt filozofi. Pe masă, în loc să fie şaisprezece tacâmuri (furculiţe şi cuţite) sunt doar opt, aşezate în aşa fel încât lângă fiecare farfurie este un cuţit şi o furculiţă (vezi fig 9.15). Fiecare filozof are nevoie de souă tacâmuri pentru a mânca, şi odată ca a luat un tacâm de pe masă nu-l va elibera până când nu termină de mâncat. Dacă toţi filozofii iau de pe masă tacâmul din stânga, ne aflăm se creează o situaţie de impas, pentru că nici un filozof nu va avea două tacâmuri pentru a putea mânca.

Impunerea blocării resurselor (luării tacâmurilor) în ordine, va preveni impasul pentru că filozoful H nu va putea ridica tacâmul 8 înaintea tacâmului 7. In consecinţă filozoful G va avea la dispoziţie două tacâmuri pentru a mânca. In momentul în care G va termina, va elibera tacâmul 6 şi filozoful F va putea mânca şi el. Procesul se va continua astfel până când toţi filozofii vor fi mâncat.

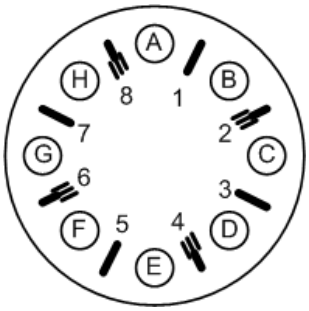

Figura 9.15 Problema filozofilor

In realitate însă, acestă soluţie este dificil de implementat pentru programatorii unui sistem de operare. Având în vedere cantitatea mare de resurse dintr-un sistem (de ordinul miilor) respectarea unei ordini prestabilite devine practic imposibilă. In plus, ordinea resurselor va fi de cele mai multe ori diferită de ordinea logică în care un program are nevoie de resurse, cauzând astfel scrierea de algoritmi mult mai complecşi sau mai puţin performanţi.

9.1.3.5 Alocarea controlată (conservativă) de resurse; algoritmul bancherului.

Evitarea impasului se poate face şi impunând o modalitate restrictivă de alocare de resurse. In esenţă, înainte de a servi o cerere de resurse, sistemul de operare analizează alocările de resurse. Dacă servirea cererii conduce la o stare cu potenţial de a genera un impas, cererea de resurse este amânată şi reanalizată în momentul în care are loc a modificarea în starea alocării de resurse în sistem. Soluţiile pe care le vom discuta vor adresa cazul general al impasului în care lucrăm cu clase de resurse.

Pentru a evalua dacă o stare a alocărilor poate conduce la impas este necesară cunoaşterea pentru fiecare proces a necesităţilor maxime de resurse din fiecare clasă. Astfel, la încărcare, fiecare proces va transmite sistemului de operare cantităţile de resurse din fiecare clasă de care va avea nevoie în cel mai rău caz.

Soluţie ineficientă. Folosind aceste maxime declarate de fiecare proces la încărcare, este foarte simplu să evităm impasul. Dacă adunăm maximul declarat de un nou proces pentru o resursă la cantitatea din acea resursă deja alocată şi suma depăşeşte maximul disponibil în sistem, pur şi simplu punem procesul în aşteptare până când se vor elibera suficiente resurse.

Aspectul pozitiv al acestei soluţii este că garantează servirea imediată a tuturor cererilor de resurse venite din partea proceselor încărcate. Acest lucru este posibil pentru că algoritmul menţine întotdeauna un număr redus de procese încărcate pentru a avea resurse suficiente pentru ele.

Soluţia este ineficientă pentru că este foarte puţin probabil ca un proces să folosească imediat după încărcare maximul declarat de resurse. In general resursele sunt folosite treptat şi rareori se va atinge maximul. Ca urmare, în majoritatea covârşitoarea a cazurilor un proces poate să-şi înceapă lucrul fără ca sistemul de operare să poată acoperi întreaga plaja de resurse pe care o poate solicita. Algortimul bancherului evită impasul fără a impune condiţii atât de restrictive.

Algoritmul Bancherului [4], [19] datorat lui Dijkstra este probabil cel mai renumit pentru alocarea controlată. Spre diferenţă de soluţia precedentă algoritmul bancherului permite tuturor proceselor să se încarce şi să înceapă execuţia. In schimb, algoritmul nu garantează servirea imediată a tuturor cererilor proceselor încărcate (un lucru de altfel rezonabil). Tot ce

garantează algoritmul bancherului este că orice proces încărcat poate fi servit (nu va rămane blocat la nesfârşit) cu condiţia să aştepte până când se mai eliberează resurse.

Pentru a clarifica numele algoritmului, funcţionarea algoritmului este similară cu procedura prin care un bancher (sistemul de operare) decide când şi cât (resurse) să împrumute unui client (procesul) dintr-o sumă maximă cu care îl poate credita. Diferenţele faţă de lumea reală sunt:

1. Bancherul dispune de o sumă limitată de împrumut, mai mică decât suma creditelor maxime ale clienţilor
2. Bancherul nu percepe nici dobândă şi nici comision
3. Toţi clienţii returnează banii în cele din urmă

Inainte de a prezenta algoritmul propriu-zis trebuie să prezentăm structurile de date pe care le vom folosi.

- `resurseExistente`
  - Cantităţile de resurse existente în sistem pentru fiecare clasă de resurse.
  - Tabel unidimensional. Fiecărei clase de resurse îi corespunde o poziţie
- `maxResursePerProces`
  - Cantităţile maxime de resurse pe care fiecare proces le poate solicita
  - Tabel bi-dimensional. Fiecare linie corespunde unui proces. Fiecare celulă dintr-o linie conţine cantitatea corespunzătoare unei clase de resurse.
- `resurseAlocatePerProces`
  - Cantităţile de resurse alocate fiecărui proces
  - Tabel bi-dimensional. Fiecare linie corespunde unui proces. Fiecare celulă dintr-o linie conţine cantitatea corespunzătoare unei clase de resurse.
- `solicitant`
  - Procesul care solicită resurse
- `cerere`
  - Cererea de analizat emisă de procesul solicitant
  - Tabel unidimensional. Fiecărei clase de resurse îi corespunde o poziţie
- `resurseDisponibile`
  - Resursele disponibile în sistem
  - Tabel unidimensional. Fiecărei clase de resurse îi corespunde o poziţie
- `proceseImplicate`
  - Tablou pentru marcarea proceselor implicate în algoritm
  - Fiecărui proces îi corespunde o poziţie

Folosind notaţiile de mai sus, algoritmul bancherului este prezentat în limbaj pseudocod in fig 9.16.

```
<Fă o copie tabelului "resurseAlocatePerProces" în tabelul "alocari">;

<Dacă nici un proces nu mai este implicat în algoritm, cererea poate
 fi servită. Sfârşit algoritm>;

<Populează tabelul "resurseDisponibile" cu diferenţele
 dintre "resurseExistente" şi sumele resurselor alocate fiecărui
 proces în tabelul "alocari">;

<Găseşte un proces Pi pentru care sunt suficiente resurse pentru ai
 servi cererea maximă>;

<Dacă nu există nici un astfel de proces, însemnă că cererea iniţială
```

```
     nu poate fi servită şi procesul solicitant este pus în
     aşteptare. Sfârşit algoritm>;

    <Simulează terminarea procesului Pi scoţându-l din lista proceselor
     implicate şi eliberându-i resursele (populează cu 0 intrarea lui Pi
     din tabelul "alocari")>;

    <Mergi la pasul 2>;
```

Figura 9.16 Algoritmul bancherului

In esenţă, algoritmul bancherului consideră cererea servită şi apoi verifică dacă există o ieşire din cel mai rău caz posibil, adică atunci când toate procesele îşi cer maximul de resurse. Aceasta se face găsind rând pe rând câte un proces căruia i se poate servi maximul de resurse solicitat, şi apoi simulându-i terminarea. Dacă folosind acest procedeu, se ajunge la terminarea tuturor proceselor, înseamnă că servirea cererii nu implică riscul unui impas.

In cele ce urmează vom oferi o implementare a algoritmului bancherului. Pentru simplitate vom considera că sistemul de operare dispune de patru tipuri de resurse şi că asupra acestor resurse vor acţiona cinci procese. Implementarea conţine şi date de test pentru care algoritmul poate fi verificat.

```
#include<stdio.h>

#define PROCESE 5
#define RESURSE 4


// Date initiale care descriu alocarea de resurse din sistem
int resurseExistente[PROCESE] = {7, 7, 6, 5};

int maxResursePerProces[PROCESE][RESURSE] = {{3, 2, 4, 2},  // P0
                                             {3, 2, 3, 1},  // P1
                                             {4, 1, 2, 3},  // P2
                                             {3, 3, 4, 4},  // P3
                                             {6, 5, 5, 1}}; // P4

int resurseAlocatePerProces[PROCESE][RESURSE] = {{1, 1, 0, 1},  // P0
                                                 {1, 0, 1, 0},  // P1
                                                 {1, 0, 1, 1},  // P2
                                                 {0, 1, 0, 2},  // P3
                                                 {0, 1, 1, 0}}; // P4

void tiparesteStare(int procesAles,
                    int proceseImplicate[PROCESE],
                    int resurseDisponibile[RESURSE],
                    int alocari[PROCESE][RESURSE]) {
  int i, j;

  printf("\nProcese implicate: ");
  for(i=0; i<PROCESE; i++) {
    if(proceseImplicate[i] == 1) {
      printf("P%d  ", i);
    }
  }

  printf("\nProces ales: P%d\n", procesAles);

  printf("Cereri maxime de resurse ale lui P%d:", procesAles);
  for(j=0; j<RESURSE; j++) {
    printf("  %d", maxResursePerProces[procesAles][j]);
```

```
  }

  printf("\nResurse disponibile:");
  for(j=0; j<RESURSE; j++) {
    printf("  %d", resurseDisponibile[j]);
  }

  printf("\nResurse alocate lui P%d:", procesAles);
  for(j=0; j<RESURSE; j++) {
    printf("  %d", alocari[procesAles][j]);
  }
  printf("\n");
}

int verificaCerere(int solicitant, int *cerere) {
  // Variabile de lucru
  int resurseDisponibile[RESURSE];
  int proceseImplicate[PROCESE] = {1, 1, 1, 1, 1};
  int alocari[PROCESE][RESURSE];
  int i, j, procesAles, gasit, n;

  // Copiem tabela de alocariare de resurse la procese si adaugam la copie
  // noua cerere
  for(i=0; i<PROCESE; i++) {
    for(j=0; j<RESURSE; j++) {
      alocari[i][j] = resurseAlocatePerProces[i][j];
      if(i == solicitant) {
        alocari[i][j] += cerere[j];

        // Procesul cere mai multe resurse decat maximul permis
        if(alocari[i][j] > maxResursePerProces[i][j])
          return -1; //Sfarsit algoritm
      }
    }
  }

  while(1) {
    // Calculam cantitatile de resurse disponibile
    for(j=0; j<RESURSE; j++) {
      resurseDisponibile[j] = resurseExistente[j];
      for(i=0; i<PROCESE; i++) {
        resurseDisponibile[j] -= alocari[i][j];

        // Insuficiente resurse disponibile. Starea curenta conduce la impas
        if(resurseDisponibile[j] < 0)
          return 0; // Sfarsit algoritm
      }
    }

    // Calculam numarul de procese inca implicate in algoritm
    n = 0;
    for(i=0; i<PROCESE; i++) {
      n += proceseImplicate[i];
    }

    // Toate procesele au trecut cu bine prin analiza. Cererea va fi servita
    if(n == 0)
      return 1; // sfarsit algoritm

    // Cautam un proces pentru care avem suficiente resurse pentru a-i
    // servi cererea maxima
    procesAles = -1;
    for(i=0; i<PROCESE; i++) {
```

```
      if(proceseImplicate[i] != 1)
        continue;

      gasit = 1;
      for(j=0; j<RESURSE; j++) {
        if(maxResursePerProces[i][j]-alocari[i][j] > resurseDisponibile[j]) {
          gasit = 0;
          break;
        }
      }
      if(gasit == 1) {
        procesAles = i;
        break;
      }
    }

    // Nici unul dintre procesele implicate in algoritm nu poate fi
    // servit cu maximul posibil, deci exista posibilitatea unui impas
    if(procesAles == -1)
      return 0; // sfarsit algoritm

    tiparesteStare(procesAles,proceseImplicate, resurseDisponibile, alocari);

    // Simulam faptul ca procesul procesAles ajunge sa se termine cu bine,
    // adica resursele ii sunt dealocariate si este marcat ca neimplicat in
    // algoritm
    proceseImplicate[procesAles] = 0;
    for(j=0; j<RESURSE; j++)
      alocari[procesAles][j] = 0;
  }
}

int main() {
  int cerere[RESURSE] = {1, 2, 2, 1};
  int solicitant = 1;

  switch(verificaCerere(solicitant, cerere)) {
    case -1:
      printf("\n\nEROARE. Procesul cere mai mult decat maximul declarat.\n");
      break;
    case  0:
      printf("\n\nCererea nu poate fi servita fara risc de impas. "
             "Procesul va fi pus in asteptare.\n");
      break;
    case  1:
      printf("\n\nCererea poate fi servita fara risc de impas.\n");
      break;
  }

  return 0;
}
```

Figura 9.17 Implementarea algoritmului bancherului

Pentru datele de test din figura 9.17 algoritmul decide că cererea poate fi servită, eliminând toate procesele în următoarea secvenţă: P1, P0, P2, P3, P4. Dacă însă în tabelul `resurseAlocatePerProces` se modfică alocarea resursei `R2` pentru procesul `P2` din 1 in 2, algoritmul nu va servi cererea din cauza apariţiei riscului de impas. Lăsăm pe seama cititorului să ruleze programul din fig. 9.17 şi să vadă rezultatele obţinute.

9.2 Conceptul de multiprogramare.

Cel mai important concept, introdus mai întâi la sistemele seriale şi preluat apoi la toate sistemele de calcul moderne, este conceptul de *multiprogramare*. El reprezintă modul de exploatare a unui sistem de calcul cu un singur procesor central, care presupune existenţa simultană în memoria internă a mai multor procese care se execută concurent. Multiprogramarea duce la o mai bună utilizare a procesorului şi a memoriei.

Primele sisteme de calcul ce lucrează în multiprogramare au apărut cu ceva mai înainte de anul 1965. Tehnologic, în aceeaşi perioadă au apărut plăcile cu circuite integrate. De acum se poate vorbi de generaţia a III-a de calculatoare.

Pe scurt, lucrul în multiprogramare se desfăşoară astfel: în fiecare moment procesorul execută o instrucţiune a unui proces. Vom spune că acest proces este în starea **RUN**. Restul proceselor, fie că aşteaptă apariţia unui eveniment extern (terminareaunei operaţii I/O, scurgerea unui interval de timp etc.), fie că sunt pregătite pentru a fi servite, în orice moment, de către procesor. Despre cele care aşteaptă un eveniment extern spunem că sunt în starea **WAIT**, iar cele care sunt pregătite de execuţie spunem că sunt în starea **READY**.

Pentru ca un sistem de operare să poată lucra în multiprogramare, este necesar ca:
Sistemul de calcul să dispună de un sistem de întreruperi pentru a semnala apariţia evenimentelor externe;
Sistemul de operare să gestioneze, să aloce şi să protejeze resursele între utlizatori. Prin resurse înţelegem: memoria, dispozitivele periferice, fişierele, timpul fizic etc.

9.2.1 Trecerea unui proces dintr-o stare într-alta

Trecerea unui proces din starea **RUN** în starea **WAIT** este (evident) comandată de către procesul însuşi; el (procesul) ştie, în funcţie de rezultatele obţinute până în prezent, când trebuie să urmeze o instrucţiune de I/O.

Trecerea proceselor din starea **RUN** în starea **READY** şi invers poate fi comandată sau de proces (cedare voluntară de procesor) sau poate fi comandată de către sistemul de operare (cedare involuntară de procesor).

