

## 10 Gestiunea memoriei

### 10.1 Structură; calculul de adresă

#### 10.1.1 Problematika 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 folosește zona (zonele) de memorie alocată (alocate) lui, independent de eventuale alte programe active.

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

*Segmentarea* este o tehnică prin care un program executabil este decupat în entități distincte numite *segmente*. Segmentele pot fi: de cod, date sau de stivă. Pe durata vieții programului unele dintre segmente pot fi prezente în memorie, altele nu. Programul însuși 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 și are forma din fig. 10.1a. Atunci segmentele active la un moment dat pot fi: A, AC, AB, AF, ACE sau ACD, așa cum reiese din fig.10.1b.

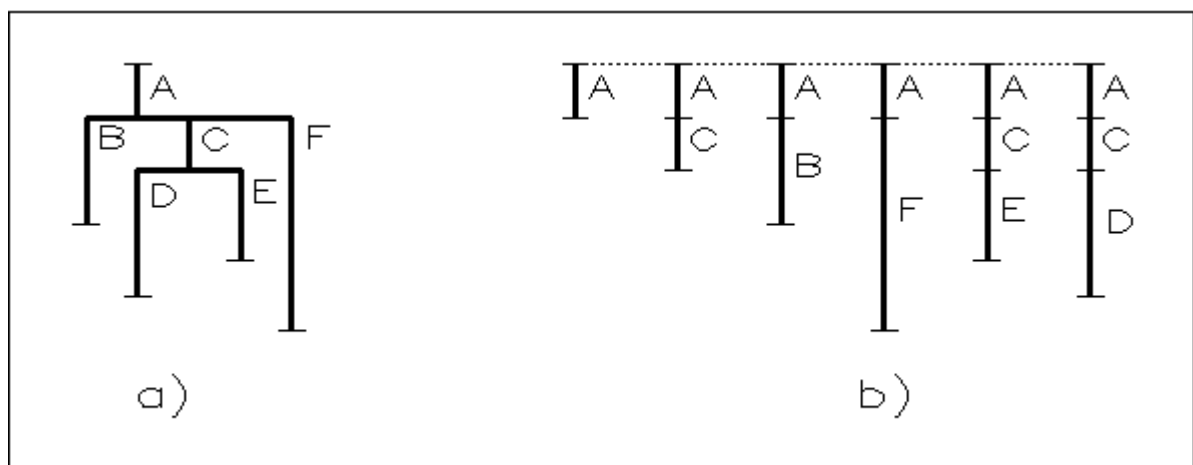


Figura 10.1 Instanțe posibile ale unui program segmentat

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

*Spațiul* 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 spațiul de memorie oferit de actualele **SC** este mult mai mare decât cel oferit acum 20 de ani sau chiar 10 ani (iar prețul de cost al memoriei a scăzut drastic), limitarea totuși rămâne. Din această cauză, **SO** cu sprijinul **SC** trebuie să gestioneze eficient folosirea acestui spațiu.

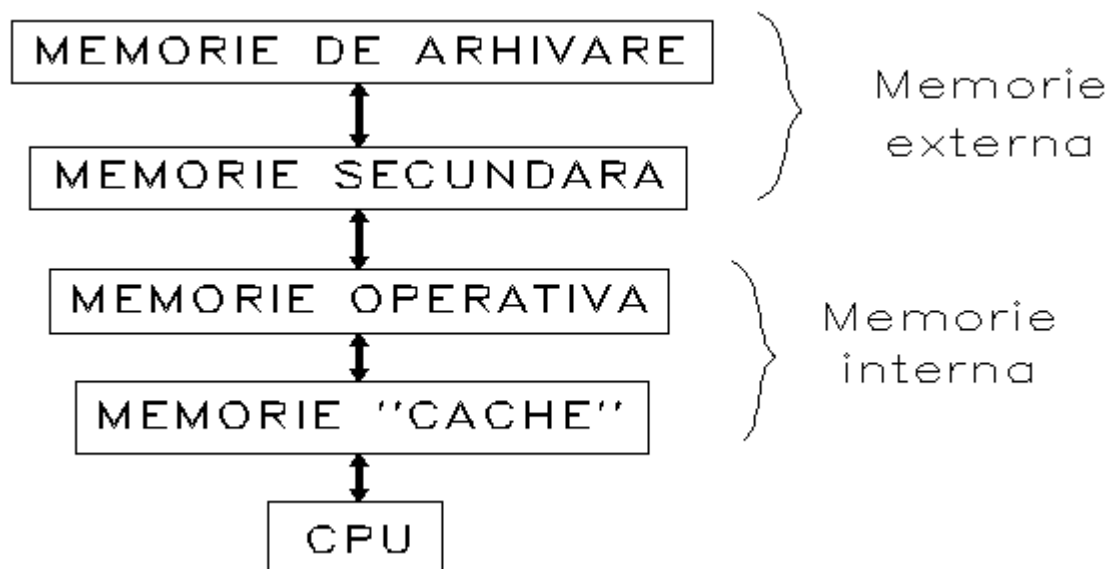
Problemele privind gestiunea memoriei sunt rezolvate la nivelul inferior de către **SC**, extins eventual cu o componentă de *management al memoriei*. La nivelul superior, rezolvarea se face de către **SO** în colaborare cu **SC**. Principalele obiective ale gestiunii memoriei sunt:

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

Vom aborda fiecare dintre aceste obiective, cu excepția protecției memoriei. În general protecția 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 protecție a memoriei, deci aceasta trebuie studiată în context concret.

### 10.1.2 Structura ierarhică de organizare a memoriei

În 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.

*Memoria "cache"* conține informațiile 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" și abia apoi solicită memoria operativă. Dacă este, atunci are loc schimbul între **CPU** și ea. Dacă nu, atunci data este căutată în nivelele superioare. Aplicând *principiul vecinătății* [40], data respectivă este adusă din nivelul la care se află, dar odată cu ea se aduce un număr de locații vecine ei, astfel încât, împreună să umple memoria "cache". Principiul de lucru este cel al memoriilor tampon temporare, prezentat în 11.1. Într-o secțiune viitoare vom prezenta în detaliu cum funcționează memoria cache.

În ce constă principiul vecinătății? 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* conține programele și datele pentru toate procesele existente în sistem. În momentul în care un proces este terminat și distrus, spațiul de memorie operativă pe care l-a ocupat este eliberat și 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 dețin mecanisme de memorie virtuală. De asemenea, tot în cadrul memoriei secundare poate fi inclus spațiul 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 și constă din fișiere, baze de date ș.a. rezidente pe diferite suporturi magnetice (discuri, benzi, etc.).

Memoria "cache" și memoria operativă formează ceea ce cunoaștem 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 și de arhivare, acestea trebuie mai întâi mutate în memoria internă.

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

- viteza de acces la memorie scade;
- prețul de cost pe unitatea de alocare scade;
- capacitatea de memorare crește.

### 10.1.3 Mecanisme de translatare a adresei

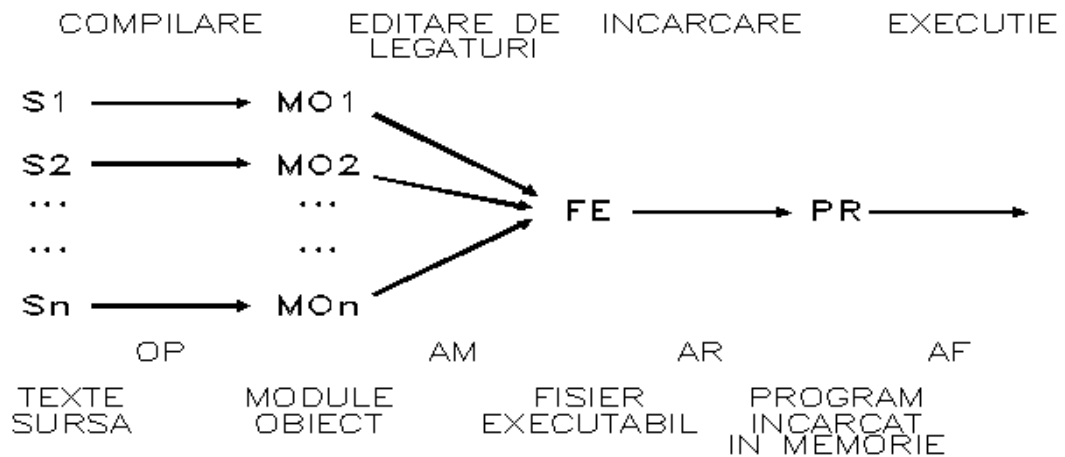
Adresarea memoriei constă în realizarea legăturii între un obiect al programului și adresa corespunzătoare din memoria operativă a **SC**. Pentru a explica mecanismele de translatare, să adoptăm câteva *notații*:

- **OP** notăm spațiul 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 fișier executabil.
- **AF** notăm mulțimea adreselor fizice din memoria operativă la care face referire programul în timpul execuției.

