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.

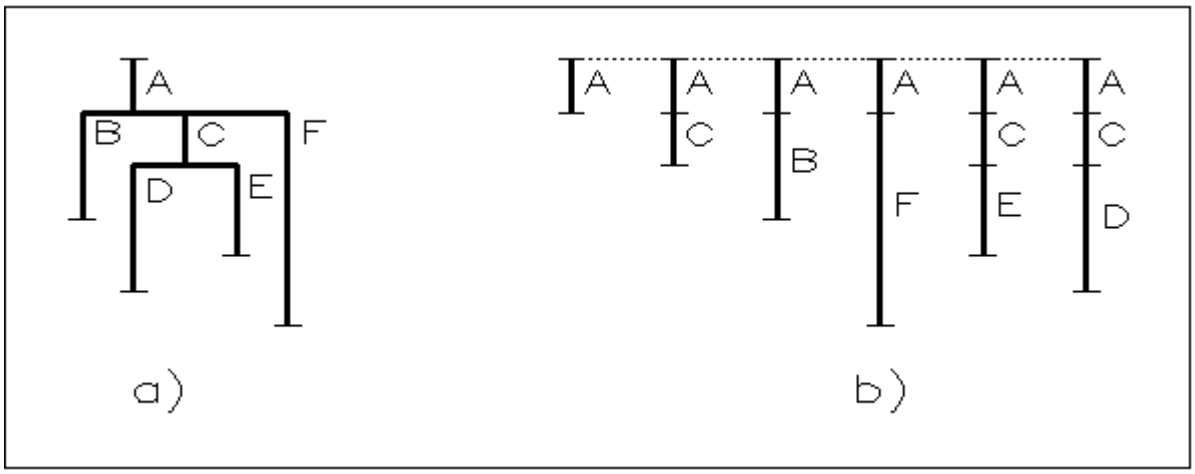

Figura 10.1 Instanţe posibile ale unui program segmentat

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.

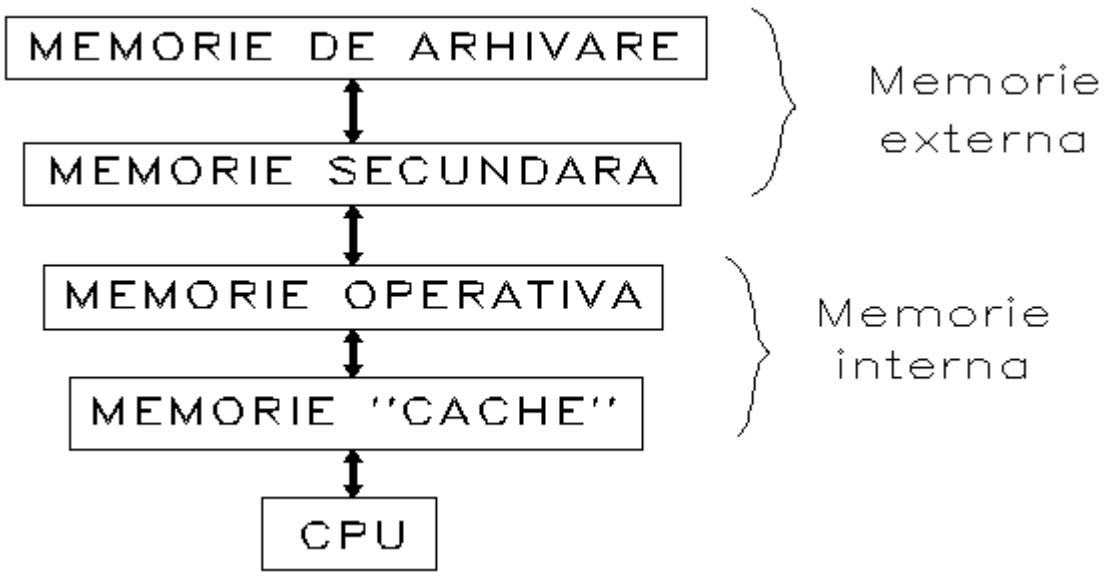

Figura 10.2 Structura memoriei unui sistem de calcul

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.