Trecere unui proces dintr-o stare într-alta este făcută pe baza unui *algoritm de planificare* rulat de o componentă a sistemului de operare numită *planificator*. Planificatorul este o componentă centrală a oricărui sistem de operare modern.

9.2.1.1 Cedarea voluntară a procesorului

Această modalitate de cedare a procesorului se bazează în general pe programatorul care dezvoltă programul să introducă instrucţiuni de cedare în codul sursă. In alte situaţii sistemul de calcul în sine introduce instrucţiuni de cedare printre instrucţiunile binare ale programului. Un planificator care foloseşte această metode se numeşte *nepreemptiv*.

Cedarea voluntară a procesorului este o practică din vremurile de început ale sistememlor de operare. Deşi simplifică mult munca planificatorului, acest gen de cedare dă unui proces posibilitatea de a bloca întregului sistem de operare (o problemă foarte serioasă care se poate

remedia doar prin re-startarea sistemului de calcul). Pentru a bloca sistemul, un proces nu trebuie decât să ruleze un ciclu infinit în care să nu facă nici o cerere de resurse care să-l întrerupă şi nici să nu apeleze instrucţiuni de cedare a procesorului.

9.2.1.2 Cedarea involuntară a procesorului

Cedarea involuntară a procesorului presupune intervenţia planificatorului pentru a întrerupe procesul curent. Un astfel de planificator se numeşte *preemptiv*. Intreurperea procesului curent se face în general folosind o întrerupere activată de sistemul de calcul la intervale constante (în general) de timp. Procedura de servire a întreruperii este aceeaşi pentru toate procesele şi cheamă instrucţiunea de cedare a procesorului. In urma acestei cedări, planificatorul sistemului având o proritate absolută, preia procesorul (devine activ) şi decide care dintre procesele existente va primi procesorul.

9.2.2 Funcţionarea unui planificator

In fig. 9.18 este prezentat algoritmul după care funcţionează un planificator. In el se consideră că sunt n procese, iar priorităţile lor sunt numerele 1, 2, ..., ***n***, cu 1 prioritatea ca mai mică şi ***n*** prioritatea ca mai mare.

Fiecare proces este încărcat într-o *partiţie* de memorie. Fiecare partiţie are un *număr de prioritate*. In fig. 9.19 se prezintă o posibilă evoluţie a trei procese în regim de multiprogramare. Fiecare dintre cele trei procese este încărcat într-o partiţie proprie.

Procesului **P3**, are nevoie să efectueze până la terminare două operaţii I/O şi are cea mai mare prioritate dintre cele trei procese. Procesul **P2** are nevoie de trei operaţii I/O. Procesului **P1**, cel mai puţin prioritar, are nevoie să efectueze până la terminare o singură operaţie I/O. Presupunem că cele trei procese intră în sistem în acelaşi timp. Desenul ilustrează pe orizontală axa timpului, iar porţiunile în care este scris "**RUN**" indică faptul că procesorul execută succesiv instrucţiuni maşină ale procesului respectiv. In funcţie de momentele în timp în care cele trei procese solicită operaţii de I/O, fig. 9.19 ilustrează 12 momente eveniment, momente în care se schimbă starea unora dintre cele trei procese.

```
do {
  for(i=n; i>=1; i--) {
    if(<Pi este în starea RUN>) {
      EXECUTA(<urmatoarea instructiune masina din Pi>);
      continue;
    } else if(<Pi este în starea READY>) {
      // Exista, în mod sigur, cel putin un proces Pj cu i>j în starea RUN
      STARE(Pj) = READY;
      STARE(Pi) = RUN;
      EXECUTA(<urmatoarea instructiune masina din Pi>);
      continue;
    }
  }
} while(true);
```

Figura 9.18 Implementarea unui planificator

Observăm că **P1** din cauza priorităţii mici, reuşeste să intre în starea **RUN** numai când **P2** şi **P3** sunt în starea **WAIT** sau **FINISH** (procesul s-a terminat). Acestea sunt momentele 2, 6, 8 şi 11. Să mai observăm, de asemenea, că sistemul de calcul este bine exploatat, singura perioadă de timp în care procesorul este neîncărcat este între momentele 7 şi 8. Acesta este de fapt scopul principal al multi programării: valorificarea procesorului de către alt proces câtă vreme cel curent aşteaptă un eveniment extern.

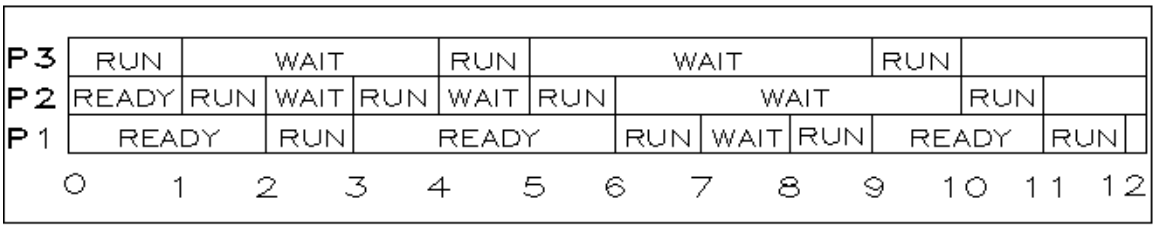

Figura 9.19 Exemplu de evoluţie în multiprogramare

Evoluţiile din fig. 9.19 descriu o situaţie particulară. Astfel, toate procesele sunt lansate în execuţie simultan, iar după terminarea lui **P**1 şi **P**2 nu se mai lansează nimic. In realitate, fiecare partiţie primeşte spre execuţie un nou proces (dacă există) după terminarea celui curent. Deci, spre exemplu, la momentul 10 în partiţia 1 se va lansa un nou proces.

9.3 Planificarea proceselor

9.3.1 Sarcinile planificatorului de procese.

Planificatorul de procese este responsabil pentru trecerea proceselor din starea **READY** în starea **RUN** şi invers. Pentru aceasta panificatorul trebuie să îndeplinească următoarele sub-sarcini:

- ţinerea evidenţei tuturor proceselor din sistem;
- alegerea procesului căruia i se va atribui procesorul şi pentru cât timp;
- alocarea procesului un procesor;
- eliberarea procesorului la ieşirea procesului din starea **RUN**.

Pentru fiecare proces din sistem planificatorul întreţine o strctură de date numită Process Control Block (**PCB**) în care trebuie să apară, printre altele, următoarele informaţii:

- starea procesului;
- pointer la următorul **PCB** în aceeaşi stare;
- numărul procesului (acordat de planificator);
- contorul de proces (adresa instrucţiunii maşină care urmează a fi executată);
- zonă pentru copia regiştrilor generali ai maşinii;
- limitele zonei (zonelor) de memorie alocate procesului;
- lista fişierelor deschise etc.

Toate structurile **PCB** sunt memorate într-o zonă de memorie proprie planificatorului de procese. Starea unui proces poate fi **RUN**, **READY**, **WAIT** sau **SWAP** (vezi 8.2.1.1). Procesele din aceeaşi stare sunt înlănţuite între ele.

Deoarece toate procesele folosesc regiştrii generali ai procesorului, este necesar ca la ieşirea din starea **RUN** conţinutul acestora să fie salvat. La o nouă intrare în **RUN** se reface conţinutul regiştrior. Spre exemplu, în fig. 9.20 este prezentat un schimb **READY** ↔ **RUN** între două procese **A** şi **B**. Evident, la trecerea din B în **A** operaţiile se inversează.

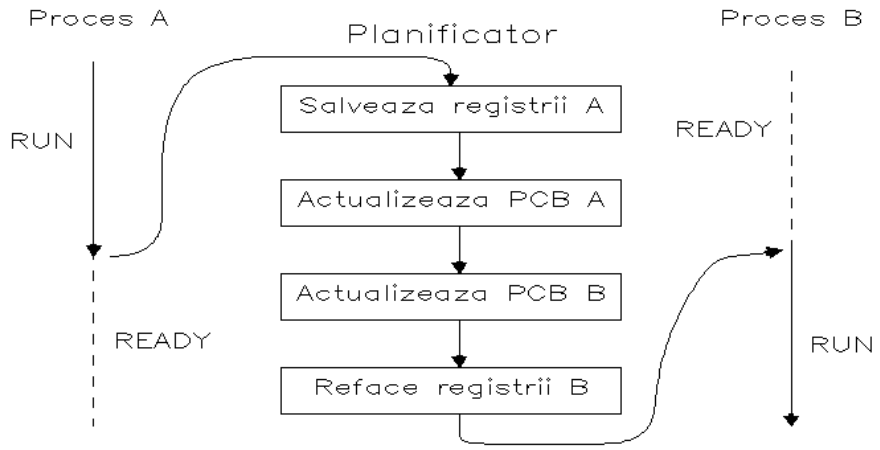

Figura 9.20 Schimbul RUN ↔ READY între două procese

In sarcina planificatorului de procese stă şi organizarea cozilor la perifericele de I/O. Deoarece mai multe procese pot solicita simultan accesul la un periferic, planificatorul întreţine cozi de aşteptare pentru toate perifericele. In fig. 9.21 este exemplificat un ansamblu de cozi proprii planificatorului de procese. Cozile **MM0**, **MM1**, **DKO** şi **TTO** sunt cozi de aşteptare asociate cu perifericele cu acelaşi nume. Coada **READY** conţine procese aflate în starea **READY** (pregătite pentru a intra în execuţie în orice moment). Coada **RUN** conţine procese aflate în exeuţie deja. Cum un singur proces poate fi în execuţie pe un procesor, se deduce că fig 9.21 se referă la un sistem de calcul bi-procesor.

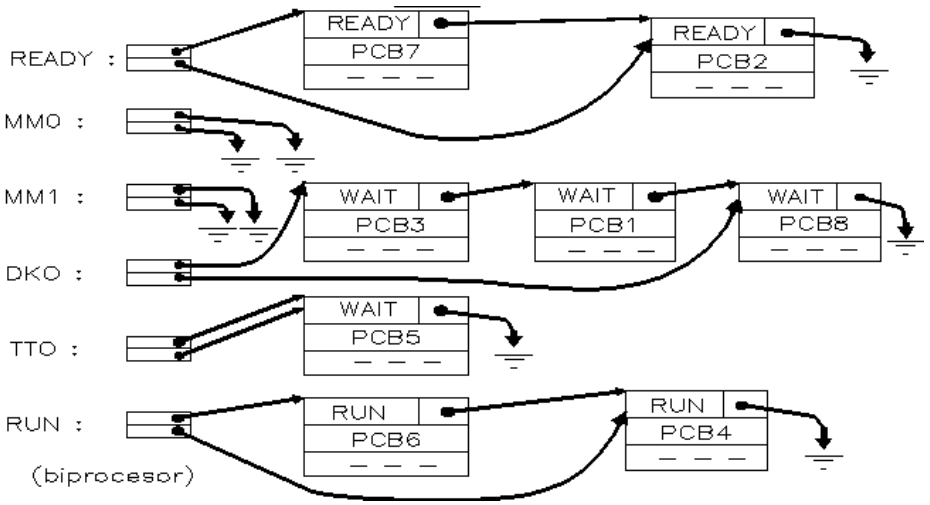

Figura 9.21 Cozi gestionate de planificatorul proceselor

9.3.2 Algoritmi de planificare.

O mare parte a literaturii a pus un accent pe algoritmii de planificare. Iată numai câteva lucrări de referinţă în domeniul sistemelor de operare, care tratează pe larg această problemă: [4], [10], [45], [49]. Motivele principale pentru care s-a dat importanţă mare planificării sunt că performanţa unui sistem de operare este influenţată decisiv de funcţionarea planificatorului şi nu în ultimul rând pentru că planificare se pretează uşor la modelarea matematică.

Alte lucrări, cum ar fi [22], [29], enumeră foarte pe scurt câţiva algoritmi, trecând repede peste acest capitol. Punctul de vedere al celor din urmă este justificat prin faptul că *ponderea planificării în sistemele de operare este destul de mică.* In cele ce urmează, enumerăm cei mai utilizaţi algoritmi de planificare a proceselor.

9.3.2.1 FCFS (First Come First Served)

Numele alogritmului însemna în traducere directă înseamnă *primul venit - primul servit*. Este cunoscut şi sub numele de *FIFO (First In First Out)*. Procesele sunt servite în ordinea lor cronologică. Este un algoritm simplu, dar nu prea eficient.

Spre exemplu, presupunem că există 3 procese cu următorii timpi de execuţie:

- procesul 1 durează 24 minute;
- procesul 2 durează 3 minute;
- procesul 3 durează 3 minute.

Dacă acestea sosesc la sistem în ordinea 1, 2, 3, atunci timpul mediu de servire este:

$$\frac{24 + 27 + 30}{3} = 27 \text{ minute}$$

Dacă sosesc în ordinea 2, 3, 1 timpul mediu de servire este:

$$\frac{3 + 6 + 30}{3} = 13 \text{ minute}$$

9.3.2.2 SJF (Shortest Job First).

Se execută primul, jobul (procesul) care consumă cel mai puţin timp procesor. Probabil că acest algoritm este cel mai bun, însă are un mare dezavantaj: presupune cunoaşterea exactă a timpului procesor necesar execuţiei procesului.

9.3.2.3 Algoritmul bazat pe priorităţi.

Este algoritmul cel mai des folosit. Toţi ceilalţi algoritmi descrişi în literatură sunt cazuri particulare ale acestuia. Fiecare proces primeşte un număr de prioritate. Procesele se ordonează după aceste priorităţi, apoi se execută în această ordine.

Se disting două metode de stabilire a priorităţilor:

1. procesul primeşte prioritatea la intrarea în sistem şi şi-o păstrează până la sfârşit;
2. Sistemul de operare calculează priorităţile după reguli proprii şi le ataşează, dinamic, proceselor în execuţie.

Varianta *1*) este folosită destul de des dar are dezavantajul că dacă apar multe procese cu prioritate mare, atunci cele cu prioritate mică aşteaptă practic indefinit. Acest fenomen este cunoscut sub numele de *starvation*. Pentru a evita acest fenomen, de obicei cele două metode sunt combinate. Astfel, unui proces cu prioritate mică i se va mări prioritatea cu cât aşteaptă mai mult şi se readuce la valoare iniţială după ce intră în execuţie.

9.3.2.4 Algoritm bazat pe termene de terminare (deadline scheduling)

Acest algoritm de planificare este folosit în principal la sistemele cu cerinţe de timp real foarte stricte. In asemenea sisteme este critic pentru un task sau proces să se termine la timp. Astfel fiecărui task i se ataşează un termen de terminare. Un caz special sunt task-urile repetitive cărora li se asociază termene de terminare asociate fiecărei iteraţii. Planificatorul foloseşte aceste termene pentru a decide când şi cărui task şi pentru cât timp să-i acorde procesorul în aşa fel încât să se temrine la timp. Evident pentru acest gen de planificare este esenţială cunoaşterea exactă a duratei fiecărui task. La apariţia unui nou task în sistem, planificatorul îl va accepta pentru execuţie doar dacă poate găsi o alocare prin care termenele noului task să fie respectate fără a le afecta pe cele ale task-urilor deja existente.

O strategie posibilă pentru calcularea secvenţei de execuţie a task-urilor este aducerea în stare **RUN** întotdeauna a task-ului cu termenul de terminare cel mai apropiat.

9.3.2.5 Round-Robin (planificare circulară).

Aceşti algoritmi sunt destinaţi în special sistemelor de operare care lucrează în timesharing. Pentru realizare, se defineşte o *cuantă de timp,* de obicei între 10-100 milisecunde. Coada READY a proceselor este tratată circular. Pe durata unei cuante se alocă procesorul unui proces. După epuizarea acestei cuante, procesul este trecut la sfârşitul cozii, al doilea proces este preluat de către procesor ş.a.m.d.