*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 funcții: **c**, **l**, **t**, astfel:

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

În fig. 10.3 sunt ilustrate fazele prin care trece un program componentele invocate și etapele calculului de adresă, de la textul sursă până la execuție.



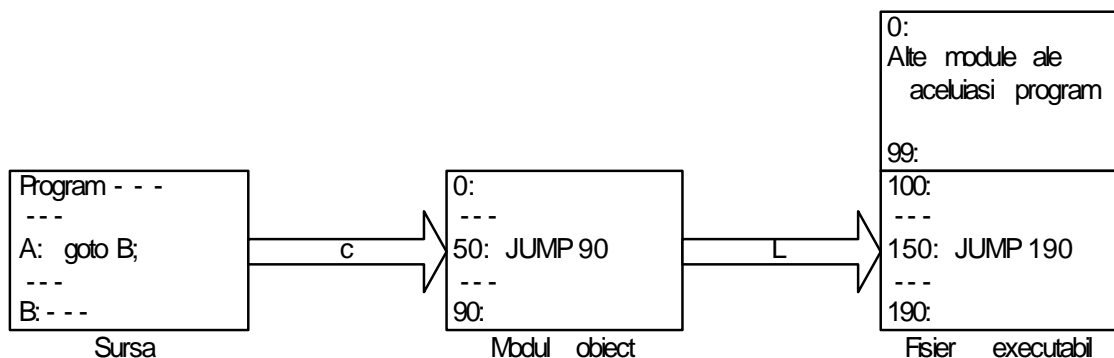
**Figura 10.3 Fazele translătării unui program**

### 10.1.3.1 Faza de compilare

Faza de *compilare* transformă un text sursă  $S_i$  într-un *modul obiect*  $MO_i$ . Corespunzător, numele obiectelor program sunt transformate în numere reprezentând  $AM$ , adică adrese în cadrul modulului obiect. În cadrul fiecărui modul obiect aceste adrese încep de la 0. Deci prima funcție din calculul de adresă:

$$c: OP \rightarrow AM$$

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



**Figura 10.4 Translatore de la sursă la fișier executabil**

### 10.1.3.2 Faza editării de legături

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

$$l: AM \rightarrow AR$$

Particularitățile de evaluare a acestei funcții sunt proprii editorului de legături. În fig. 10.4 sugerăm modul în care acționează compunerea celor două funcții.

### 10.1.3.3 Faza de încărcare și execuție

Funcția

$t: AR \rightarrow AF$

este funcția de *translatare (relocare) a adresei*. În mod obișnuit ea este executată de către CPU. Translatarea depinde de tipul sistemului de calcul, în particular de existența dispozitivului de management al memoriei. Pentru a vedea principiul ei de lucru, presupunem că fișierul executabil conține înregistrări succesive cu instrucțiuni de forma celei din fig. 10.5.

ARI CO A1 A2 --- An

**Figura 10.5** Formatul unei instrucțiuni mașină

Semnificațiile câmpurilor instrucțiunii sunt:

- *ARI* este adresa relocabilă a instrucțiunii;
- *CO* este codul operației,
- *A1, ..., An* sunt argumentele instrucțiunii mașină: nume de regiștri, constante (argumente *immediate*), adrese relocabile din memorie

Presupunem că fiecare instrucțiune încapă într-o locație de memorie și că instrucțiunile sunt plasate una după cealaltă în locații succesive. Atunci încărcarea unui astfel de fișier se poate face folosind un încărcător analog lui *LOADERABSOLUT*, descris în 8.3 fig. 8.5.

Să presupunem că mașina dispune de un singur registru general pe care-l vom numi *A* (de la *registru acumulator*). De asemenea, presupunem că între *A1, ..., An* există o singură adresă relocabilă. Evident că în realitate structura unui fișier executabil este mai complexă, dar pentru moment aceasta ne este suficientă.

Să adoptăm următoarele notații:

- $M[0..m]$  conține locațiile memoriei operative;
- *pc* (Program Counter) indică adresa fizică a instrucțiunii care urmează a fi executată;
- *w* este conținutul instrucțiunii curente;
- *Opcod(w)* este o funcție care furnizează codul operației din instrucțiunea curentă. Pentru fixarea ideilor, să presupunem că:
  - 1 este codul adunării;
  - 2 este codul operației de memorare (depunere din *A*) într-o anumită locație;
  - 3 este codul instrucțiunii de salt necondiționat;
  - ș.a.m.d.
- *Adress(w)* este o funcție care furnizează valoarea adresei relocabile aflată între argumentele instrucțiunii curente.

Cu aceste notații, în fig. 10.6 este descris modul de funcționare a CPU și de acțiune a funcției de translatare *t*. De fapt, în fig. 10.6 este schițat un *interpretor* al limbajului care are instrucțiuni de forma celei din fig. 10.5.

```

pc = t(adresa-de-start-a-programului);
do {
    w = M[pc];           // operația 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 necondiționat
        -  -  -
    }
} while (false);

```

**Figura 10.6 Funcționarea CPU și calculul funcției de traducere**

## 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 împărțindu-se la rândul ei în alte subcategorii, ca mai jos [19] [10].

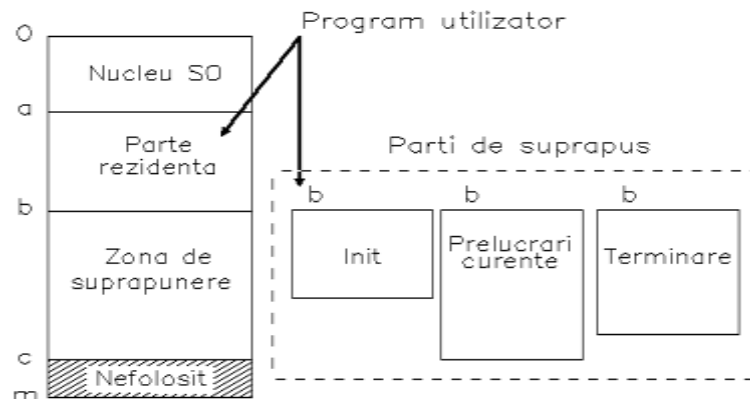
- alocare reală:
  - la **SO** monoutilizator;
  - la **SO** multiutilizator:
    - cu partiții fixe (statică):
      - absolută;
      - relocabilă;
    - cu partiții variabile (dinamică);
- alocare virtuală:
  - paginată;
  - segmentată;
  - segmentată și paginată.

### 10.2.2 Alocarea la sistemele monoutilizator

La sistemele monoutilizator este disponibil aproape întreg spațiul de memorie. Gestiunea acestui spațiu cade exclusiv în sarcina utilizatorului. El are la dispoziție tehnici de *suprapunere (overlay)* pentru a-și putea rula programele mari. În fig. 10.6 este ilustrat acest mod de lucru.