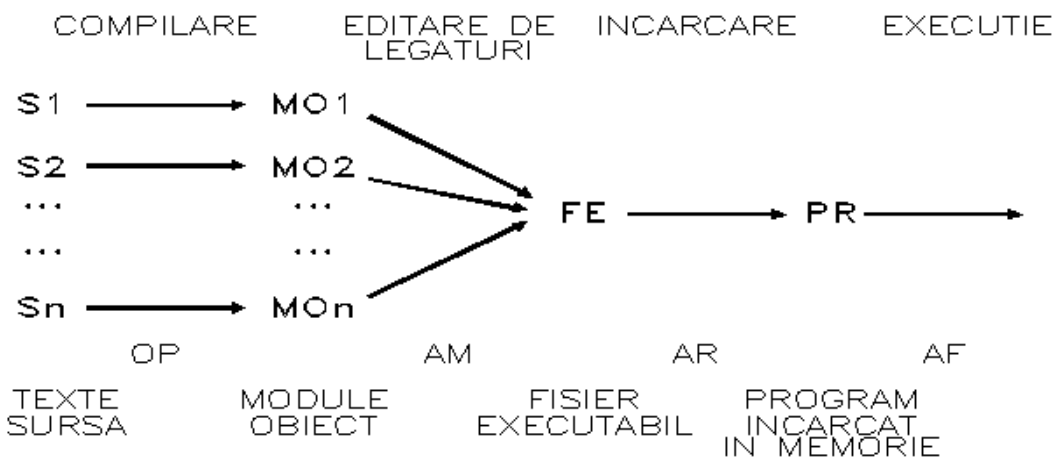

Figura 10.3 Fazele translatării unui program

10.1.3.1 Faza de compilare

Faza de *compilare* transformă un text sursă ***Si*** într-un *modul obiect* ***MOi***. 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.

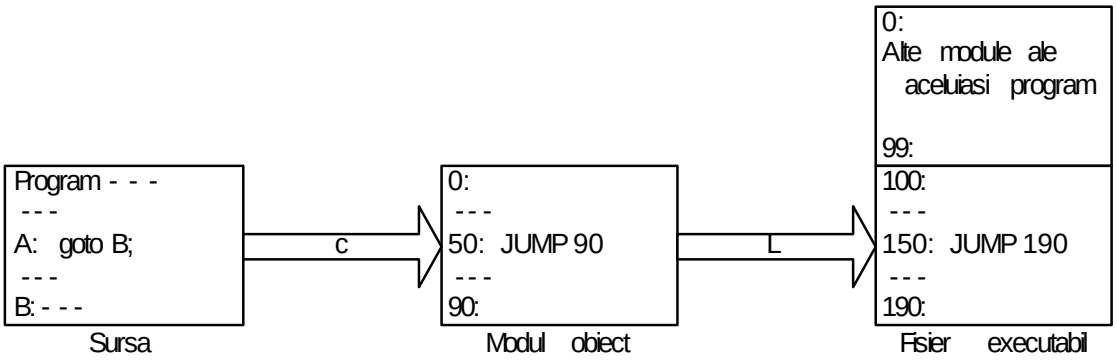

Figura 10.4 Translatare de la sursă la fişier executabil

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

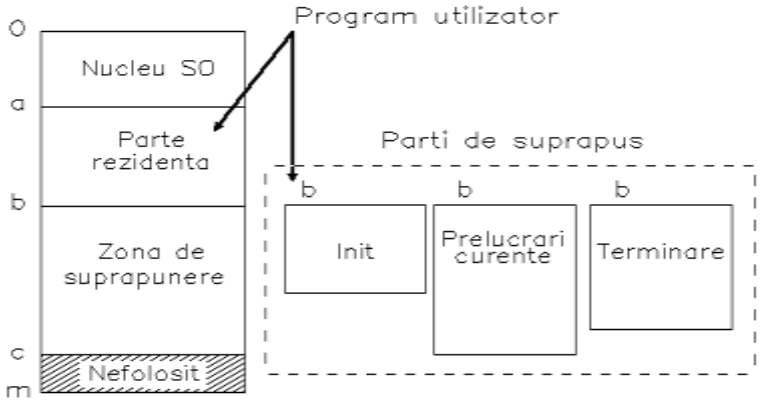

Figura 10.7 Alocarea memoriei la sistemele monoutilizator

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.