Există şi *variante de Round-Robin.* Spre exemplu, dacă un proces nu şi-a consumat în întregime cuanta (spre exemplu din cauza unei operaţii de I/O), atunci locul lui în coada **READY** este invers proporţional cu partea consumată din cuantă. De exemplu, dacă şi-a consumat toată cuanta, atunci procesul trece la sfârşitul cozii; dacă şi-a consumat numai jumătate din cuantă, atunci el trece la mijlocul cozii etc.

O altă variantă urmăreşte echilibrarea sistemului folosind aşa-zisul *Round-Robin cu reacţie*. Când un proces nou este acceptat, el se rulează mai întâi atâtea cuante câte au consumat celelalte procese, după care trece la planificarea Round-Robin simplă.

9.3.2.6 Algoritmul de cozi pe mai multe nivele

Acest algoritm se aplică atunci când lucrările pot fi clasificate uşor în grupe distincte. Spre exemplu, la un sistem de operare de tip mixt, interactiv şi serial, pot fi create 5 cozi distincte, înşirate mai jos, de la cea mai prioritară, la cea mai puţin prioritară:

Taskuri sistem.
Lucrările interactive.
Lucrări în care se editează texte.
Lucrări seriale obişnuite.
Lucrări seriale ale studenţilor.
...

10 Gestiunea memoriei

10.1 Structură; calculul de adresă

10.1.1 Problematica gestiunii memoriei

Pentru a fi executat, un program are nevoie de o anumită cantitate de memorie. Dacă se lucrează în multiprogramare este necesar ca în memorie să fie prezente simultan mai multe programe. Fiecare program foloseşte zona (zonele) de memorie alocată (alocate) lui, independent de eventuale alte programe active.

Se ştie că, în general, pe durata execuţiei unui program, necesarul de memorie variază. Aceast necesar variabil de memorie apare din două surse: folosirea *variabilelor dinamice* şi *segmentarea*. Variabilele dinamice sunt alocate de către proces, de regulă dintr-o zonă de memorie numită *heap* şi spaţiul este eliberat tot de către proces. Alocarea şi eliberarea se face prin perechi de funcţii `new – dispose`, `malloc – mfree`, `constructor – destructor` etc.

Segmentarea este o tehnică prin care un program executabil este decupat în entităţi distincte numite *segmente*. Segmentele pot fi: de cod, date sau de stivă. Pe durata vieţii programului unele dintre segmente pot fi prezente în memorie, altele nu. Programul însuşi poate cere încărcarea sau reîncărcarea unui segment, fie într-o zonă de memorie liberă, fie în locul altui (altor) segment(e). La construirea programului executabil se poate defini *structura de acoperire a segmentelor*. Spre exemplu, să presupunem că un program este segmentat şi are forma din fig. 10.1a. Atunci segmentele active la un moment dat pot fi: A, AC, AB, AF, ACE sau ACD, aşa cum reiese din fig.10.1b.

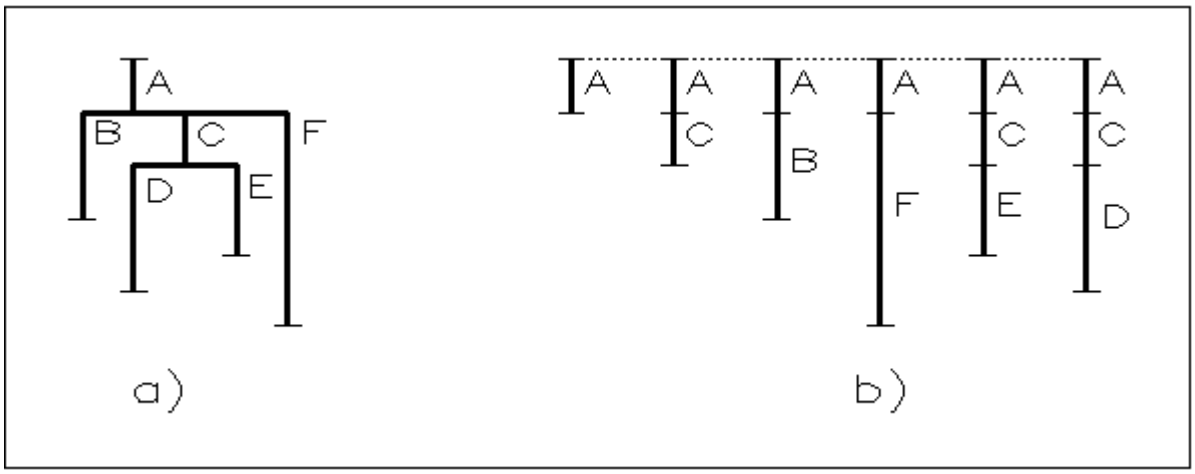

Figura 10.1 Instanţe posibile ale unui program segmentat

Evident, la fiecare din situaţiile precedente se solicită o altă cantitate de memorie.

Spaţiul de memorie principală al unui **SC**, adică ceea ce poate fi accesat în mod direct de către **CPU**, este în prezent încă *limitat*. Chiar dacă spaţiul de memorie oferit de actualele **SC** este mult mai mare decât cel oferit acum 20 de ani sau chiar 10 ani (iar preţul de cost al memoriei a scăzut drastic), limitarea totuşi rămâne. Din această cauză, **SO** cu sprijinul **SC** trebuie să gestioneze eficient folosirea acestui spaţiu.

Problemele privind gestiunea memoriei sunt rezolvate la nivelul inferior de către **SC**, extins eventual cu o componentă de *management al memoriei*. La nivelul superior, rezolvarea se face de către **SO** în colaborare cu **SC**. Principalele obiective ale gestiunii memoriei sunt:

- Calculul de translatare a adresei (relocare) ;
- Protecţia memoriei;
- Organizarea şi alocarea memoriei operative;
- Gestiunea memoriei secundare;
- Politici de schimb între proces, memoria operativă şi memoria secundară.

Vom aborda fiecare dintre aceste obiective, cu excepţia protecţiei memoriei. In general protecţia memoriei este puternic dependentă de hardware, motiv pentru care nu o tratăm aici, într-un cadru general. Fiecare pereche sistem de calcul – sistem de operare are propria politică de protecţie a memoriei, deci aceasta trebuie studiată în context concret.

10.1.2 Structura ierarhică de organizare a memoriei

In structura sa actuală, memoria unui sistem de calcul apare ca în fig. 10.2.

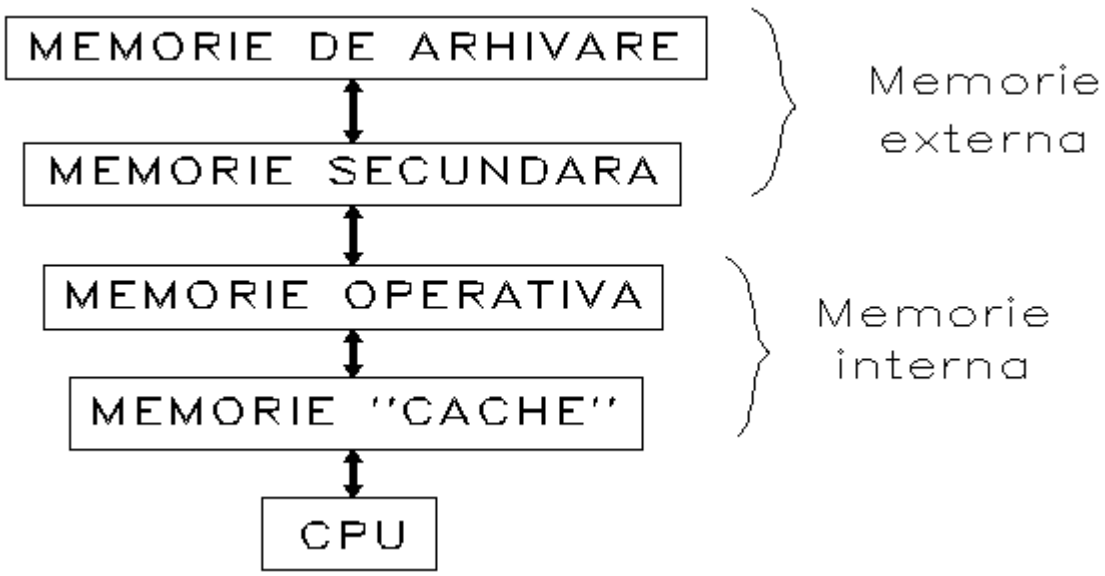

Figura 10.2 Structura memoriei unui sistem de calcul

Să prezentăm componentele de jos în sus.

Memeoria "cache" conţine informaţiile cele mai recent utilizate de către **CPU**. Ea are o capacitate relativ mică, dar cu timp de acces foarte rapid. La fiecare acces, **CPU** verifică dacă data invocată se află în memoria "cache" şi abia apoi solicită memoria operativă. Dacă este, atunci are loc schimbul între **CPU** şi ea. Dacă nu, atunci data este căutată în nivelele superioare. Aplicând *principiul vecinătăţii* [40], data respectivă este adusă din nivelul la care se află, *dar odată cu ea se aduce un număr de locaţii vecine ei*, astfel încât, împreună să umple memoria "cache". Principiul de lucru este cel al memoriilor tampon temporare, prezentat în 11.1. Intr-o secţiune viitoare vom prezenta în detaliu cum funcţionează memoria cache.

In ce constă principiul vecinătăţii? P. Denning afirmă, pe baza unor studii de simulare temeinice, că dacă la un moment dat se solicită o dată dintr-un anumit loc atunci solicitarea din momentul următor se va face, cu mare probabilitate, la o dată din apropierea precedentei.

Memoria "cache" este practic prezentă la toate sistemele de calcul moderne. De exemplu, un notebook actual, Intel Pentium 2GHz poate avea o memorie cache de până la 1Mo.

Memoria operativă conţine programele şi datele pentru toate procesele existente în sistem. In momentul în care un proces este terminat şi distrus, spaţiul de memorie operativă pe care l-a ocupat este eliberat şi va fi ocupat de alte procese. Capacitatea memoriei operative variază azi de la 128Mo până la 8 Go sau chiar mai mult. Viteza de acces este foarte mare, dar mai mică decât a memoriei "cache".

Memoria secundară apare la **SO** care deţin mecanisme de memorie virtuală. De asemenea, tot în cadrul memoriei secundare poate fi inclus spaţiul disc de swap (vezi 8.2.1.1). Această memorie este privită ca o extensie a memoriei operative. Suportul ei principal este discul magnetic. Accesul la această memorie este mult mai lent decât la cea operativă.

Memoria de arhivare este gestionată de utilizator şi constă din fişiere, baze de date ş.a. rezidente pe diferite suporturi magnetice (discuri, benzi, etc.).

Memoria "cache" şi memoria operativă formează ceea ce cunoaştem sub numele de *memoria internă*. Accesul **CPU** la acestea se face în mod direct. Pentru ca **CPU** să aibă acces la datele din memoria secundară şi de arhivare, acestea trebuie mai întâi mutate în memoria internă.

Din punct de vedere a performanţelor, privind fig. 10.2 de jos în sus se disting următoarele caracteristici ale componentelor memoriei:

- viteza de acces la memorie scade;
- preţul de cost pe unitatea de alocare scade;
- capacitatea de memorare creşte.

10.1.3 Mecanisme de translatare a adresei

Adresarea memoriei constă în realizarea legăturii între un obiect al programului şi adresa corespunzătoare din memoria operativă a **SC**. Pentru a explica mecanismele de translatare, să adoptăm câteva *notaţii*:

- ***OP*** notăm spaţiul de nume al obiectelor din programul sursă: nume de constante, de variabile, de etichete, de proceduri etc.
- ***AM*** este adresa relativă din cadrul unui modul compilat.
- ***AR*** este adresa relocabilă – adresă relativă din cadrul unui segment dintr-un fişier executabil.
- ***AF*** notăm mulţimea adreselor fizice din memoria operativă la care face referire programul în timpul execuţiei.

Calculul de adresă este modalitatea prin care se ajunge de la un obiect sursă din ***OP*** la adresa lui fizică din ***AF***. După cum se vede, acest calcul necesită trei faze, corespunzătoare fazelor în care se poate afla un program (vezi 8.2.1.2). Matematic vorbind, calculul de adresă se realizează prin compunerea a trei funcţii: ***c, l, t***, astfel:

$$OP \xrightarrow{c} AM \xrightarrow{l} AR \xrightarrow{t} AF$$

In fig. 10.3 sunt ilustrate fazele prin care trece un program componentele invocate şi etapele calculului de adresă, de la textul sursă până la execuţie.

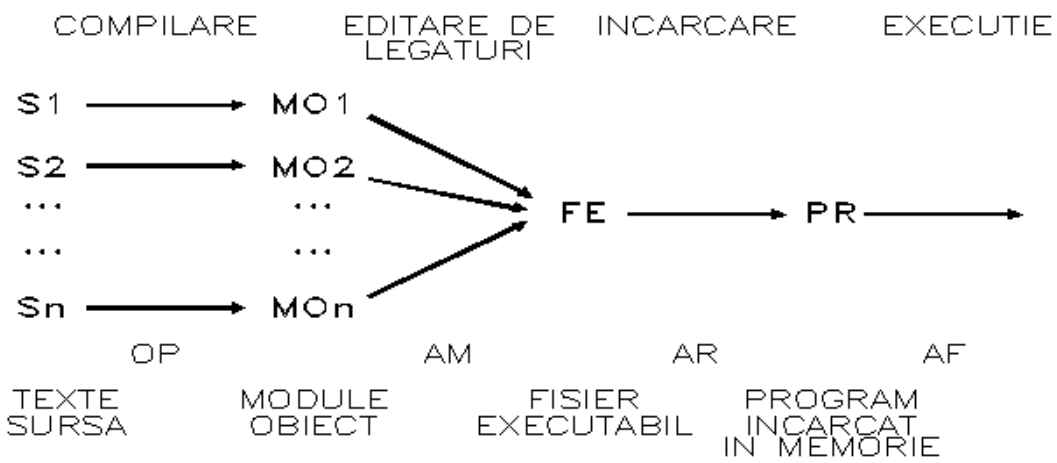

Figura 10.3 Fazele translatării unui program

10.1.3.1 Faza de compilare

Faza de *compilare* transformă un text sursă ***Si*** într-un *modul obiect* ***MO*i**. Corespunzător, numele obiectelor program sunt transformate în numere reprezentând ***AM***, adică adrese în cadrul modulului obiect. In cadrul fiecărui modul obiect aceste adrese încep de la 0. Deci prima funcţie din calculul de adresă:

c: *OP* ---> *AM*

este executată de către compilator sau asamblor. Modul ei de evaluare depinde de limbajul, de compilatorul şi de **SO** concret. Teoria compilării [33] are în vedere această evaluare, ea nu intră în scopurile noastre actuale. In fig. 10.4, prima parte, este prezentat un exemplu de aplicare a acestei funcţii.

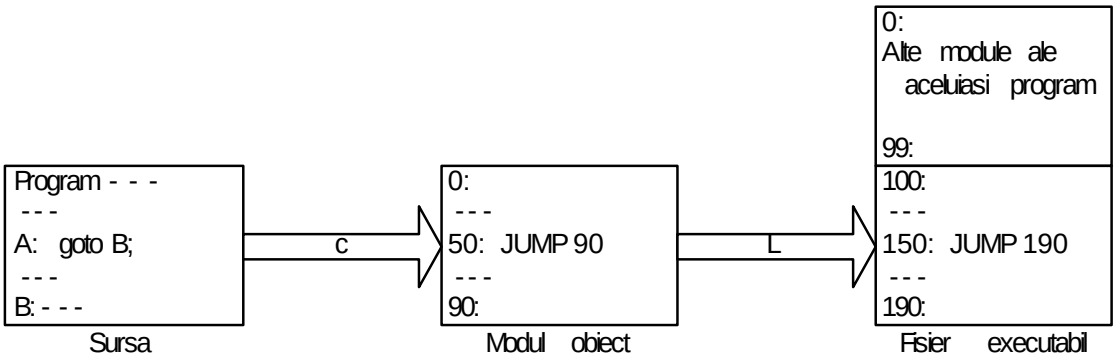

Figura 10.4 Translatare de la sursă la fişier executabil

10.1.3.2 Faza editării de legături

Faza de *editare de legături* grupează mai multe module formând un *fişier executabil*. Editorului de legături îi revine sarcina evaluării celei de-a doua funcţii de calcul a adresei. Această funcţie transformă adresele din cadrul modulelor în aşa zisele *adrese relocabile (relocatabile)*. Funcţia de legare este:

l: *AM* ---> *AR*

Particularităţile de evaluare a acestei funcţii sunt proprii editorului de legături. In fig. 10.4 sugerăm modul în care acţionează compunerea celor două funcţii.