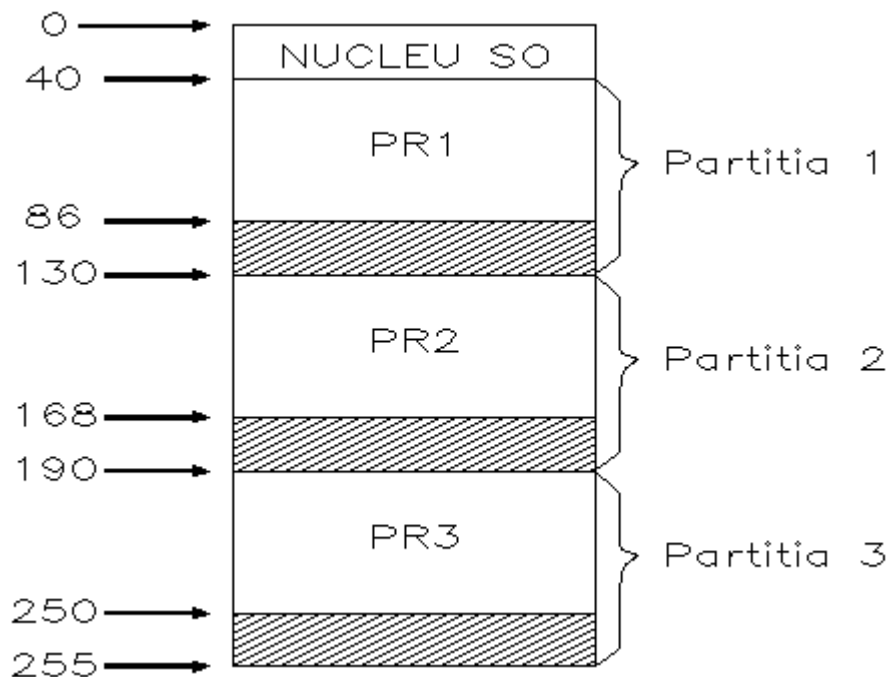
Porțiunea dintre adresele 0 și  $a-1$  este rezervată nucleului **SO**, care rămâne acolo de la încărcare și până la oprirea sistemului. Între adresele  $c$  și  $m-1$  (dacă memoria are capacitatea de  $m$  locații) este spațiu nefolosit de către programul utilizator activ. Evident, adresa  $c$  variază de la un program utilizator la altul.

### 10.2.3 Alocarea cu partiții 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 *partiții*. O partiție este alocată unui proces pe toată durata execuției lui, indiferent dacă o ocupă complet sau nu. Un exemplu al acestui mod de alocare este descris în fig. 10.8. Zonele hașurate fac parte din partiții, dar procesele active din ele nu le utilizează.



**Figura 10.8 Exemplu de alocare cu partiții fixe**

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

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

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

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

Acest mod de alocare este utilizat preponderent de către sistemele seriale. La fiecare partiție există un șir de procese care așteaptă să fie executate. Modul în care se organizează acest sistem de așteptare poate influența performanțele de ansamblu ale sistemului și poate eventual atenua efectul unei dimensionări defectuoase a partițiilor. În general există două moduri de legare a proceselor la partiții:

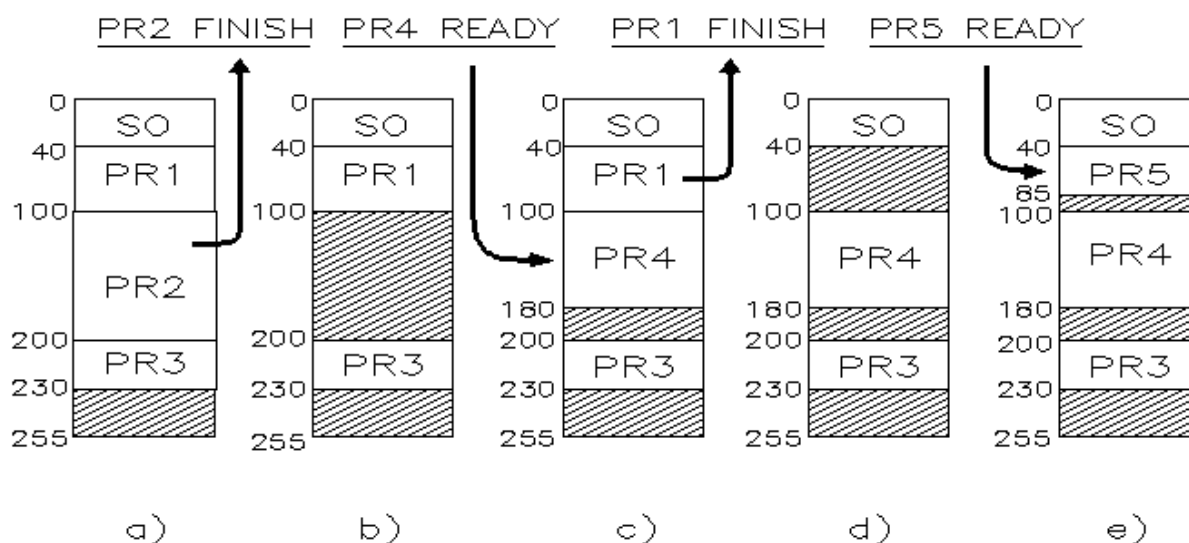
- *Fiecare partiție are coada proprie*; operatorul stabilește de la început care sunt procesele care vor fi executate în fiecare partiție.
- *O singură coadă pentru toate partițiile*; **SO** alege, pentru procesul care urmează să intre în lucru, în ce partiție se va executa.

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

#### 10.2.4 Alocarea cu partiții variabile

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

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



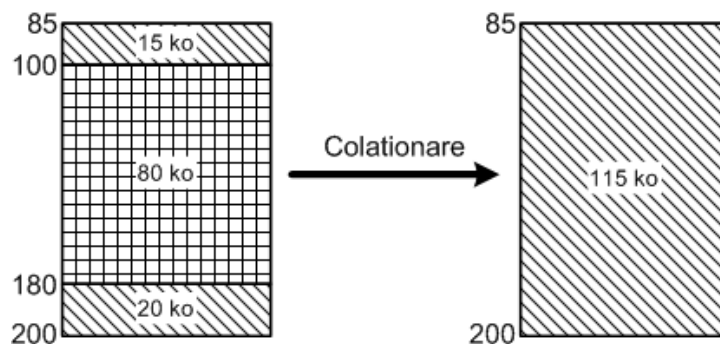
**Figura 10.9** Evoluția proceselor la alocarea cu partiții variabile

În momentul în care procesul intră în sistem, el este plasat în memorie într-un spațiu în care încapă cea mai lungă ramură a sa. Spațiul liber în care a intrat procesul este acum descompus în două partiții: una în care se află procesul, iar cealaltă într-un spațiu liber mai mic. Este ușor de observat că dacă sistemul funcționează timp îndelungat, atunci numărul spațiilor libere va crește, iar dimensiunile lor vor scădea. Fenomenul este cunoscut sub numele de *fragmentarea*

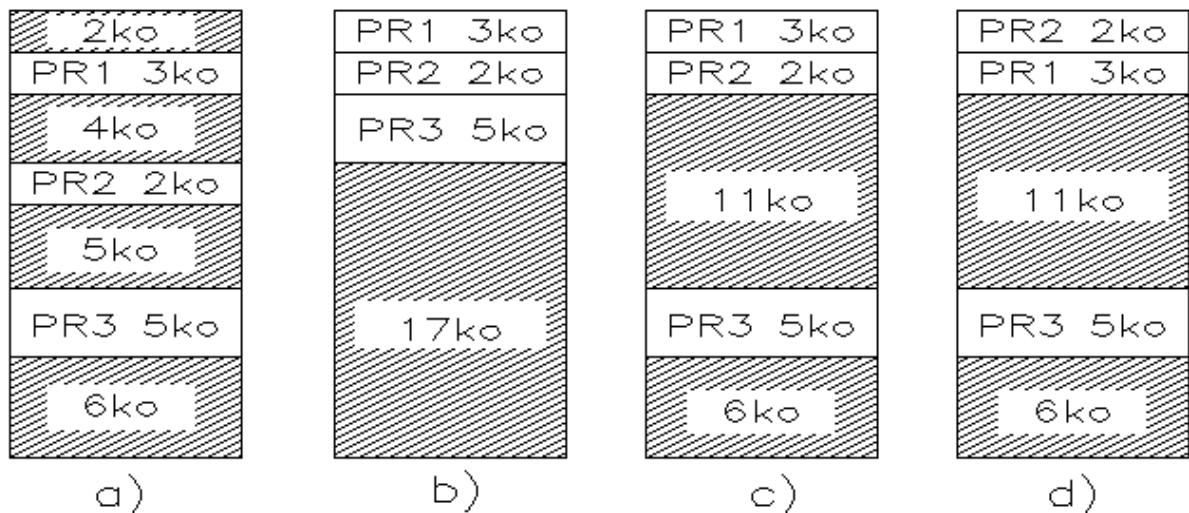


*internă a memoriei.* După cum se va vedea, acest fenomen poate avea efecte neplăcute. În momentul în care un proces nu are spațiu în care să se încarce, **SO** poate lua una din următoarele trei decizii:

- Procesul așteaptă* până când i se eliberează o cantitate suficientă de memorie.
- SO** încearcă *alipirea unor spații libere vecine - colajonare*, în speranța că se va obține un spațiu de memorie suficient de mare. Spre exemplu, dacă momentul care urmează după cel din fig. 10.9d este terminarea procesului *PR4*, 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 și face în mod automat) să formeze din aceste trei spații unul singur de 115Ko, așa cum se vede în fig. 10.10.
- SO** decide efectuarea unei operații de *compactare a memoriei (relocare)* adică de deplasare a partițiilor active către partiția monitor pentru a se absorbi toate "fragmentele" de memorie neutilizate. Este posibil ca spațiul astfel obținut să fie suficient pentru încărcarea procesului. În fig. 10.11b este dat un exemplu de compactare.



**Figura 10.10** Colajonarea de spații libere vecine



**Figura 10.11** Posibilități de compactare prin relocare totală sau parțială

De regulă, compactarea este o operație costisitoare și în practică se aleg soluții de compromis, cum ar fi:

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

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

Intre alocările de tip MFT și MVT nu există practic diferențe hard. Alocarea MVT este realizată de cele mai multe ori prin intermediul unor rutine specializate, eventual microprograme, 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 spațiu 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 Universității Manchester, Anglia, 1960 [19]. Se cunosc două metode de virtualizare, mult înrudite după cum vom vedea în continuare. Este vorba de alocarea paginată și 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, și drept consecință, la evitarea aplicării relocării. Această alocare presupune cinci lucruri și anume:

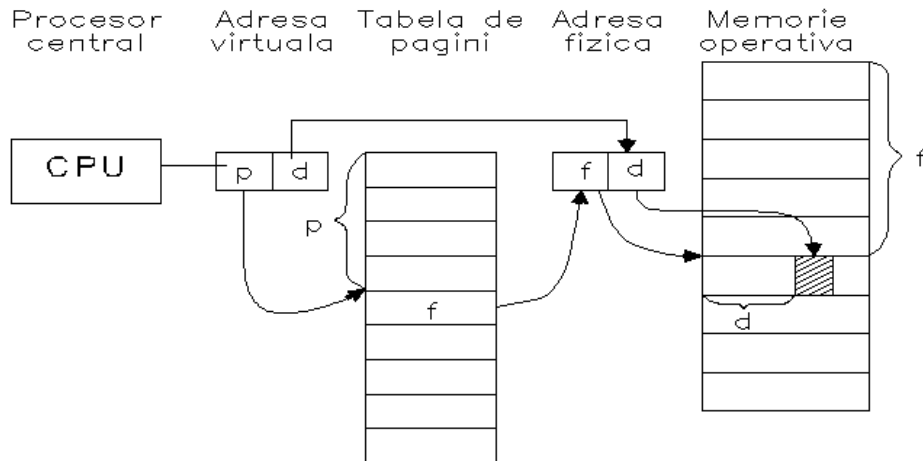
- a) Instrucțiunile și datele fiecărui program sunt împărțite în zone de lungime fixă, numite *pagini virtuale*. Fiecare **AR** (notațiile din 10.1.3) aparține unei pagini virtuale. Paginile virtuale se păstrează în memoria secundară.
- b) Memoria operativă este împărțită în zone de lungime fixă, numite *pagini fizice*. Lungimea unei pagini fizice este fixată prin hard. Paginile virtuale și cele reale au aceeași lungime, lungime care este o putere a lui 2, și 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 funcției de translație  $t : AR \rightarrow AF$  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 biți) care dă lungimea unei pagini, prin  $TP$  adresa de start a tabelii de pagini, atunci algoritmul calculului funcției  $t$  este:

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

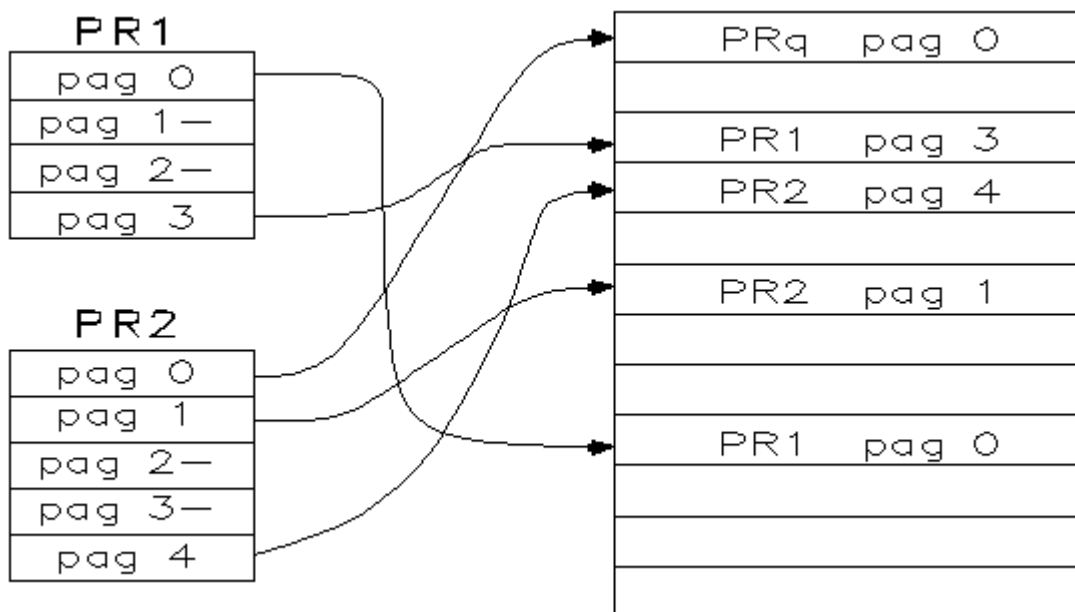
Acest calcul este valabil atunci când tabela de pagini ocupă un spațiu în memoria operativă. Există însă **SC** ce dispun, hard, de o memorie specială de capacitate mică, numită *memorie asociativă*. Calitatea ei fundamentală este *adresarea prin conținut*, ceea ce înseamnă că

găsește locația care are un conținut specificat, căutând simultan în toate locațiile ei. Memoria asociativă conține atâtea locații câte pagini fizice are. În fiecare locație a memoriei asociative este trecut numărul paginii virtuale care se află în pagina fizică având numărul de ordine identic cu numărul de ordine al locației de memorie asociativă. Atunci când se dă un număr de pagină virtuală, se obține automat numărul paginii fizice care o găzduiește.



**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 spațiului virtual peste cel real*. Este posibil, la un moment dat, să existe situația din fig. 10.13.



**Figura 10.13** Două procese, într-o alocare paginată

Acest mecanism are avantajul că folosește 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 porțiuni de cod.

Să presupunem că avem un editor de texte al cărui cod (instrucțiuni 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. În fig. 10.14 se arată cum pot fi consumate numai cinci pagini.