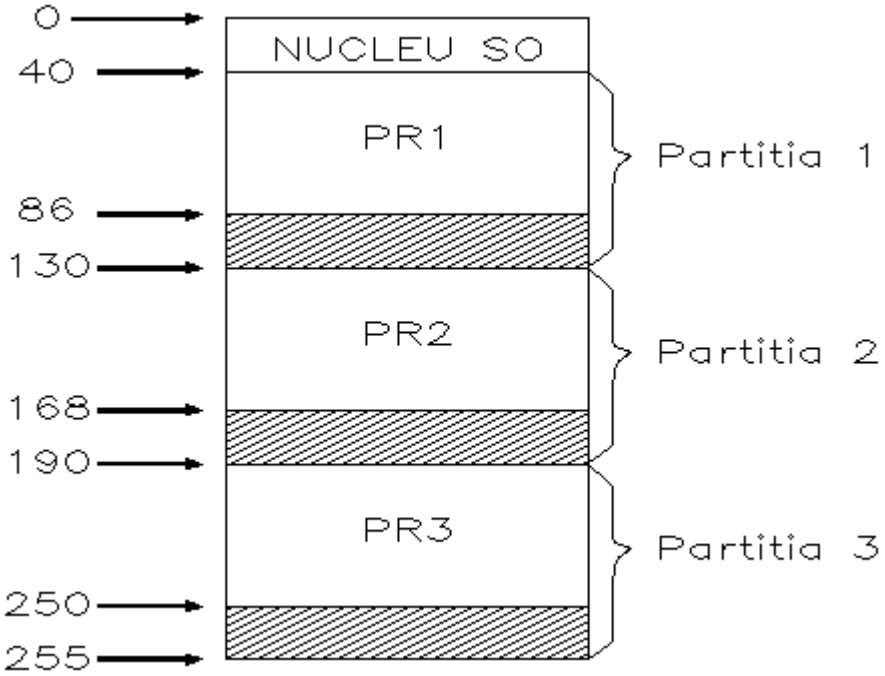

Figura 10.8 Exemplu de alocare cu partiţii fixe

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.

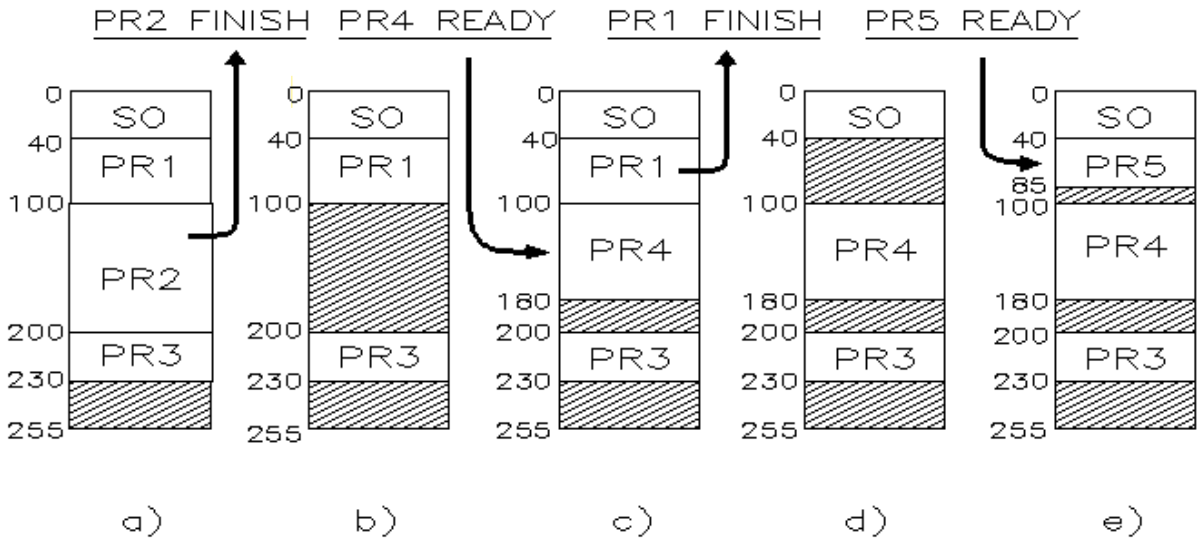

Figura 10.9 Evoluţia proceselor la alocarea cu partiţii variabile

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.