10.1.3.3 Faza de încărcare şi execuţie

Funcţia

t: *AR* ---> *AF*

este *funcţia de translatare (relocare) a adresei.* In mod obişnuit ea este executată de către **CPU**. Translatarea depinde de tipul sistemului de calcul, în particular de existenţa dispozitivului de management al memoriei. Pentru a vedea principiul ei de lucru, presupunem că fişierul executabil conţine înregistrări succesive cu instrucţiuni de forma celei din fig. 10.5.

ARI CO A1 A2 - - - An

Figura 10.5 Formatul unei instrucţiuni maşină

Semnificaţiile câmpurilor instrucţiunii sunt:

- *ARI* este adresa relocabilă a instrucţiunii;
- *CO* este codul operaţiei,
- A1, ..., *A*n sunt argumentele instrucţiunii maşină: nume de regiştri, constante (argumente *imediate*), adrese relocabile din memorie

Presupunem că fiecare instrucţiune începe într-o locaţie de memorie şi că instrucţiunile sunt plasate una după cealaltă în locaţii succesive. Atunci încărcarea unui astfel de fişier se poate face folosind un încărcător analog lui LOADERABSOLUT, descris în 8.3 fig. 8.5.

Să presupunem că maşina dispune de un singur registru general pe care-l vom numi *A* (de la *registrul acumulator*). De asemenea, presupunem că între *A*1, ..., *An* există o singură adresă relocabilă. Evident că în realitate structura unui fişier executabil este mai complexă, dar pentru moment aceasta ne este suficientă.

Să adoptăm următoarele notaţii:

- *M*[0..*m*] conţine locaţiile memoriei operative;
- *pc* (Program Counter) indică adresa fizică a instrucţiunii care urmează a fi executată;
- *w* este conţinutul instrucţiunii curente;
- *Opcode*(w) este o funcţie care furnizează codul operaţiei din instrucţiunea curentă. Pentru fixarea ideilor, să presupunem că:
  - 1 este codul adunării;
  - 2 este codul operaţiei de memorare (depunere din *A*) într-o anumită locaţie;
  - 3 este codul instrucţiunii de salt necondiţionat;
  - ş.a.m.d.
- *Adress*(w) este o funcţie care furnizează valoarea adresei relocabile aflată între argumentele instrucţiunii curente.

Cu aceste notaţii, în fig. 10.6 este descris modul de funcţionare a **CPU** şi de acţiune a funcţiei de translatare *t*. De fapt, în fig. 10.6 este schiţat un *interpretor* al limbajului care are instrucţiuni de forma celei din fig. 10.5.

```
pc = t(adresa-de-start-a-programului);
do {
  w = M[pc];            // operaţia fetch
  co = Opcode(w);
  adr = Adress(w);
  pc = pc+1;
  switch (co) {
    1:  A:=A+M[t(adr)];  // adunare
    2:  M[t(adr)]:=A;    // memorare
    3:  pc:=t(adr);      // salt necondiţionat
    -  -  -
  }
 } while (false);
```

Figura 10.6 Funcţionarea CPU şi calculul funcţiei de translatare

10.2 Scheme simple de alocare a memoriei

10.2.1 Clasificarea tehnicilor de alocare

Problema alocării memoriei se pune în special la sistemele multiutilizator, motiv pentru care în continuare ne vom ocupa aproape exclusiv numai de aceste tipuri de sisteme. Tehnicile de alocare utilizate la diferite **SO** se împart în două mari categorii, fiecare categorie împărţindu-se la rândul ei în alte subcategorii, ca mai jos [19] [10].

- alocare reală:
  - la **SO** monoutilizator;
  - la **SO** multiutilizator:
    - cu partiţii fixe (statică):
      - absolută;
      - relocabilă;
    - cu partiţii variabile (dinamică);
- alocare virtuală:
    - paginată;
    - segmentată;
    - segmentată şi paginată.

10.2.2 Alocarea la sistemele monoutilizator

La sistemele monoutilizator este disponibil aproape întreg spaţiul de memorie. Gestiunea acestui spaţiu cade exclusiv în sarcina utilizatorului. El are la dispoziţie tehnici de *suprapunere (overlay)* pentru a-şi putea rula programele mari. In fig. 10.6 este ilustrat acest mod de lucru.

Porţiunea dintre adresele 0 şi *a*-1 este rezervată nucleului **SO**, care rămâne acolo de la încărcare şi până la oprirea sistemului. Intre adresele *c* şi *m*-1 (dacă memoria are capacitatea de *m* locaţii) este spaţiu nefolosit de către programul utilizator activ. Evident, adresa *c* variază de la un program utilizator la altul.

10.2.3 Alocarea cu partiţii fixe

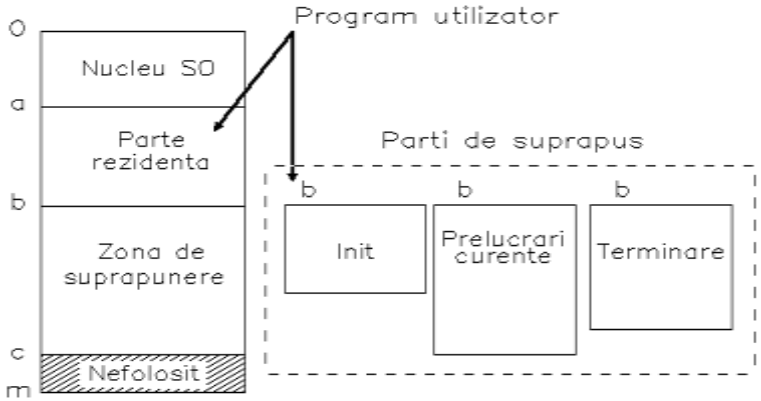

Figura 10.7 Alocarea memoriei la sistemele monoutilizator

Acest mod de alocare mai poartă numele de alocare statică sau alocare MFT - Memory Fix Tasks. El presupune decuparea memoriei în zone de lungime fixă numite *partiţii*. O partiţie este alocată unui proces pe toată durata execuţiei lui, indiferent dacă o ocupă complet sau nu. Un exemplu al acestui mod de alocare este descris în fig. 10.8. Zonele haşurate fac parte din partiţii, dar procesele active din ele nu le utilizează.

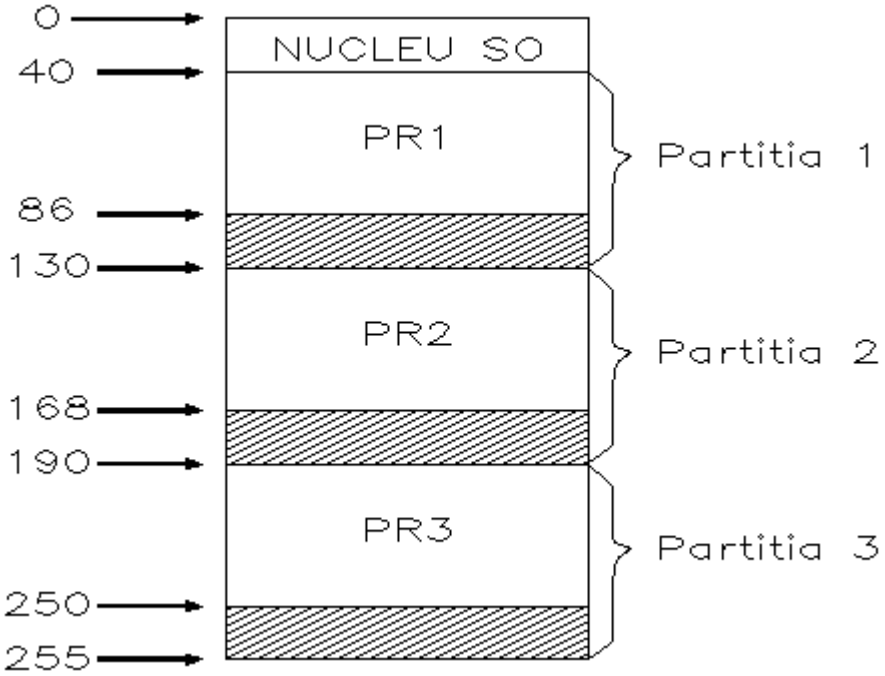

Figura 10.8 Exemplu de alocare cu partiţii fixe

Alocarea absolută se face pentru programe pregătite de editorul de legături pentru a fi rulate într-o zonă de memorie prestabilită şi numai acolo.

Mult mai folosită este **alocarea relocabilă**, la care adresarea în partiţie se face cu bază şi deplasament. La încărcarea unui program în memorie, în registrul lui de bază se pune adresa de început a partiţiei. In cazul sistemelor seriale cu multiprograme, dacă un proces este plasat spre execuţie într-o partiţie insuficientă, el este eliminat din sistem fără a fi executat.

De obicei, partiţiile au lungimi diferite. Una dintre problemele cele mai dificile este fixarea acestor dimensiuni. Dificultatea constă în faptul că nu se pot prevedea în viitor cantităţile de memorie pe care le vor solicita procesele încărcate în aceste partiţii. Alegerea unor dimensiuni

mai mari scade probabilitatea ca unele procese să nu poată fi executate, dar scade şi numărul proceselor active din sistem.

Acest mod de alocare este utilizat preponderent de către sistemele seriale. La fiecare partiţie există un şir de procese care aşteaptă să fie executate. Modul în care se organizează acest sistem de aşteptare poate influenţa performanţele de ansamblu ale sistemului şi poate eventual atenua efectul unei dimensionări defectuoase a partiţiilor. In general există două moduri de legare a proceselor la partiţii:

- *Fiecare partiţie are coada proprie*; operatorul stabileşte de la început care sunt procesele care vor fi executate în fiecare partiţie.
- *O singură coadă pentru toate partiţiile*; **SO** alege, pentru procesul care urmează să intre în lucru, în ce partiţie se va executa.

Legarea prin cozi proprii partiţiilor este mai simplă din punctul de vedere al **SO**. Primele sisteme multiutilizator au adoptat acest mod de legare. In schimb, legarea cu o singură coadă este mai avantajoasă, pentru faptul că se poate alege partiţia cea mai potrivită pentru plasarea unui proces.

10.2.4 Alocarea cu partiţii variabile

Acest mod de legare mai este cunoscut şi sub numele de alocare dinamică sau alocare MVT - Memory Variable Task. El reprezintă o extensie a alocării cu partiţii fixe, care permite o exploatare mai suplă şi mai economică a memoriei **SC**. *In funcţie de solicitările la sistem şi de capacitatea de memorie încă disponibilă la un moment dat, numărul şi dimensiunea partiţiilor se modifică automat.*

In fig. 10.9 sunt prezentate mai multe stări succesive ale memoriei, în acest mod de alocare.

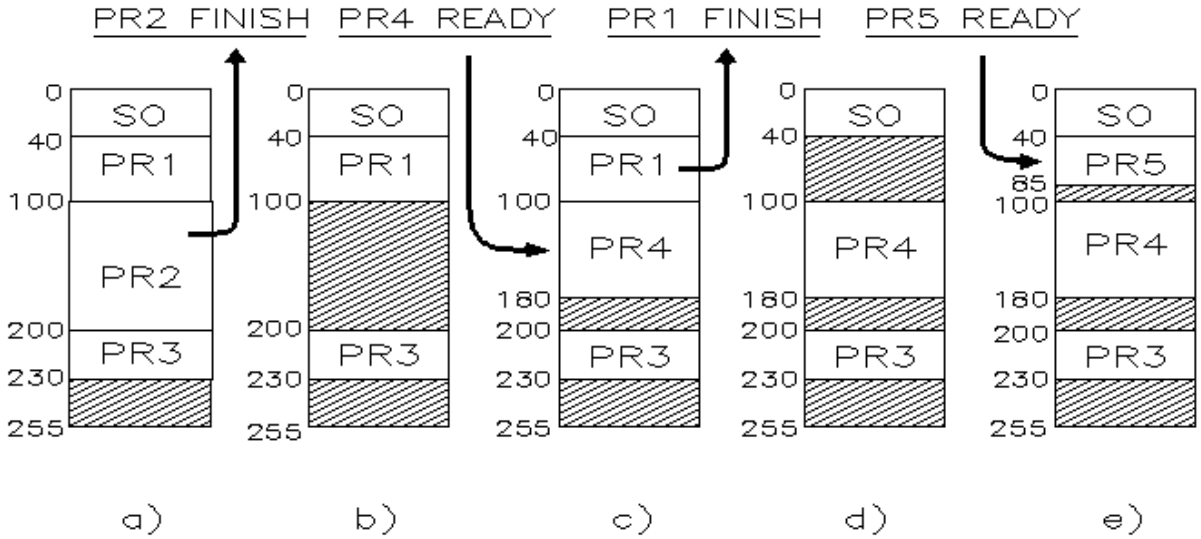

Figura 10.9 Evoluţia proceselor la alocarea cu partiţii variabile

In momentul în care procesul intră în sistem, el este plasat în memorie într-un spaţiu în care încape cea mai lungă ramură a sa. Spaţiul liber în care a intrat procesul este acum descompus în două partiţii: una în care se află procesul, iar cealaltă într-un spaţiu liber mai mic. Este uşor de observat că dacă sistemul funcţionează timp îndelungat, atunci numărul spaţiilor libere va creşte, iar dimensiunile lor vor scădea. Fenomenul este cunoscut sub numele de *fragmentarea*

internă a memoriei. După cum se va vedea, acest fenomen poate avea efecte neplăcute. In momentul în care un proces nu are spaţiu în care să se încarce, **SO** poate lua una din următoarele trei decizii:

a) *Procesul aşteaptă* până când i se eliberează o cantitate suficientă de memorie.
b) **SO** încearcă *alipirea unor spaţii libere vecine - colaţionare*, în speranţa că se va obţine un spaţiu de memorie suficient de mare. Spre exemplu, dacă momentul care urmează după cel din fig. 10.9d este terminarea procesului *PR*4, atunci în gestiunea sistemului apar trei zone libere adiacente: prima de 15Ko, a doua de 80Ko, iar a treia de 20Ko (vezi fig. 10.10). **SO** poate (nu întotdeauna o şi face în mod automat) să formeze din aceste trei spaţii unul singur de 115Ko, aşa cum se vede în fig. 10.10.
c) **SO** decide efectuarea unei operaţii de *compactare a memoriei (relocare)* adică de deplasare a partiţiilor active către partiţia monitor pentru a se absorbi toate "fragmentele" de memorie neutilizate. Este posibil ca spaţiul astfel obţinut să fie suficient pentru încărcarea procesului. In fig. 10.11b este dat un exemplu de compactare.