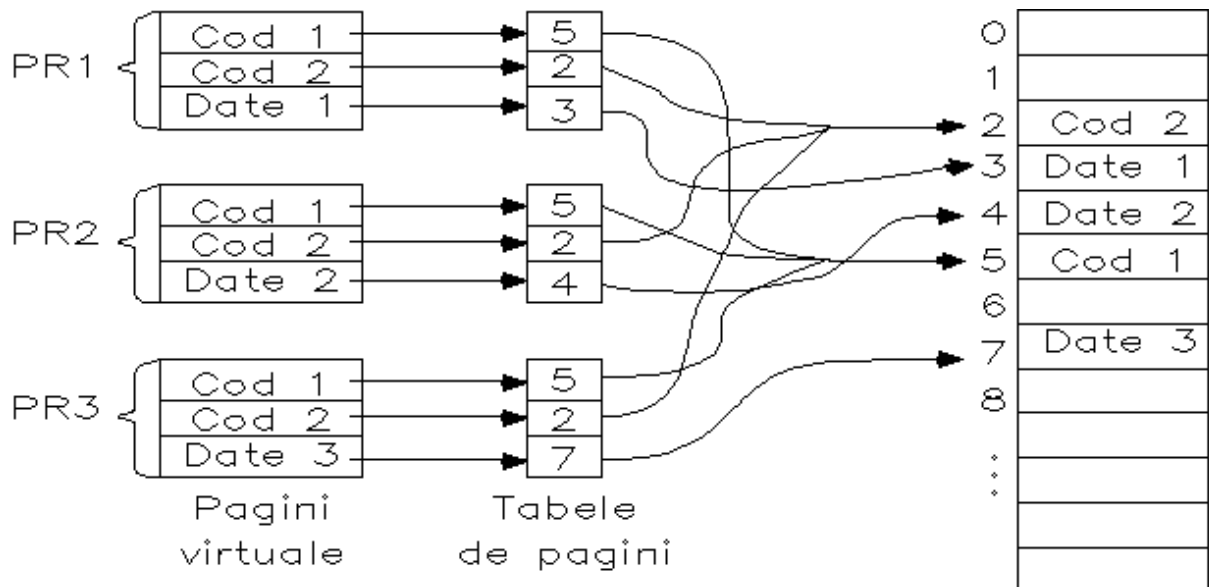


Figura 10.14 Folosirea în comun a unui cod

### 10.3.2 Alocare segmentată

Atunci când am vorbit despre alocarea reală, am văzut că fiecare proces trebuia să ocupe un spațiu contiguu de memorie, numit partiție. 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 conținând o bucată de program numită *segment*. Singura deosebire principală dintre alocarea paginată și cea segmentată este aceea că segmentele *sunt de lungimi diferite*. În fig. 10.15 am ilustrat această situație cu locurile ce vor fi ocupate de un program format din trei segmente.

În 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 obișnuită. Fiecare proces activ are o *tabelă de segmente*. Fiecare intrare în această tabelă conține *adresa de început a segmentului*. Calculul de adresă se face analog celui de la alocarea paginată. Presupunem că la adresa  $TS$  se află începutul tabelii de segmente. Cu notațiile obișnuite, funcția  $t$  de traducere a adresei se calculează astfel:

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

Pe lângă avantajul net față de alocările pe partiții, 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 aceeași adresă pentru segmentul pur. A se vedea aceeași problemă discutată la alocarea paginată.
- Se poate realiza o *foarte bună protecție 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 și astfel de verificări. Pentru detalii se pot consulta lucrările [10], [19].

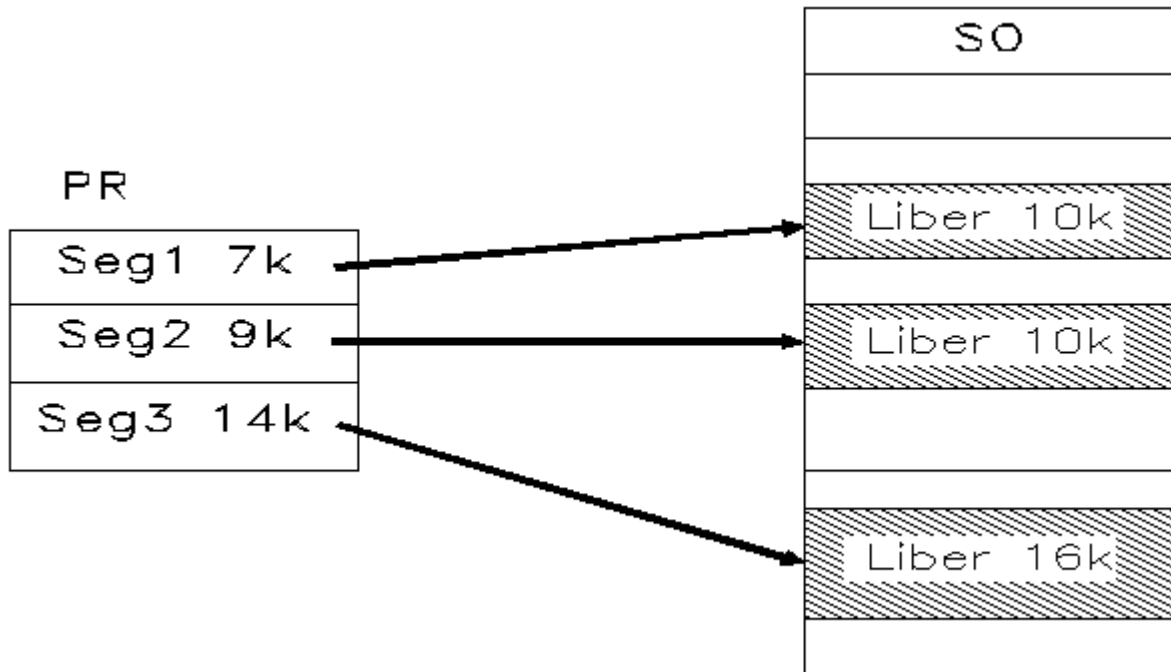


Figura 10.15 Alocare necontiguă prin segmentare

### 10.3.3 Alocare segmentată și 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 partiții variabile (10.2.4). Ideea alocării segmentate și paginate este aceea că alocarea spațiului pentru fiecare segment să se facă paginat.

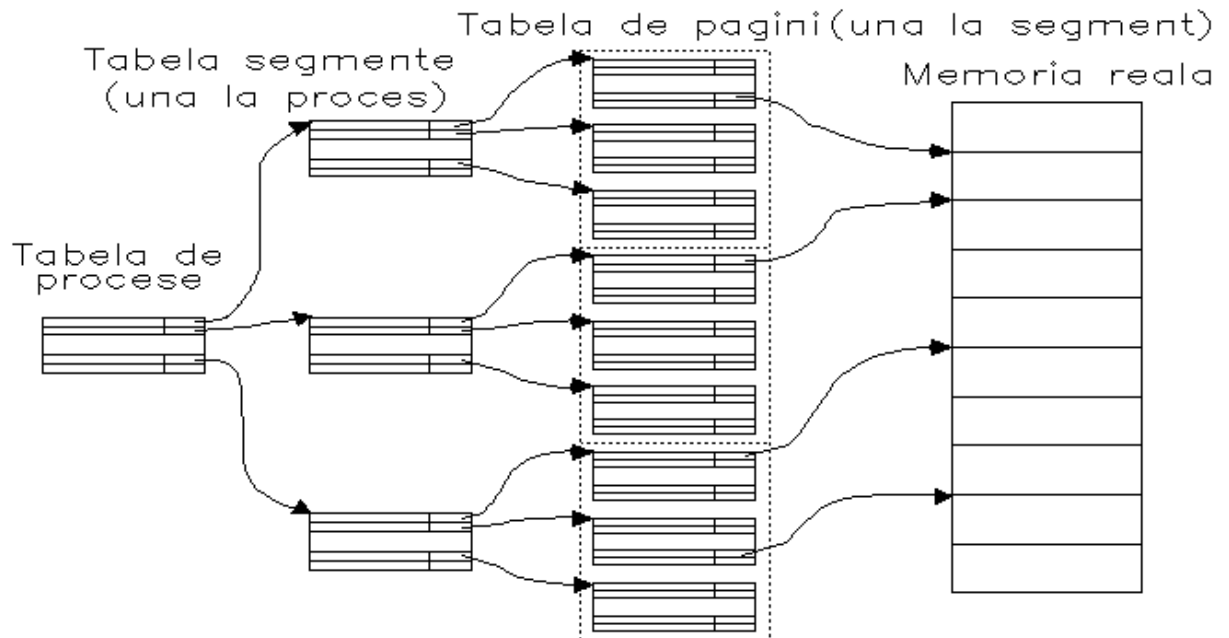
Pentru aceasta, mai întâi *fiecare proces* activ are *propria lui tabelă de segmente*. Apoi, *fiecare segment* dintre cele încărcate în memorie are *propria lui tabelă de pagini*. Fiecare intrare în tabela de segmente are un câmp rezervat adresei de început a tabelii de pagini proprii segmentului respectiv, așa 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 tabelii de segmente a unui proces și presupunem că primul câmp al fiecărei intrări din tabela de segmente este pointerul spre tabela lui de pagini, atunci funcția  $t$  de traducere se calculează astfel:

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

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 și paginată

Calculatoarele IBM-PC cu microprocesor cel puțin 80386, dispun de un mecanism hard de gestiune paginată și segmentată a memoriei extinse. Conceptual el funcționează așa cum am arătat mai sus, completat evident cu o serie de particularități legate de adresarea și protecția memoriei la acest tip de mașină.