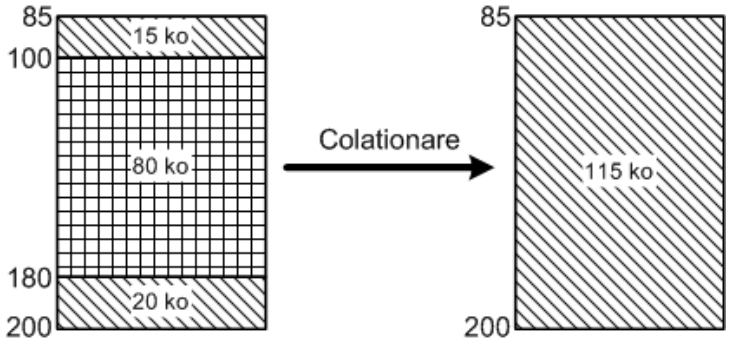

Figura 10.10 Colaţionarea de spaţii libere vecine

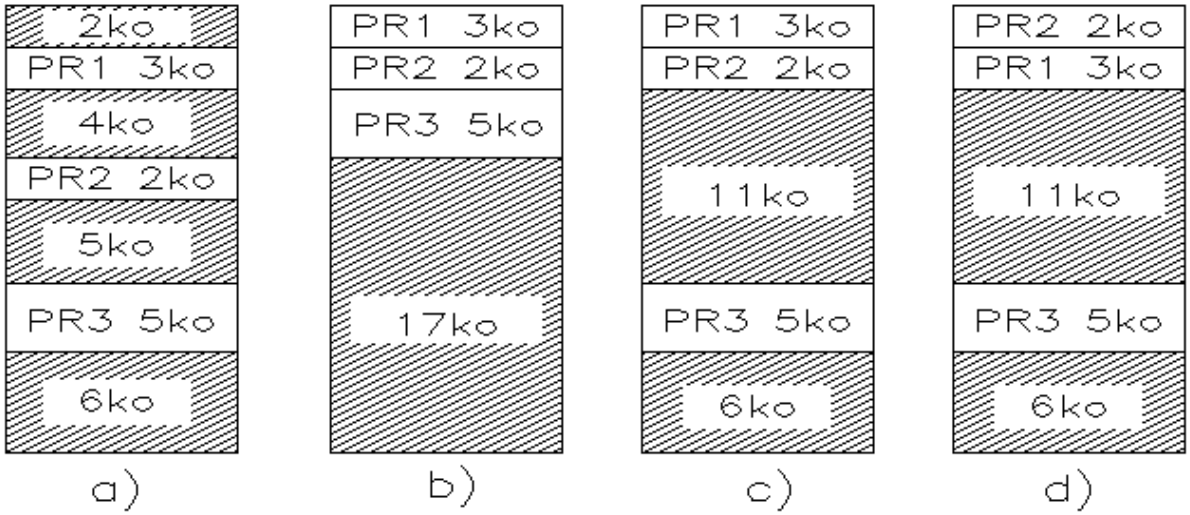

Figura 10.11 Posibilităţi de compactare prin relocare totală sau parţială

De regulă, compactarea este o operaţie costisitoare şi în practică se aleg soluţii de compromis, cum ar fi:

- Se lansează periodic compactarea (de exemplu la 10 secunde), indiferent de starea sistemului. In intervalul dintre compactări memoria apare ca un mozaic de spaţii ocupate care alternează cu spaţii libere. Procesele care nu au loc în memorie aşteaptă compactarea sau terminarea altui proces.

- Se realizează o compactare parţială pentru a asigura loc numai procesului care aşteaptă. Spre exemplu, dacă harta memoriei este cea din fig. 10.11a şi un proces cere 10Ko, se poate realiza numai compactarea parţială din fig. 10.11c.
- Se încearcă numai mutarea unora dintre procese cu colaţionarea spaţiilor rămase libere. Dacă reluăm exemplul de mai sus, este posibilă mutarea procesului *PR2* în primul spaţiu liber şi problema este rezolvată, harta fiind cea din fig. 10.11d.

Intre alocările de tip MFT şi MVT nu există practic diferenţe hard. Alocarea MVT este realizată de cele mai multe ori prin intermediul unor rutine specializate, eventual microprogramate, deci de către **SO**.

Alocarea MVT a fost utilizată mai întâi la **SC** IBM-360 sub **SO** OS-360 MVT, apoi la PDP 11/45.

10.3 Mecanisme de memorie virtuală

Termenul de *memorie virtuală* este de regulă asociat cu capacitatea de a adresa un spaţiu de memorie mai mare decât este cel disponibil la memoria operativă a **SC** concret. Conceptul este destul de vechi, el apărând odată cu **SO** ATLAS al Universităţii Manchester, Anglia, 1960 [19]. Se cunosc două metode de virtualizare, mult înrudite după cum vom vedea în continuare. Este vorba de alocarea paginată şi alocarea segmentată. Practic, toate sistemele de calcul actuale folosesc, într-o formă sau alta, mecanisme de memorie virtuală.

10.3.1 Alocarea paginată

Alocarea paginată a apărut la diverse **SC** pentru a evita fragmentarea excesivă, care apare la alocarea MVT, şi drept consecinţă, la evitarea aplicării relocării. Această alocare presupune cinci lucruri şi anume:

a) Instrucţiunile şi datele fiecărui program sunt împărţite în zone de lungime fixă, numite *pagini virtuale*. Fiecare ***AR*** (notaţiile din 10.1.3) aparţine unei pagini virtuale. Paginile virtuale se păstrează în memoria secundară.
b) Memoria operativă este împărţită în zone de lungime fixă, numite *pagini fizice*. Lungimea unei pagini fizice este fixată prin hard. Paginile virtuale şi cele reale au aceeaşi lungime, lungime care este o putere a lui 2, şi care este o constantă a sistemului (de exemplu 1Ko, 2Ko etc).
c) Fiecare ***AR*** este o pereche de forma: ***(p,d)***, unde ***p*** este numărul paginii virtuale, iar ***d*** adresa în cadrul paginii.
d) Fiecare ***AF*** este de forma ***(f,d),*** unde ***f*** este numărul paginii fizice, iar ***d*** adresa în cadrul paginii.
e) Calculul funcţiei de translatare **t : *AR*** ----> ***AF*** se face prin hard, conform schemei din fig. 10.12.

Dacă prin ***M[0..m]*** notăm memoria operativă, prin ***k*** puterea lui 2 (numărul de biţi) care dă lungimea unei pagini, prin ***TP*** adresa de start a tabelei de pagini, atunci algoritmul calculul funcţiei ***t*** este:

$$t(p, d) = M[TP + p] * 2 \wedge k + d$$

Acest calcul este valabil atunci când tabela de pagini ocupă un spaţiu în memoria operativă. Există însă **SC** ce dispun, hard, de o memorie specială de capacitate mică, numită *memorie asociativă.* Calitatea ei fundamentală este *adresarea prin conţinut*, ceea ce înseamnă că

găseşte locaţia care are un conţinut specificat, căutând simultan în toate locaţiile ei. Memoria asociativă conţine atâtea locaţii câte pagini fizice are. In fiecare locaţie a memoriei asociative este trecut numărul paginii virtuale care se află în pagina fizică având numărul de ordine identic cu numărul de ordine al locaţiei de memorie asociativă. Atunci când se dă un număr de pagină virtuală, se obţine automat numărul paginii fizice care o găzduieşte.

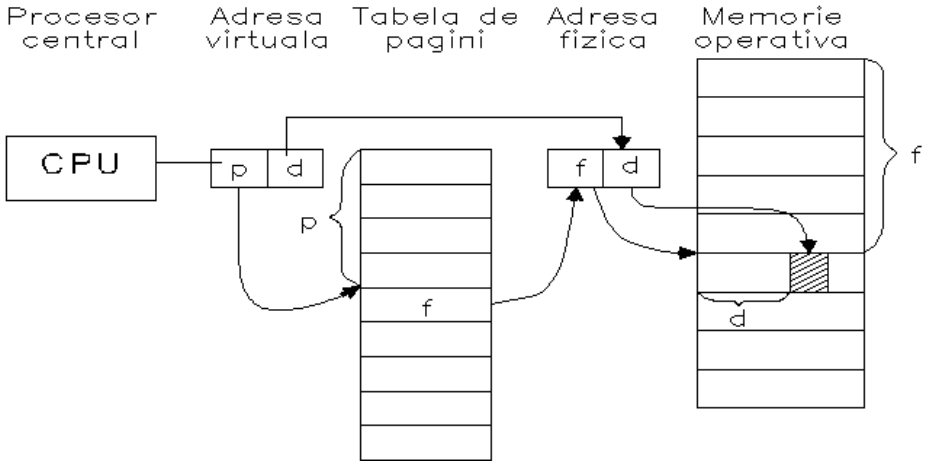

Figura 10.12 Translatarea unei pagini virtuale într-una fizică

Fiecare proces are propria lui tabelă de pagini, în care este trecută adresa fizică a paginii virtuale, dacă ea este prezentă în memoria operativă. La încărcarea unei noi pagini virtuale, aceasta se depune într-o pagină fizică liberă. Deci, în memoria operativă, paginile fizice sunt distribuite în general necontiguu, între mai multe procese. Spunem că are loc o *proiectare a spaţiului virtual peste cel real*. Este posibil, la un moment dat, să existe situaţia din fig. 10.13.

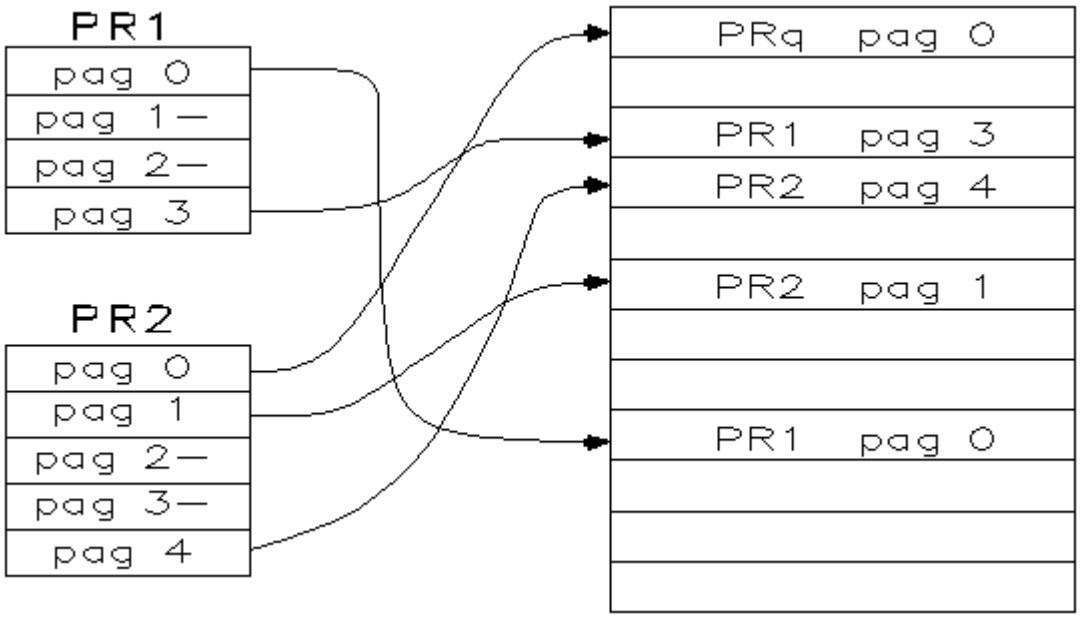

Figura 10.13 Două procese, într-o alocare paginată

Acest mecanism are avantajul că foloseşte mai eficient memoria operativă, fiecare program ocupând numai memoria strict necesară la un moment dat. Un avantaj colateral, dar nu de neglijat, este folosirea în comun a unei porţiuni de cod.

Să presupunem că avem un editor de texte al cărui cod (instrucţiuni pure, fără date) ocupă două pagini. Mai presupunem că fiecare utilizator consumă câte o pagină pentru datele proprii

de editat. Dacă sunt trei utilizatori, ei trebuie în mod normal să "consume" nouă pagini din memoria operativă. In fig. 10.14 se arată cum pot fi consumate numai cinci pagini.

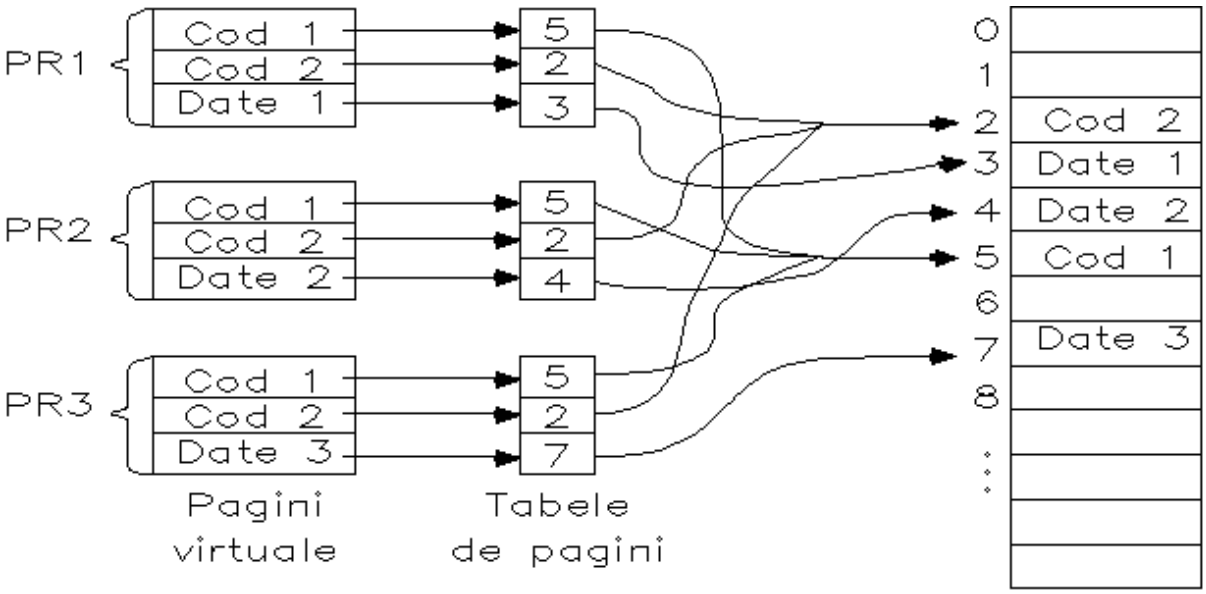

Figura 10.14 Folosirea în comun a unui cod

10.3.2 Alocare segmentată

Atunci când am vorbit despre alocarea reală, am văzut că fiecare proces trebuia să ocupe un spaţiu contiguu de memorie, numit partiţie. Ceea ce introduce nou mecanismul de alocare segmentată este faptul că textul unui program poate fi plasat în zone de memorie distincte, fiecare dintre ele conţinând o bucată de program numită *segment*. Singura deosebire principială dintre alocarea paginată şi cea segmentată este aceea că segmentele *sunt de lungimi diferite*. In fig. 10.15 am ilustrat această situaţie cu locurile ce vor fi ocupate de un program format din trei segmente.

In mod analog cu alocarea paginată, o adresă virtuală este o pereche ***(s,d),*** unde ***s*** este numărul segmentului, iar ***d*** este adresa în cadrul segmentului. Adresa reală (fizică) este o adresă obişnuită. Fiecare proces activ are o *tabelă de segmente*. Fiecare intrare în această tabelă conţine *adresa de început a segmentului*. Calculul de adresă se face analog celui de la alocarea paginată. Presupunem că la adresa **TS** se află începutul tabelei de segmente. Cu notaţiile obişnuite, funcţia ***t*** de translatare a adresei se calculează astfel:

$$t(s,d) = M[TS+s]+d$$

Pe lângă avantajul net faţă de alocările pe partiţii, alocarea segmentată mai prezintă încă două avantaje:

- Se pot crea *segmente reentrante*, - cod pur - care pot fi folosite în comun de către mai multe procese. Pentru aceasta este suficient ca toate procesele să aibă în tabelele lor aceeaşi adresă pentru segmentul pur. A se vedea aceeaşi problemă discutată la alocarea paginată.
- Se poate realiza o *foarte bună protecţie a memoriei*. Fiecare segment în parte poate primi alte drepturi de acces, drepturi trecute în tabela de segmente. La orice calcul de adresă se pot face şi astfel de verificări. Pentru detalii se pot consulta lucrările [10], [19].