## 10.4 Planificarea schimburilor cu memoria

### 10.4.1 Întrebările gestiunii memoriei și politici de schimb.

Pe lângă mecanismele de alocare descrise în 10.2 și 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?

*Întrebarea "CAT?"*, apare atunci când se pune problema cantității de memorie alocată. La alocarea pe partiții, 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. În 10.3 am prezentat aceste tehnici.

*Întrebarea "UNDE?"*, apare la alocarea cu partiții 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, depășind cadrul sistemului de operare. Din acest motiv (cât și datorită farmecului lor) le vom dedica o secțiune specială.

*Întrebarea "CAND?"*, apare cel puțin în două situații: la sistemele cu paginare și la alocarea cu partiții variabile. În sistemele cu paginare, trebuie stabilit momentul în care o pagină virtuală este depusă într-o pagină fizică. În literatură, tehnicile de răspuns sunt cunoscute sub numele de *politici de încărcare (fetch)*. Le vom dedica și acestora o secțiune specială.

Aceeași întrebare "CAND?", apare pentru a decide momentele de *compactare (relocare)* a memoriei la alocarea cu partiții variabile. În 10.2.4 am arătat în ce constă compactarea și am dat trei posibilități de acțiune 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 secțiune separată.

Deși, face parte dintre metodele implementate hard, deci nu implică deloc sistemul de operare, credem că este cazul să vedem puțin *cum funcționează o memorie cache?* Vom dedica și acesteia o secțiune specială.

## 10.4.2 Politici de plasare.

### 10.4.2.1 Metode de plasare și structuri de date folosite

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

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

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

Dată fiind importanța acestor politici, vom da câteva metode de organizare a spațiului de memorie și algoritmii de ocupare și alocare corespunzători. Pe lângă cerințele de funcționalitate, 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 deși sistemul dispune per total de memoria necesară.

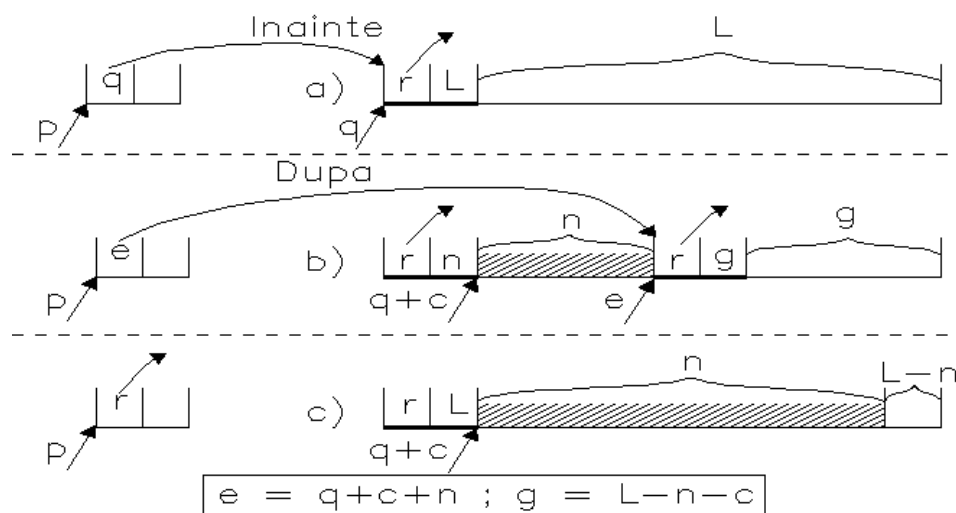
Dintre metodele de plasare, cele mai răspândite sunt următoarele patru:

- 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.

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

Vom prezenta în continuare câțiva algoritmi de plasare a cererilor de memorie. Pentru a trata unitar alocarea și eliberarea octeților, vom adopta următoarea convenție. Fiecare zonă liberă sau ocupată de memorie începe cu un cuvânt de control. Câmpul **lung** (aflat în a doua jumătate a cuvântului de control) al lui indică numărul de octeți 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. În fig. 10.17, 10.18 și 10.19, cuvintele de control sunt subliniate cu linii îngroșate, iar pointerii la zone sunt indicați prin săgeți. De asemenea, lungimea cuvântului de control este notată cu  $c$  și este o constantă specifică sistemului de operare. După cum vom vedea apar situații în care numărul de octeți alocați depășește (cu maximum  $c$ ) numărul de octeți solicitați. O zonă ocupată este reperată *după* cuvântul ei de control.

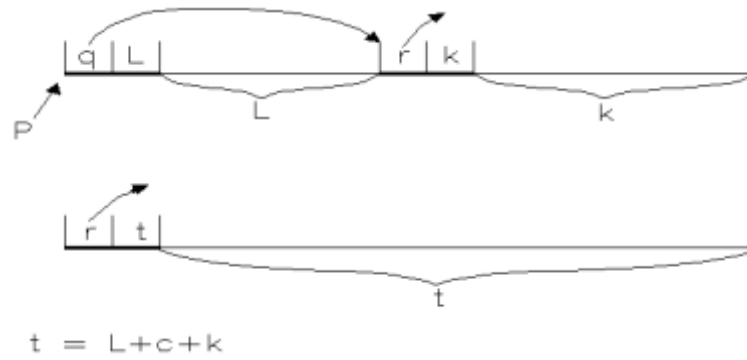


**Figura 10.17** Alocarea de octeți într-o zonă liberă

În fig. 10.17a este prezentată o zonă liberă al cărei cuvânt de control începe la adresa  $q$  și are  $L$  octeți liberi. Presupunem că se cere alocarea de  $n$  octeți și că  $L > n$ . Ca rezultat al alocării, va apare una dintre situațiile din fig. 10.17b sau 10.17c. Zona hașurată reprezintă octeții ceruți pentru alocare. Situația din fig. 10.17b apare dacă  $L - n > c + 1$ , adică spațiul rămas în zonă permite crearea unei zone libere de cel puțin un octet. În cazul în care această condiție nu este satisfăcută, se alocă întreaga zonă și ultimii  $L - n$  octeți rămân nefolosiți fig. 10.17c).

Înainte 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. Situația în care este posibilă concatenarea a două zone libere este ilustrată în fig. 10.18.





**Figura 10.18 Comasarea a două zone libere adiacente**

În fig. 10.19 este ilustrat un exemplu de eliberare a unei zone. La eliberare, partiția devenită liberă se repune în lista de spații libere. De asemenea, se verifică dacă nu cumva partiția 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*.