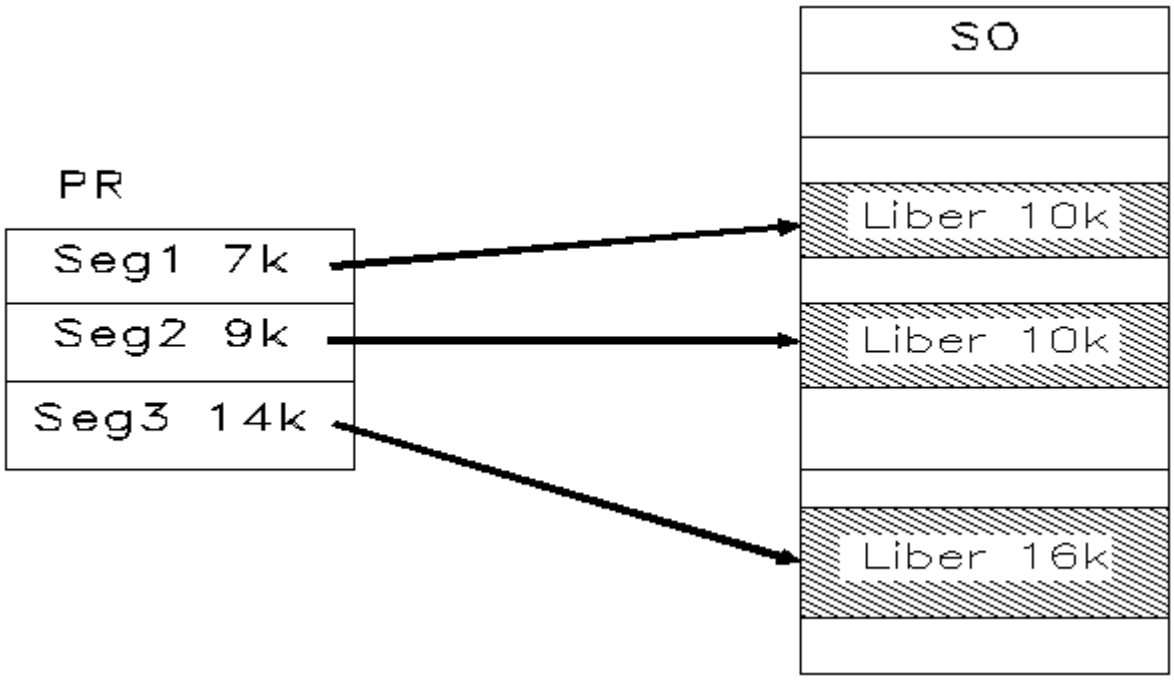

Figura 10.15 Alocare necontiguă prin segmentare

10.3.3 Alocare segmentată şi paginată

La alocarea segmentată am arătat că adresa fizică este una oarecare. Este deci posibil să apară fenomenul de fragmentare, despre care am vorbit la alocarea cu partiţii variabile (10.2.4). Ideea alocării segmentate şi paginate este aceea că alocarea spaţiului pentru fiecare segment să se facă paginat.

Pentru aceasta, mai întâi *fiecare proces* activ are *propria lui tabelă de segmente.* Apoi, *fiecare segment* dintre cele încărcate în memorie are *propria lui tabelă de pagini*. Fiecare intrare în tabela de segmente are un câmp rezervat adresei de început a tabelei de pagini proprii segmentului respectiv, aşa cum se vede fig. 10.16.

O adresă virtuală este de forma: ***(s, p, d),*** în care ***s*** este numărul segmentului***, p*** este numărul paginii virtuale în cadrul segmentului, iar ***d*** este deplasamentul în cadrul paginii. O adresă fizică este de forma: ***(f, d),*** unde ***f*** este numărul paginii fizice, iar ***d*** este deplasamentul în cadrul paginii.

Fie *k* constanta ce dă dimensiunea unei pagini (2^k), ***TS*** adresa de început a tabelei de segmente a unui proces şi presupunem că primul câmp al fiecărei intrări din tabela de segmente este pointerul spre tabela lui de pagini, atunci funcţia ***t*** de translatare se calculează astfel:

$$t(s,p,d) = M[M[TS+s]+p]*2k+d$$

Printre **SO** remarcabile care utilizează acest mod de alocare trebuie amintit în primul rând MULTICS, cel care a introdus de fapt acest mod de alocare. De asemenea, **SO** VAX/VMS adoptă acest mod de alocare.

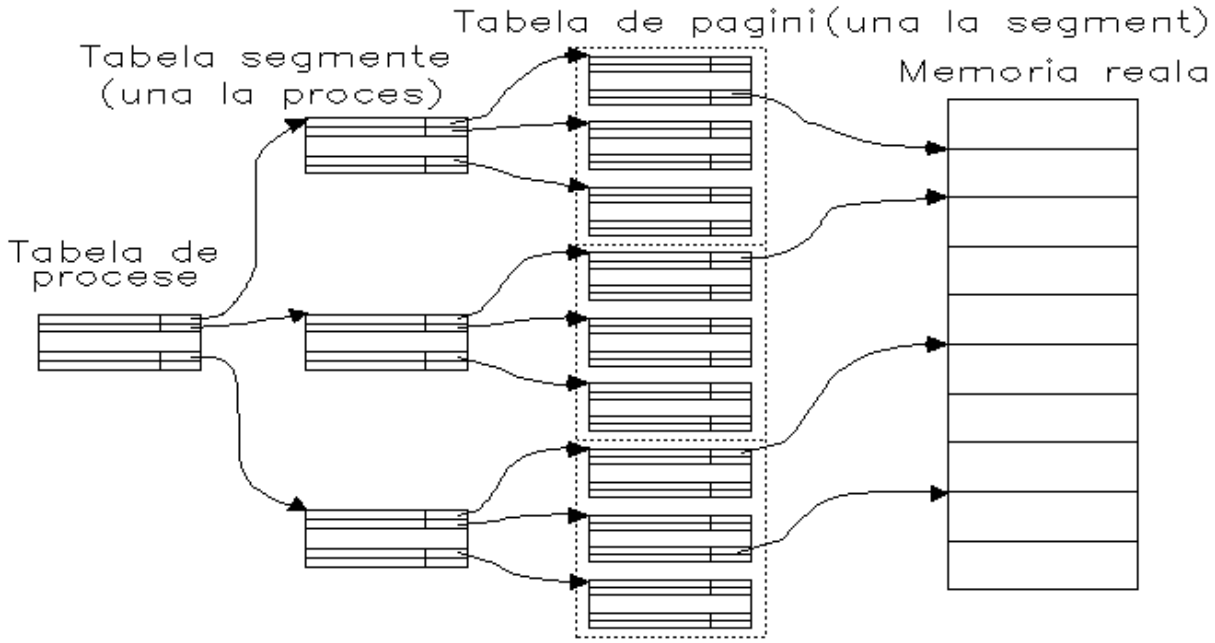

Figura 10.16 Alocare segmentată şi paginată

Calculatoarele IBM-PC cu microprocesor cel puţin 80386, dispun de un mecanism hard de gestiune paginată şi segmentată a memoriei extinse. Conceptual el funcţionează aşa cum am arătat mai sus, completat evident cu o serie de particularităţi legate de adresarea şi protecţia memoriei la acest tip de maşină.

10.4 Planificarea schimburilor cu memoria

10.4.1 Intrebările gestiunii memoriei şi politici de schimb.

Pe lângă mecanismele de alocare descrise în 10.2 şi 10.3, sistemul de operare trebuie să rezolve o serie de probleme care se pot ivi la modul de alocare respectiv. Rezolvarea acestor probleme înseamnă răspunsul la întrebările: CAT? UNDE? CAND? CARE?

Intrebarea "CAT?", apare atunci când se pune problema cantităţii de memorie alocată. La alocarea pe partiţii, se alocă la început toată cantitatea cerută; avem deci o *alocare statică*. La alocarea paginată avem o *alocare dinamică*, fiecare program consumă numai memoria necesară la un moment dat. In 10.3 am prezentat aceste tehnici.

Intrebarea "UNDE?", apare la alocarea cu partiţii variabile. Atunci când un program cere intrarea în sistem, trebuie luată decizia: dintre locurile goale pe care le poate ocupa, unde va fi plasat programul? Rezolvările posibile sunt cunoscute în literatură sub numele de *politici de plasare.* Acestea au o utilizare răspândită, depăşind cadrul sistemului de operare. Din acest motiv (cât şi datorită farmecului lor) le vom dedica o secţiune specială.

Intrebarea "CAND?", apare cel puţin în două situaţii: la sistemele cu paginare şi la alocarea cu partiţii variabile. In sistemele cu paginare, trebuie stabilit momentul în care o pagină virtuală este depusă într-o pagină fizică. In literatură, tehnicile de răspuns sunt cunoscute sub numele de *politici de încărcare (fetch).* Le vom dedica şi acestora o secţiune specială.

Aceeaşi întrebare "*CAND?*", apare pentru a decide momentele de *compactare (relocare)* a memoriei la alocarea cu partiţii variabile. In 10.2.4 am arătat în ce constă compactarea şi am dat trei posibilităţi de acţiune în acest sens.

Intrebarea "*CARE?*", apare la sistemele cu paginare. Să presupunem că la un moment dat toate paginile fizice sunt ocupate. Dacă un program mai cere încărcarea unei pagini, atunci este necesar ca una din paginile fizice să fie evacuată, în memoria secundară, pentru a i se face loc noii pagini. Această manevră poartă numele (după cum am mai spus) de *swapping*. Alegerea paginii care va fi înlocuită face obiectul unor *politici de înlocuire (replacement)*, cărora le vom dedica o secţiune separată.

Deşi, face parte dintre metodele implementate hard, deci nu implică deloc sistemul de operare, credem că este cazul să vedem puţin *cum funcţionează o memorie cache?* Vom dedica şi acesteia o secţiune specială.

10.4.2 Politici de plasare.

10.4.2.1 Metode de plasare şi structuri de date folosite

Nu numai sistemul de operare, ci foarte multe programe de aplicaţii solicită în timpul execuţiei lor diverse cantităţi de memorie. Vom presupune că întreaga cantitate de memorie solicitată la un moment dat *este formată dintr-un şir de octeţi consecutivi*. De asemenea, presupunem că există un *depozit de memorie* (numit *heap* în limbajele de programare) de unde se poate obţîne memorie liberă.

Rezolvarea cererilor presupune existenţa a două rutine. O primă rutină are sarcina de a *ocupa (aloca)* o zonă de memorie şi de a întoarce adresa ei de început. De exemplu, funcţia **malloc** din limbajul C şi operatorul **new** din limbajul C++ au acest rol. O a doua rutină are rolul de a *elibera* spaţiul alocat anterior, în vederea refolosirii lui.

Analogia dintre cele spuse mai sus şi alocarea cu partiţii variabile este mai mult decât evidentă. Problema politicilor de plasare nu lipseşte practic din nici un curs de sisteme de operare. Dintre lucrările mai consistente în acest domeniu amintim [4], [10], [22], [49].

Dată fiind importanţa acestor politici, vom da câteva metode de organizare a spaţiului de memorie şi algoritmii de ocupare şi alocare corespunzători. Pe lângă cerinţele de funcţionalitate, este bine ca plasarea succesivă a programelor (zonelor alocate) să împiedice, pe cât posibil, fragmentarea excesivă a memoriei operative. Altfel, este posibil ca cererile de memorie mai mari să nu poată fi servite deşi sistemul dispune per total de memoria necesară.

Dintre metodele de plasare, cele mai răspândite sunt următoarele patru:

- Metoda primei potriviri (First-fit).
- Metoda celei mai bune potriviri (Best-fit).
- Metoda celei mai rele potriviri (Worst-fit).
- Metoda alocării prin camarazi (Buddy-system).

Vom analiza pe rând fiecare dintre ele. Primele trei sunt foarte asemănătoare, iar ultima este oarecum deosebită. Pentru început ne vom ocupa de primele trei.

Dată fiind fragmentarea inerentă a memoriei cea mai convenabilă structură de date pentru regăsirea zonelor libere este *lista înlănţuită.* Fiecare nod al listei va descrie o zonă de memorie liberă specificându-i adresa de început, lungimea şi adresa nodului următor. Cum însă această listă înlanţuită trebuie şi ea să fie stocată în undeva memorie, cea mai convenabilă soluţie este ca fiecare nod să fie stocat la începutul zonei de memorie pe care o descrie. In această abordare, un nod va conţine doar lungimea zonei de memorie şi adresa următoarei zone libere (adresa următorului nod). Un nod nu va mai conţine adresa de început a zonei la care se referă pentru că este implicită prin adresa lui de memorie. Un nod al acestei liste înlănţuite se numeşte în literatura de specialitate *cuvânt de control*.

Vom prezenta în continuare câţiva algoritmi de plasare a cererilor de memorie. Pentru a trata unitar alocarea şi eliberarea octeţilor, vom adopta următoarea convenţie. Fiecare zonă liberă sau ocupată de memori începe cu un cuvânt de control. Câmpul **lung** (aflat în a doua jumătate a cuvântului de control) al lui indică numărul de octeţi liberi de *după* cuvântul de control. Pentru zonele libere de memorie pointerul **next** (aflat în prima jumătate a cuvântului de control) indică următoarea zonă liberă. Pentru zonele de memorie ocupate acest pointer nu este folosit. In fig. 10.17, 10.18 şi 10.19, cuvintele de control sunt subliniate cu linii îngroşate, iar pointerii la zone sunt indicaţi prin săgeţi. De asemenea, lungimea cuvântului de control este notată cu ***c*** şi este o constantă specifică sistemului de operare. După cum vom vedea apar situaţii în care numărul de octeţi alocaţi depăşeşte (cu maximum ***c***) numărul de octeţi solicitaţi. O zona ocupată este reperată *după* cuvântul ei de control.

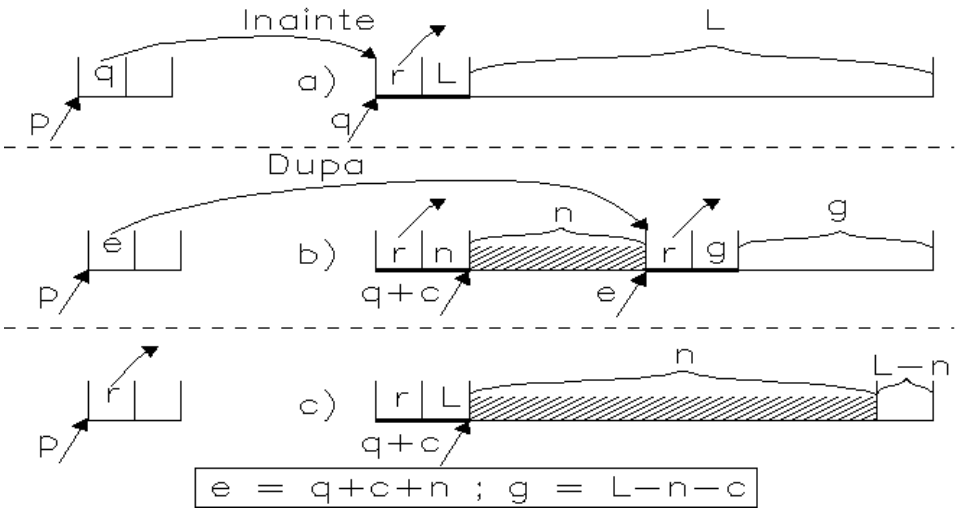

Figura 10.17 Alocarea de octeţi într-o zonă liberă

In fig. 10.17a este prezentată o zonă liberă al cărei cuvânt de control începe la adresa ***q*** şi are ***L*** octeţi liberi. Presupunem că se cere alocarea de ***n*** octeţi şi că ***L>n***. Ca rezultat al alocării, va apare una dintre situaţiile din fig. 10.17b sau 10.17c. Zona haşurată reprezintă octeţii ceruţi pentru alocare. Situaţia din fig. 10.17b apare dacă ***L-n > c+1***, adică spaţiul rămas în zonă permite crearea unei zone libere de cel puţin un octet. In cazul în care această condiţie nu este satisfăcută, se alocă întreaga zonă şi ultimii *L-n* octeţi rămân nefolosiţi fig. 10.17c).

Inainte de a descrie procedura inversă (de eliberare), să ne ocupăm de problema *comasării a două zone libere adiacente*. Pentru simplificare, vom presupune că lista zonelor libere este păstrată în ordinea crescătoare a adreselor. Situaţia în care este posibilă concatenarea a două zone libere este ilustrată în fig. 10.18.

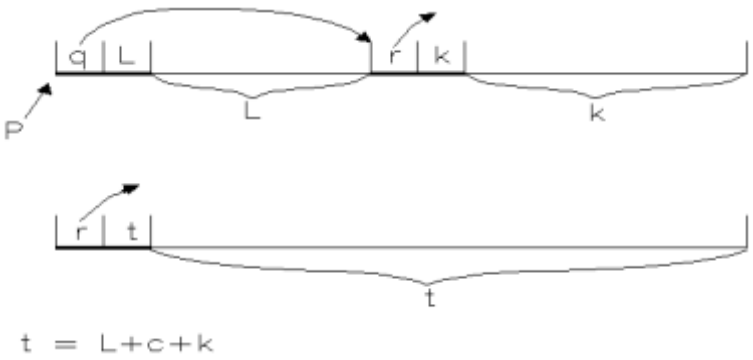

Figura 10.18 Comasarea a două zone libere adiacente

In fig. 10.19 este ilustrat un exemplu de eliberare a unei zone. La eliberare, partiţia devenită liberă se repune în lista de spaţii libere. De asemenea, se verifică dacă nu cumva partiţia proaspăt eliberată *poate fi comasată cu una vecină* din dreapta sau din stânga ei. Verificând sistematic, la fiecare eliberare, dacă este posibilă concatenarea, *nu vor exista două zone libere adiacente*.

In fig. 10.19a este prezentată situaţia înainte de eliberarea zonei. In fig. 10.19b se prezintă situaţia în care nu este posibilă nici un fel de comasare. Posibilitatea de *comasare la stânga* are loc dacă:

$$p+2*c = q$$

Ca rezultat se obţine zona din fig. 10.19c. O *comasare la dreapta* are loc dacă:

$$q+n = r$$

Ca efect se obţine zona din fig. 10.19d.

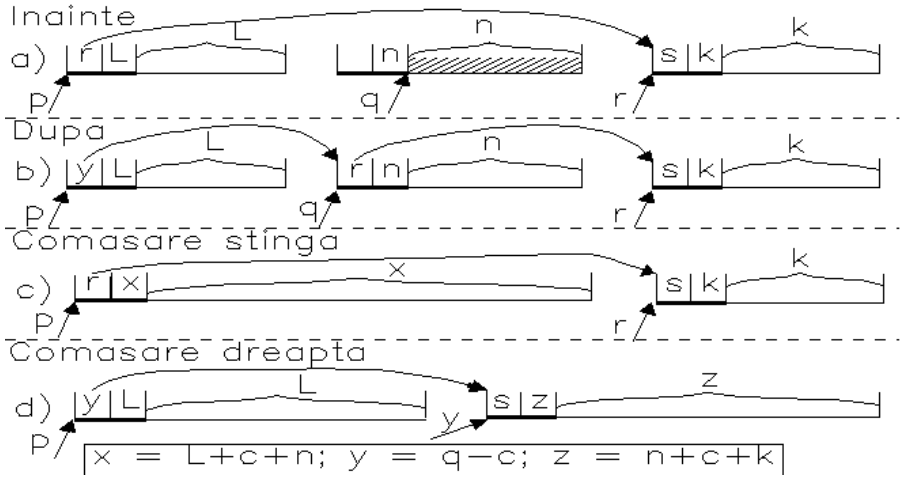

Figura 10.19 Eliberarea unei zone ocupate

Acum este momentul să prezentăm cele patru politici de plasare. Primele trei le vom detalia utilizând descrierile procedurilor de mai sus. Punctul de pornire al fiecărui astfel de algoritm este faptul ca în momentul pornirii sistemului, se alocă întreaga zonă de memorie sub forma unei singure zone libere.

10.4.2.2 Metoda primei potriviri (First-fit).

Esenţa metodei constă în aceea că partiţia solicitată este alocată în prima zonă liberă, în care începe. Principalul avantaj al metodei este simplitatea căutării de spaţiu liber. Pentru această metodă, structura listei de spaţii libere prezentate mai sus este cea mai adecvată.

10.4.2.3 Metoda celei mai bune potriviri (Best-fit).

Esenţa metodei constă în căutarea acelei zone libere, care lasă după alocare cel mai puţin spaţiu liber. Metoda Best-Fit a fost larg utilizată mulţi ani. Ea pare a fi destul de bună, deoarece economiseşte zonele de memorie mai mari astfel încât dacă ulterior va fi nevoie de ele vor fi disponibile. Există însă şi obiecţii, dintre care amintim două: timpul suplimentar de căutare şi proliferarea blocurilor libere de lungime mică, adică *fragmentarea internă excesivă* (vezi 10.2.4).

Primul neajuns este eliminat parţial dacă lista de spaţii libere se păstrează, nu în ordinea crescătoare a adreselor, ci *în ordinea crescătoare a lungimilor spaţiilor libere*. Din păcate, în acest caz problema comasării zonelor libere adiacente se complică foarte mult.

10.4.2.4 Metoda celei mai rele potriviri (Worst-fit).

Metoda Worst-Fit este oarecum dual㠹 metodei Best-Fit. Esenţa ei constă în căutarea acelei zone libere care lasă după alocare cel mai mult spaţiu liber.

Deşi, numele ei sugerează că este vorba de o metodă slabă, în realitate nu este chiar aşa. Faptul, că după alocare rămâne un spaţiu liber mare, este benefic, deoarece în spaţiul rămas poate fi plasată, în viitor, o altă partiţie. Fragmentarea internă probabil că nu evoluează prea rapid, însă timpul de căutare este mai mare decât cel de la metoda primei potriviri.

Si în acest caz este posibil ca lista de spaţii libere să se păstreze nu în ordinea crescătoare a adreselor, ci *în ordinea descrescătoare a lungimilor spaţiilor libere*. Din păcate, în acest caz problema comasării zonelor libere adiacente se complică, de asemenea, foarte mult.

10.4.2.5 Metoda alocării prin camarazi (Buddy-system).

Metoda alocării prin camarazi (Buddy) este deosebit de interesantă. Ea exploatează reprezentarea binară a adreselor şi faptul că din raţiuni tehnologice, dimensiunea memoriei interne este un multiplu al unei puteri a lui doi. Fie **c*2^n** dimensiunea memoriei interne. De exemplu, memoria unui IBM PC XT (unul dintre primele calculatoare personale) era 640 Ko, adică ***10*2^16*** octeţi.

Să notăm cu ***n*** cea mai mare putere a lui 2 prin care se poate exprima dimensiunea memoriei interne. In exemplul de mai sus ***n = 16***. Din raţiuni practice, se stabileşte ca unitate de alocare a memoriei tot o putere a lui 2. Fie ***m*** această putere a lui 2. Pentru exemplul de mai sus să considerăm că unitatea de alocare este **256** octeţi, adică **2^8** .

La sistemele Buddy, dimensiunile spaţiilor ocupate şi a celor libere sunt de forma ***2^k*** , unde ***m*** **<=** **k** **<=** ***n***. Ideea fundamentală este de a păstra liste separate de spaţii disponibile pentru

fiecare dimensiune **2^k** dintre cele de mai sus. Vor exista astfel ***n-m*+1** liste de spaţii disponibile. In exemplul de mai sus, vom avea **9** liste: lista de ordin **8** având dimensiunea unui spaţiu de **256** octeţi, lista de ordin **9** cu spaţii de dimensiune **512** etc. Ultima listă va fi de ordinul **16** şi poate avea maximum **10** spaţii a câte **65536 (2^16)** octeţi fiecare.

Prin definiţie, fiecare spaţiu liber sau ocupat de dimensiune **2^k** are adresa de început un multiplu de **2^k** .

Tot prin definiţie, două spaţii libere de ordinul ***k*** se numesc *camarazi (Buddy)* de ordin ***k***, dacă adresele lor **A1** şi **A2** verifică:

$$A1 < A2 , \ A2 = A1 + 2^k \text{ şi } A1 \bmod 2^{(k+1)} = 0$$

sau

$$A2 < A1 , \ A1 = A2 + 2^k \text{ şi } A2 \bmod 2^{(k+1)} = 0$$

Atunci când într-o listă de ordin ***k*** apar doi camarazi, sistemul îi concatenează într-un spaţiu de dimensiune **2^(k+1).**

Alocarea într-un sistem Buddy se desfăşoară astfel:

- Se determină cel mai mic număr ***p***, cu ***m* <= *p* <= *n***, pentru care numărul ***o*** de octeţi solicitaţi verifică: ***o* <= 2^p**
- Se caută, în această ordine, în listele de ordin ***p*, *p*+1, *p*+2, ..., *n*** o zonă liberă de dimensiune cel puţin ***o***.
- Dacă se găseşte o zonă de ordin ***p***, atunci aceasta este alocată şi se şterge din lista de ordinul ***p***.
- Dacă se găseşte o zonă de ordin ***k*>*p***, atunci se alocă primii **2^p** octeţi, se şterge zona din lista de ordin *k* şi se crează în schimb alte ***k*-*p*** zone libere, având dimensiunile:

  2^p, 2^(p+1) , ... , 2^(k-1)

In fig. 10.20 este dat un astfel de exemplu. Se doreşte alocarea a **1000** octeţi, deci ***p* = 10**. Nu s-au găsit zone libere nici de dimensiune **2^10** , nici **2^11** şi nici **2^12** . Prima zonă liberă de dimensiune **2^13** are adresa de început **5*2^13** şi am notat-o cu ***I*** în fig. 10.20. Ca rezultat al alocării a fost ocupată zona ***A*** de dimensiune **2^10** şi au fost create încă trei zone libere: ***B*** de dimensiune **2^10** , ***C*** de dimensiune **2^11** şi ***D*** de dimensiune **2^12** . Zonele ***B, C, D*** se trec respectiv în listele de ordine **10**, **11** şi **12**, iar zona ***I*** se şterge din lista de ordin **13**.

Eliberarea într-un sistem Buddy a unei zone de dimensiune **2^p** este un proces invers alocării. Astfel:

1. Se introduce zona respectivă în lista de ordin ***p***.
2. Se verifică dacă zona eliberată are un camarad de ordin ***p***. Dacă da, atunci zona este comasată cu acest camarad şi formează împreună o zonă liberă de dimensiune **2^(p+1)**. Atât zona eliberată cât şi camaradul ei se şterg din lista de ordinul ***p***, iar zona nou apărută se trece în lista de ordin ***p*+1**.
3. Se execută pasul 2 în mod repetat, mărind de fiecare dată ***p*** cu o unitate, până când nu se mai pot face comasări.

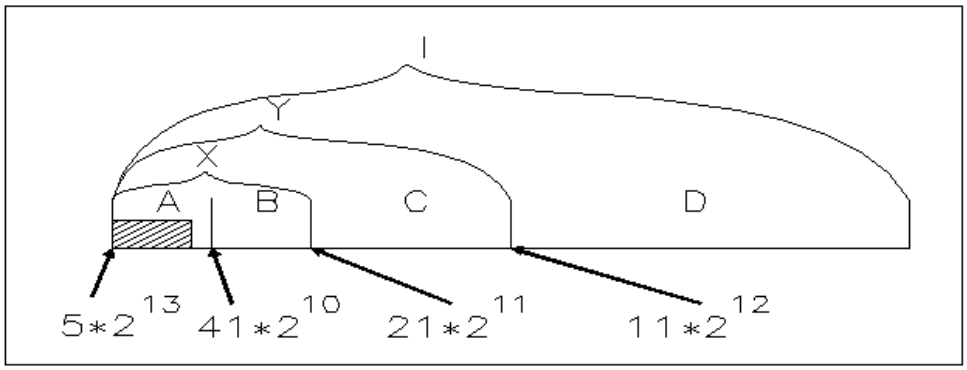

Figura 10.20 Alocare în sistem Buddy

De exemplu, să presupunem că în fig. 10.20 sunt libere la un moment dat zonele ***A, C, D,*** iar zona ***B*** este ocupată. La momentul următor, se eliberează şi zona ***B***. In conformitate cu paşii descrişi mai sus, se execută următoarele acţiuni:

- Se trece zona ***B*** în lista de ordin **10**.
- Se depistează că zonele ***A*** şi ***B*** sunt camarazi. Drept urmare, cele două zone sunt comasate şi fomează o nouă zonă ***X***. Zona ***X*** se trece în lista de ordin **11**, iar zonele ***A*** şi ***B*** se şterg din lista de ordin **10**.
- Se depistează că zonele ***X*** şi ***C*** sunt camarazi. Drept urmare, ele sunt comasate şi formează o zonă ***Y*** care se trece în lista de ordin **12**, înlocuind zonele ***X*** şi ***C*** din lista de ordin **11**.
- In sfârşit, se depistează că ***Y*** şi ***D*** sunt camarazi. Ele sunt şterse din lista de ordin **12**, iar în lista de ordin **13** se introduce rezultatul comasării lor.

Alocarea Buddy are multe avantaje. Unul dintre ele, deloc de neglijat, se referă la manipularea comodă a adreselor de zone. Se ştie că un număr binar este multiplu de **2** dacă el se termină cu ***k*** zerouri binare. Rezultă deci că adresele a doi camarazi de ordin ***k*** diferă doar prin bitul de pe poziţia ***k*** (numerotarea începând cu 0) şi ambele se termină prin ***k*** zerouri. Spre exemplu, adresele zonelor ***A*** şi ***B*** din fig. 10.20 sunt:

```
A:  10100000000000000
B:  10100100000000000
          ^
```

Evident, astfel de teste se fac deosebit de uşor folosind instrucţiunile maşină care operează asupra biţilor.

Până în prezent nu s-a găsit un criteriu solid de comparare a acestor patru metode de plasare. Compararea lor se face empiric, eventual prin simulare. De multe ori, se adoptă o metodă mai simplă, poate şi numai din raţiuni tehnologice. In orice caz, trebuie necondiţionat să se aibă în vedere şi concluziile, mai mult sau mai puţin empirice, rezultate din experienţă. Una dintre acestea spune că nu întotdeauna o metodă mai sofisticată este şi mai bună!

10.4.3 Politici de încărcare.

La sistemele cu alocare paginată, în momentul lansării în execuţie a programului acesta nu are nici o pagină în memorie. La prima solicitare a programului, sistemul de operare îi va aduce în memorie numai pagina solicitată. Dacă este vorba de un program mare, acesta va funcţiona normal un timp, după care va cere din nou o pagină care nu este în memorie etc. Intrebarea

care se pune este: *când* să se aducă o anumită pagină în memorie, pentru ca cererile de pagini să se reducă?

O soluţie simplă, dar evident ineficientă, este *încărcarea la început a tuturor paginilor*. Prin aceasta va dispare însuşi efectul mecanismului de paginare! O altă modalitate constă în aducerea unei pagini *la cerere*, adică atunci când este ea solicitată. Această modalitate pare a fi cea mai naturală, şi ea este într-adevăr şi cea utilizată în sistemele de operare moderne.

Există însă metode de încărcare prin care se aduc pagini *în avans*. Astfel, odată cu o pagină se aduc şi câteva pagini vecine, în ipoteza că ele vor fi invocate în viitorul apropiat. O evidenţă statistică a utilizării paginilor, poate furniza, cu o oarecare probabilitate, care ar fi paginile invocabile în viitor. Dacă se poate, acestea sunt aduse în avans în memoria operativă.