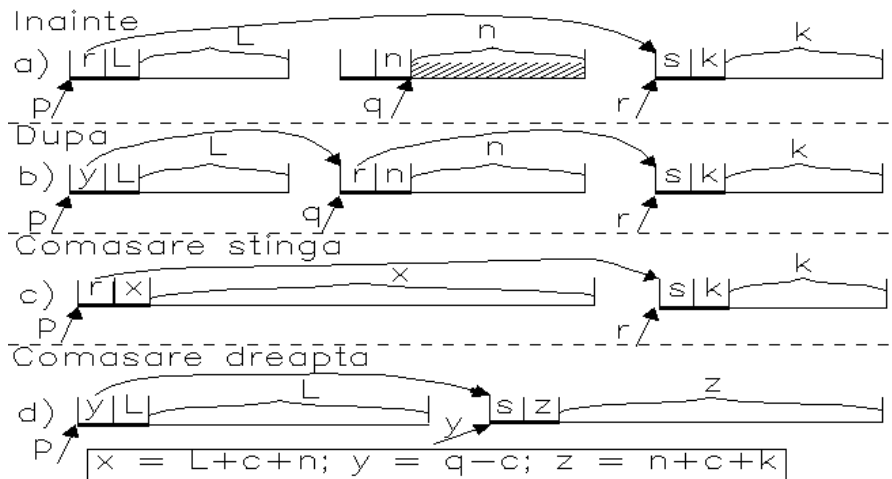
În fig. 10.19a este prezentată situația înainte de eliberarea zonei. În fig. 10.19b se prezintă situația în care nu este posibilă nici un fel de comasare. Posibilitatea de *comasare la stânga* are loc dacă:

$$p + 2 * c = q$$

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

$$q + n = r$$

Ca efect se obține 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 că 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*).

Esența metodei constă în aceea că partiția solicitată este alocată în prima zonă liberă, în care încap. Principalul avantaj al metodei este simplitatea căutării de spațiu liber. Pentru această metodă, structura listei de spații libere prezentate mai sus este cea mai adecvată.

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

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

Primul neajuns este eliminat parțial dacă lista de spații libere se păstrează, nu în ordinea crescătoare a adreselor, ci *în ordinea crescătoare a lungimilor spațiilor 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 duală metodei Best-Fit. Esența ei constă în căutarea acelei zone libere care lasă după alocare cel mai mult spațiu liber.

Deși, numele ei sugerează că este vorba de o metodă slabă, în realitate nu este chiar așa. Faptul, că după alocare rămâne un spațiu liber mare, este benefic, deoarece în spațiul rămas poate fi plasată, în viitor, o altă partiție. 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 spații libere să se păstreze nu în ordinea crescătoare a adreselor, ci *în ordinea descrescătoare a lungimilor spațiilor 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 și faptul că din rațiuni tehnologice, dimensiunea memoriei interne este un multiplu al unei puteri a lui doi. Fie  $c \cdot 2^n$  dimensiunea memoriei interne. De exemplu, memoria unui IBM PC XT (unul dintre primele calculatoare personale) era 640 Ko, adică  $10 \cdot 2^{16}$  octeți.

Să notăm cu  $n$  cea mai mare putere a lui 2 prin care se poate exprima dimensiunea memoriei interne. În exemplul de mai sus  $n = 16$ . Din rațiuni practice, se stabilește 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 octeți, adică  $2^8$ .

La sistemele Buddy, dimensiunile spațiilor ocupate și a celor libere sunt de forma  $2^k$ , unde  $m \leq k \leq n$ . Ideea fundamentală este de a păstra liste separate de spații disponibile pentru

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

Prin definiție, fiecare spațiu liber sau ocupat de dimensiune  $2^k$  are adresa de început un multiplu de  $2^k$ .

Tot prin definiție, două spații libere de ordinul  $k$  se numesc *camarazi* (*Buddy*) de ordin  $k$ , dacă adresele lor  $A1$  și  $A2$  verifică:

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

sau

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

Atunci când într-o listă de ordin  $k$  apar doi camarazi, sistemul îi concatenează într-un spațiu de dimensiune  $2^{k+1}$ .

*Alocarea într-un sistem Buddy* se desfășoară astfel:

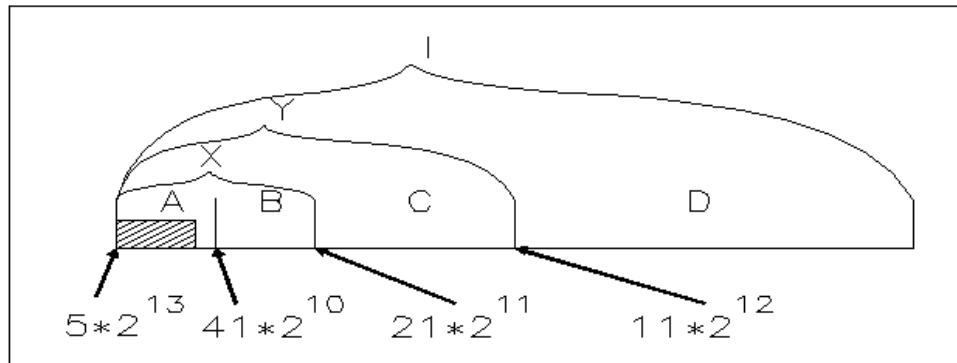
- Se determină cel mai mic număr  $p$ , cu  $m \leq p \leq n$ , pentru care numărul  $o$  de octeți solicitați verifică:  $o \leq 2^p$
- Se caută, în această ordine, în listele de ordin  $p, p+1, p+2, \dots, n$  o zonă liberă de dimensiune cel puțin  $o$ .
- Dacă se găsește o zonă de ordin  $p$ , atunci aceasta este alocată și se șterge din lista de ordinul  $p$ .
- Dacă se găsește o zonă de ordin  $k > p$ , atunci se alocă primii  $2^p$  octeți, se șterge zona din lista de ordin  $k$  și se crează în schimb alte  $k-p$  zone libere, având dimensiunile:

$$2^p, 2^{p+1}, \dots, 2^{k-1}$$

In fig. 10.20 este dat un astfel de exemplu. Se dorește alocarea a **1000** octeți, deci  $p = 10$ . Nu s-au găsit zone libere nici de dimensiune  $2^{10}$ , nici  $2^{11}$  și nici  $2^{12}$ . Prima zonă liberă de dimensiune  $2^{13}$  are adresa de început  $5 \cdot 2^{13}$  și am notat-o cu **I** în fig. 10.20. Ca rezultat al alocării a fost ocupată zona **A** de dimensiune  $2^{10}$  și au fost create încă trei zone libere: **B** de dimensiune  $2^{10}$ , **C** de dimensiune  $2^{11}$  și **D** de dimensiune  $2^{12}$ . Zonele **B**, **C**, **D** se trec respectiv în listele de ordine **10**, **11** și **12**, iar zona **I** se șterge 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 și formează împreună o zonă liberă de dimensiune  $2^{p+1}$ . Atât zona eliberată cât și camaradul ei se șterg 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, măbind 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 și zona *B*. În conformitate cu pașii descriși mai sus, se execută următoarele acțiuni:

- Se trece zona *B* în lista de ordin **10**.
- Se depistează că zonele *A* și *B* sunt camarazi. Drept urmare, cele două zone sunt comasate și formează o nouă zonă *X*. Zona *X* se trece în lista de ordin **11**, iar zonele *A* și *B* se șterg din lista de ordin **10**.
- Se depistează că zonele *X* și *C* sunt camarazi. Drept urmare, ele sunt comasate și formează o zonă *Y* care se trece în lista de ordin **12**, înlocuind zonele *X* și *C* din lista de ordin **11**.
- În sfârșit, se depistează că *Y* și *D* sunt camarazi. Ele sunt șterse 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 știe 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 poziția *k* (numerotarea începând cu 0) și ambele se termină prin *k* zerouri. Spre exemplu, adresele zonelor *A* și *B* din fig. 10.20 sunt:

```
A: 10100000000000000000
B: 10100100000000000000
    ^
```

Evident, astfel de teste se fac deosebit de ușor folosind instrucțiunile mașină care operează asupra biților.

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 și numai din rațiuni tehnologice. În orice caz, trebuie necondiționat să se aibă în vedere și concluziile, mai mult sau mai puțin empirice, rezultate din experiență. Una dintre acestea spune că nu întotdeauna o metodă mai sofisticată este și mai bună!

#### **10.4.3 Politici de încărcare.**

La sistemele cu alocare paginată, în momentul lansării în execuție 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 funcționa normal un timp, după care va cere din nou o pagină care nu este în memorie etc. Întrebarea

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

O soluție simplă, dar evident ineficientă, este *încărcarea la început a tuturor paginilor*. Prin aceasta va dispărea însuși 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, și ea este într-adevăr și 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 și câteva pagini vecine, în ipoteza că ele vor fi invocate în viitorul apropiat. O evidență 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 vecinătății*: 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 stabilește o așa 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. În conformitate cu principiul vecinătății, este foarte probabil ca următoarele referiri să fie făcute în cadrul memoriei de lucru.