Legat de încărcarea în avans, încă din 1968, P.J. Denning [10] a emis *principiul vecinătăţii:* adresele de memorie solicitate de un program nu se distribuie uniform pe întreaga memorie folosită, ci se grupează în jurul unor centre. Apelurile în apropierea acestor centre sunt mult mai frecvente decât apelurile de la un centru la altul.

Acest principiu sugerează o politică simplă de încărcare în avans a unor pagini. Se stabileşte o aşa zisă *memorie de lucru* [48] compusă din câteva pagini. Atunci când se cere aducerea unei pagini de pe disc, în memoria de lucru sunt încărcate câteva pagini vecine acesteia. In conformitate cu principiul vecinătăţii, este foarte probabil ca următoarele referiri să fie făcute în cadrul memoriei de lucru.

10.4.4 Politici de înlocuire.

In mod natural, numărul total al paginilor programelor active poate deveni mai mare decât numărul paginilor fizice din memoria operativă. Din această cauză, uneori apare situaţia că un program cere încărcarea unei pagini virtuale, dar nu există o pagină fizică disponibilă pentru a o găzdui. Intrebarea este *care* dintre paginile fizice va fi evacuată pentru a crea spaţiul necesar?

Răspunsul optim este simplu, dar imposibil de realizat: *se evacuează acea pagină care va fi solicitată în viitor cel mai târziu* !?! Desigur, acest lucru nu poate fi prevăzut în prealabil, dat fiind faptul că evoluţia unui program la un moment dat este dependentă de datele concrete asupra cărora operează. Totuşi, Belady [4] a descris un model de evidenţă statistică prin care se poate prevedea cu o oarecare probabilitate care este pagina care va fi solicitată cel mai târziu.

Dintre metodele mai "ortodoxe", de înlocuire, descrise printre altele în [22], [10], noi vom detalia trei:

- înlocuirea unei pagini care nu a fost recent utilizată (NRU - Not Recently Used);
- înlocuirea în ordinea încărcării paginilor (FIFO - First In First Out);
- înlocuirea paginii nesolicitate cel mai mult timp (LRU - Least Recently Used).

Evident, metodele de înlocuire prezentate mai sus sunt foarte simplu de descris. Pentru implementare trebuie avut în vedere faptul că întreţinerea unei structuri de date care să permită decizia trebuie făcută *la fiecare acces la memorie*. In această situaţie, nu este permisă nici măcar întreţinerea unei liste simplu înlănţuite! De cele mai multe ori aceste metode se implementează prin hard, făcându-se uneori compromisuri.

10.4.4.1 Metoda NRU.

Fiecare pagină fizică are asociaţi doi biţi, prin intermediul cărora se va decide pagina de evacuat. Bitul **R**, numit bit de *referire*, primeşte valoarea 0 la încărcarea paginii. La fiecare referire a paginii, acest bit este pus pe 1. Periodic (de obicei la 20 milisecunde), bitul este pus iarăşi pe 0. Bitul **M**, numit bit de *modificare*, primeşte valoarea 0 la încărcarea paginii. El este modificat numai la scrierea în pagină, când i se dă valoarea 1. Aceşti doi biţi împart în fiecare moment paginile fizice în patru clase:

0. clasa 0: pagini nereferite şi nemodificate;
1. clasa 1: pagini nereferite (în intervalul fixat), dar modificate de la încărcarea lor;
2. clasa 2: pagini referite dar nemodificate;
3. clasa 3: pagini referite şi modificate.

Atunci când o pagină trebuie înlocuită, pagina "victimă" se caută mai întâi în clasa 0, apoi în clasa 1, apoi în clasa 2 şi în sfârşit în clasa 3. Dacă pagina de înlocuit este în clasa 1 sau clasa 3, conţinutul ei va fi salvat pe disc înaintea înlocuirii. Acest algoritm simplu, deşi nu este optimal, s-a dovedit în practică a fi foarte eficient.

10.4.4.2 Metoda FIFO.

Implementarea acestei metode este foarte simplă. Se crează şi se întreţine o listă a paginilor în ordinea încărcării lor. Această listă se actualizează *la fiecare nouă încărcare de pagină* (nu la fiecare acces la memorie!). Atunci când se cere înlocuirea este substituită prima (cea mai veche) pagină din listă. Bitul **M** de modificare indică dacă pagina trebuie sau nu salvată înaintea înlocuirii.

O primă îmbunătăţire ar fi combinarea algoritmilor NRU şi FIFO, în sensul că se aplică întâi NRU, iar în cadrul aceleiaşi clase se aplică FIFO.

O altă îmbunătăţire este cunoscută sub numele de *metoda celei de-a doua şanse*. La această metodă, se testează bitul **R** de referinţă la pagina cea mai veche. Dacă acesta este 0, atunci pagina este înlocuită imediat. Dacă este 1, atunci este pus pe 0 şi pagina este pusă ultima în listă, ca şi cum ar fi intrat recent în memorie. Apoi căutarea se reia cu o nouă listă. Evident, şansa de scăpare a unei pagini victimă este să existe o pagină mai "tânără" ca ea şi care să nu fi fost referită.

Si acum o curiozitate! Bunul simţ ne spune că şansa ca o pagină să fie înlocuită scade pe măsură ce numărul de pagini fizice creşte. Si totuşi nu este aşa! In [4] [22] este dat un contraexemplu, cunoscut sub numele de *anomalia lui Belady*, pe care îl ilustrăm în fig. 10.21a şi 10.21b. Este vorba de un program care are 5 pagini virtuale, pe care le solicită în ordinea:

0 1 2 3 0 1 4 0 1 2 3 4

In fig. 10.21a este ilustrată evoluţia când există 3 pagini fizice, iar în fig. 10.21b când există 4 pagini fizice.

```
a) Trei pagini fizice:
       Solicitare pagina: 0 1 2 3 0 1 4 0 1 2 3 4

       Pagina recenta   : 0 1 2 3 0 1 4 4 4 2 3 3
                          . 0 1 2 3 0 1 1 1 4 2 2
       Pagina mai veche : . . 0 1 2 3 0 0 0 1 4 4

       Inlocuire pagina : I I I I I I I       I I   = 9


b) Patru pagini fizice:
       Solicitare pagina: 0 1 2 3 0 1 4 0 1 2 3 4

       Pagina recenta   : 0 1 2 3 3 3 4 0 1 2 3 4
                          . 0 1 2 2 2 3 4 0 1 2 3
                          . . 0 1 1 1 2 3 4 0 1 2
       Pagina mai veche : . . . 0 0 0 1 2 3 4 0 1

       Inlocuire pagina : I I I I     I I I I I I =  10
```

Figura 10.21 Anomalia lui Belady

10.4.4.3 Metoda LRU.

LRU (pagina mai puţin folosită în ultimul timp) este un algoritm bun de înlocuire. El are la bază următoarea observaţie, reieşită (tot) din principiul vecinătăţii. O pagină care a fost solicitată mult de către ultimele instrucţiuni, va fi probabil solicitată mult şi în continuare. Invers, o pagină solicitată puţin (sau deloc), va rămâne probabil tot aşa pentru câteva instrucţiuni.

Problema este cum să se ţină evidenţa utilizărilor? Se *exclude* din start întreţinerea unei liste înlănţuite *care să fie modificată la fiecare acces la memorie.* Preţul plătit este mult prea mare. Iată două posibile rezolvări.

Numărătorul de accese se implementează hard. Există un registru numit *contor* reprezentat (de regulă) pe 64 de biţi. La fiecare acces, valoarea lui este mărită cu o unitate. In tabela de pagini, există câte un spaţiu rezervat pentru a memora valoarea contorului. In momentul accesului la o pagină, valoarea contorului este memorată în acest spaţiu rezervat din tabela de pagini. Atunci când se impune o înlocuire, este înlocuită pagina care a reţinut cea mai mică valoare a contorului.

Matricea de referinţe. Pentru un sistem de calcul care are ***n*** pagini fizice, se utilizează o matrice binară de ***n*n***. La pornire, toate elementele au valoarea 0. Atunci când se face referire la o pagină ***k***, linia ***k*** a matricei este înlocuită peste tot cu 1, după care coloana ***k*** este înlocuită peste tot cu 0. In fiecare moment, numărul de cifre 1 de pe o linie oarecare *l* arată de câte ori a fost referită pagina *l* după încărcare.

Iată, spre exemplu, în fig. 10.22, cum arată evoluţia matricei de referinţe într-un sistem de calcul care are patru pagini fizice, solicitate în ordinea:

0 1 2 3 2 1 0 3 2 3:

```
                    0 1 2 3 2 1 0 3 2 3

0111 0011 0001 0000 0000 0000 0111 0110 0100 0100
0000 1011 1001 1000 1000 1011 0011 0010 0000 0000
0000 0000 1101 1100 1101 1001 0001 0000 1101 1100
0000 0000 0000 1110 1100 1000 0000 1110 1100 1110
```

Figura 10.22 Evoluţia unei matrice de referinţe

Implementarea mecanismului matricei de referinţe se face destul de uşor prin hard, în orice caz mult mai uşor decât implementarea mecanismului cu numărător de referinţe.

Desigur, politicile de înlocuire descrise mai sus pot fi simulate foarte bine prin soft. Din păcate, eficienţa mecanismelor scade drastic. Acesta este motivul pentru care unele implementări ale memoriei virtuale (nu dăm aici nume) au eşuat lamentabil. Nu-i nimic, s-a câştigat experienţă şi asta nu este lucru puţin.

10.4.5 Cum funcţionează o memorie cache?

Apariţia memoriei *cache* a fost dictată de necesitatea creşterii performanţelor sistemului de calcul. Memoria cache conţine copii ale unor blocuri din memoria operativă. Când procesorul încearcă citirea unui cuvânt de memorie, se verifică dacă acesta există în memoria cache. Dacă există, atunci el este livrat procesorului. Dacă nu, atunci el este căutat în memoria operativă, este adus în memoria cache împreună cu blocul din care face parte, după care este livrat procesorului. Datorită vitezei mult mai mari de acces la memoria cache, randamentul general al sistemului creşte.

Aici apar o serie de probleme: cât de mare este o memorie cache? care este dimensiunea optimă a blocului de memorie destinat schimbului cu această memorie? In lucrarea [42] se dau răspunsuri la aceste întrebări. Tot acolo se fac cuvenitele evaluări ale raportului cost/performanţă pentru dimensionarea unei memorii cache, precum şi care este probabilitatea (hit ratio) ca o adresă cerută de procesor să fie găsită în memoria cache.

Noi ne rezumăm la ilustrarea modului în care se face corespondenţa dintre memoria operativă şi memoria cache, precum şi politicile de schimb dintre cele două tipuri de memorie.

Memoria cache este împărţită în mai multe părţi egale, numite *sloturi* (poziţii). Un slot are dimensiunea unui bloc de memorie, care este în mod obligatoriu o putere a lui 2. Fiecare slot conţine în plus câţiva biţi (numiţi generic *tag* = etichetă) care indică blocul de memorie operativă depus în slotul respectiv. Dimensiunea unui tag depinde de mecanismul de schimb dintre cele două memorii, dar noi nu ne vom ocupa aici de el. De exemplu, în fig. 10.23 şi 10.24 am ales o memorie operativă de 64 Ko şi o memorie cache de 1 Ko. Dimensiunea unui slot, identică cu dimensiunea unui bloc de memorie, am ales-o de 8 octeţi (64 de biţi).

Se cunosc [42] mai multe metode de *proiectare a spaţiului memoriei operative pe memoria cache.* Cea mai simplă metodă este *proiectarea directă.* Dacă ***C*** indică numărul total de sloturi din memoria cache, ***A*** este o adresă oarecare din memoria operativă, atunci numărul ***S*** al slotului în care se proiectează adresa *A* este:

$$S = A \bmod C$$

In fig. 10.23 este ilustrată această corespondenţă.

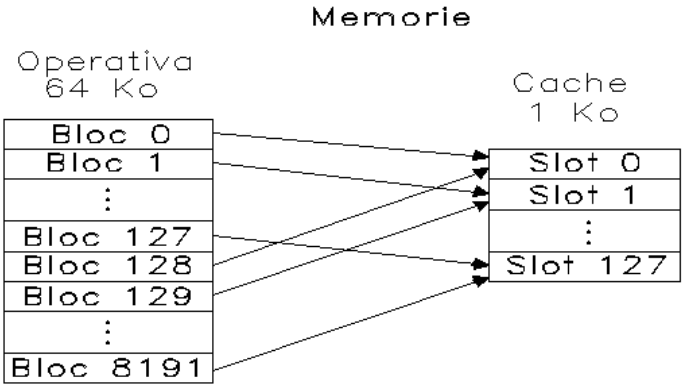

Figura 10.23 Proiectare directă pe memoria cache

Principalul ei avantaj este simplitatea. In schimb, există un mare dezavantaj, generat de faptul că fiecare bloc are o poziţie fixă în memoria cache. Dacă, spre exemplu, se cer accese alternative la două blocuri care ocupă acelaşi slot, atunci are loc un trafic intens între cele două memorii, fapt care face să scadă mult randamentul general al sistemului.

O a doua metodă poartă numele de *proiectare asociativă*, apărută pentru eliminarea dezavantajului de mai sus. Acestă metodă plasează un bloc de memorie într-un slot oarecare liber.

Gestiunea sloturilor memoriei cache în acest caz ridică aceleaşi probleme ca şi alocarea paginată a memoriei. Principala problemă care se ridică aici este cea legată de *politica de înlocuire*. In paragraful 2.4.3 am tratat această problemă, descriind algoritmii NRU, FIFO şi LRU. Tot ceea ce s-a spus acolo este valabil, fără modificări şi la înlocuirea unui slot atunci când un bloc solicitat din memoria operativă nu mai are loc în memoria cache.

O a treia metodă poartă numele de *proiectarea set-asociativă*, şi ea este o combinaţie a precedentelor două metode. Ideea ei este următoarea. Memoria cache este împărţită în *I seturi*, un set fiind compus din *J* sloturi. Avem deci relaţia ***C = I*J*.** La cererea de memorie de la adresa *A*, se calculează numărul *K* al setului în care va intra blocul, astfel:

$$K = A \bmod I$$

Având fixat numărul setului, blocul va ocupa unul dintre sloturile acestui set. Alegerea slotului este de această dată o problemă de planificare. In fig. 10.24 este prezentat cazul ***I*=64** şi ***J*=2**.

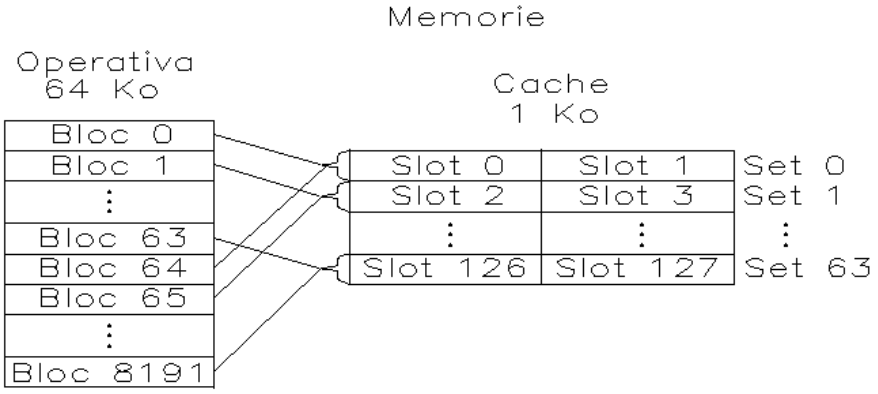

Figura 10.24 Proiectarea set - asociativă

Metoda set-asociativă este mult folosită în practică. Compromisul proiectării directe şi a celei asociative face să fie aplicată o politică de înlocuire numai atunci când într-un slot există deja ***J*** blocuri, ceea ce se întâmplă foarte rar. In rest se aplică proiectarea directă, care este mult mai simplă.