#### **10.4.4 Politici de înlocuire.**

În mod natural, numărul total al paginilor programelor active poate deveni mai mare decât numărul paginilor fizice din memoria operativă. Din această cauză, uneori apare situația că un program cere încărcarea unei pagini virtuale, dar nu există o pagină fizică disponibilă pentru a o găzdui. Întrebarea este *care* dintre paginile fizice va fi evacuată pentru a crea spațiul 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 evoluția unui program la un moment dat este dependentă de datele concrete asupra cărora operează. Totuși, Belady [4] a descris un model de evidență 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 întreținerea unei structuri de date care să permită decizia trebuie făcută *la fiecare acces la memorie*. În această situație, nu este permisă nici măcar întreținerea unei liste simplu înlănțuite! 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 asociați doi biți, prin intermediul cărora se va decide pagina de evacuat. Bitul **R**, numit bit de *referire*, primește 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 iarăși pe 0. Bitul **M**, numit bit de *modificare*, primește valoarea 0 la încărcarea paginii. El este modificat numai la scrierea în pagină, când i se dă valoarea 1. Acești doi biți împart în fiecare moment paginile fizice în patru clase:

0. clasa 0: pagini nereferite și 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 și modificate.

Atunci când o pagină trebuie înlocuită, pagina "victimă" se caută mai întâi în clasa 0, apoi în clasa 1, apoi în clasa 2 și în sfârșit în clasa 3. Dacă pagina de înlocuit este în clasa 1 sau clasa 3, conținutul ei va fi salvat pe disc înaintea înlocuirii. Acest algoritm simplu, deși 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 și se întreține 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 îmbunătățire ar fi combinarea algoritmilor NRU și FIFO, în sensul că se aplică întâi NRU, iar în cadrul aceleiași clase se aplică FIFO.

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

Si acum o curiozitate! Bunul simț ne spune că șansa ca o pagină să fie înlocuită scade pe măsură ce numărul de pagini fizice crește. Si totuși nu este așa! În [4] [22] este dat un contraexemplu, cunoscut sub numele de *anomalie lui Belady*, pe care îl ilustrăm în fig. 10.21a și 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

În fig. 10.21a este ilustrată evoluția 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 puțin folosită în ultimul timp) este un algoritm bun de înlocuire. El are la bază următoarea observație, reieșită (tot) din principiul vecinătății. O pagină care a fost solicitată mult de către ultimele instrucțiuni, va fi probabil solicitată mult și în continuare. Invers, o pagină solicitată puțin (sau deloc), va rămâne probabil tot așa pentru câteva instrucțiuni.

Problema este cum să se țină evidența utilizărilor? Se *exclude* din start întreținerea unei liste înlănțuite care să fie modificată la fiecare acces la memorie. Prețul 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 biți. La fiecare acces, valoarea lui este mărită cu o unitate. În tabela de pagini, există câte un spațiu rezervat pentru a memora valoarea contorului. În momentul accesului la o pagină, valoarea contorului este memorată în acest spațiu rezervat din tabela de pagini. Atunci când se impune o înlocuire, este înlocuită pagina care a reținut cea mai mică valoare a contorului.

*Matricea de referințe.* Pentru un sistem de calcul care are  $n$  pagini fizice, se utilizează o matrice binară de  $n \times n$ . 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. În 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 evoluția matricei de referințe 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** Evoluția unei matrice de referințe

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

Desigur, politicile de înlocuire descrise mai sus pot fi simulate foarte bine prin soft. Din păcate, eficiența mecanismelor scade drastic. Acesta este motivul pentru care unele implementări ale memoriei virtuale (nu dăm aici nume) au eșuat lamentabil. Nu-i nimic, s-a câștigat experiență și asta nu este lucru puțin.

#### 10.4.5 Cum funcționează o memorie cache?

Apariția memoriei *cache* a fost dictată de necesitatea creșterii performanțelor sistemului de calcul. Memoria cache conține 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 crește.

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? În lucrarea [42] se dau răspunsuri la aceste întrebări. Tot acolo se fac convenite evaluări ale raportului cost/performanță pentru dimensionarea unei memorii cache, precum și 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 corespondența dintre memoria operativă și memoria cache, precum și politicile de schimb dintre cele două tipuri de memorie.

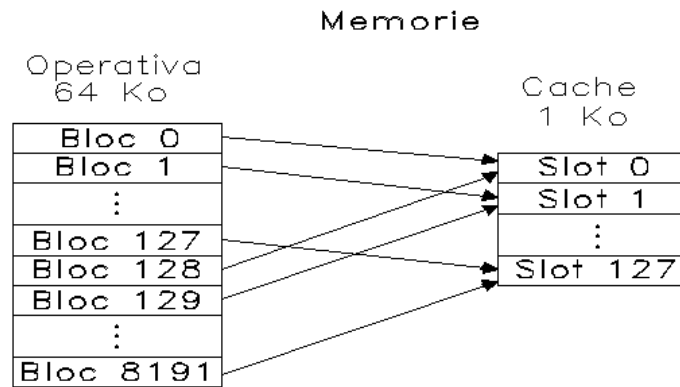
Memoria cache este împărțită în mai multe părți egale, numite *sloturi* (poziții). Un slot are dimensiunea unui bloc de memorie, care este în mod obligatoriu o putere a lui 2. Fiecare slot conține în plus câțiva biți (numiți generic *tag* = etichetă) care indică blocul de memorie operativă deplasat în 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 și 10.24 am ales o memorie operativă de 64 Ko și o memorie cache de 1 Ko. Dimensiunea unui slot, identică cu dimensiunea unui bloc de memorie, am ales-o de 8 octeți (64 de biți).

Se cunosc [42] mai multe metode de *proiectare a spațiului 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:

$$S = A \bmod C$$

În fig. 10.23 este ilustrată această corespondență.





**Figura 10.23** Proiectare directă pe memoria cache

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

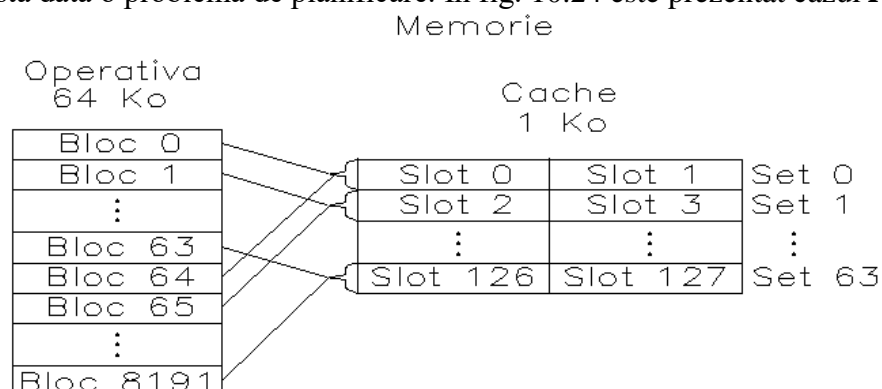
O a doua metodă poartă numele de *proiectare asociativă*, apărută pentru eliminarea dezavantajului de mai sus. Această metodă plasează un bloc de memorie într-un slot oarecare liber.

Gestiunea sloturilor memoriei cache în acest caz ridică aceleași probleme ca și alocarea paginată a memoriei. Principala problemă care se ridică aici este cea legată de *politica de înlocuire*. În paragraful 2.4.3 am tratat această problemă, descriind algoritmi NRU, FIFO și LRU. Tot ceea ce s-a spus acolo este valabil, fără modificări și 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*, și ea este o combinație a precedentelor două metode. Ideea ei este următoarea. Memoria cache este împărțită în  $I$  seturi, un set fiind compus din  $J$  sloturi. Avem deci relația  $C = I \cdot J$ . 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. În fig. 10.24 este prezentat cazul  $I=64$  și  $J=2$ .



**Figura 10.24** Proiectarea set - asociativă

Metoda set-asociativă este mult folosită în practică. Compromisul proiectării directe și 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. În rest se aplică proiectarea directă, care este mult mai simplă.