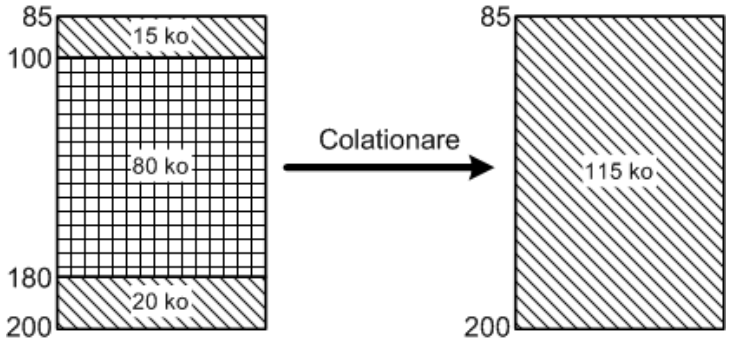

Figura 10.10 Colaţionarea de spaţii libere vecine

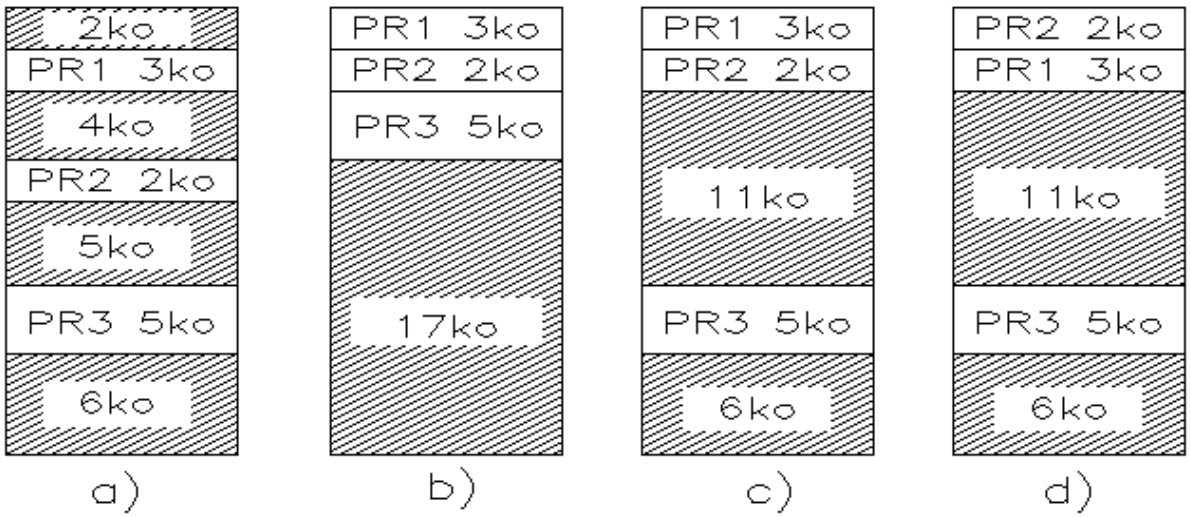

Figura 10.11 Posibilităţi de compactare prin relocare totală sau parţială

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.

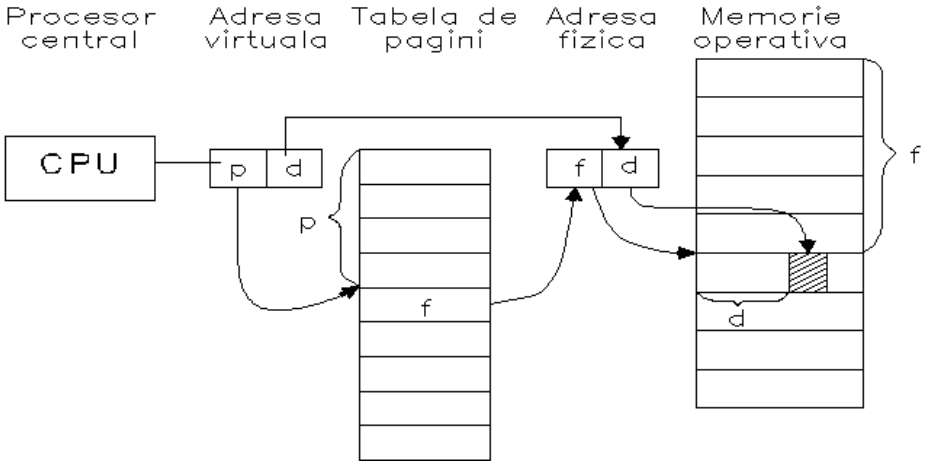

Figura 10.12 Translatarea unei pagini virtuale într-una fizică

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.

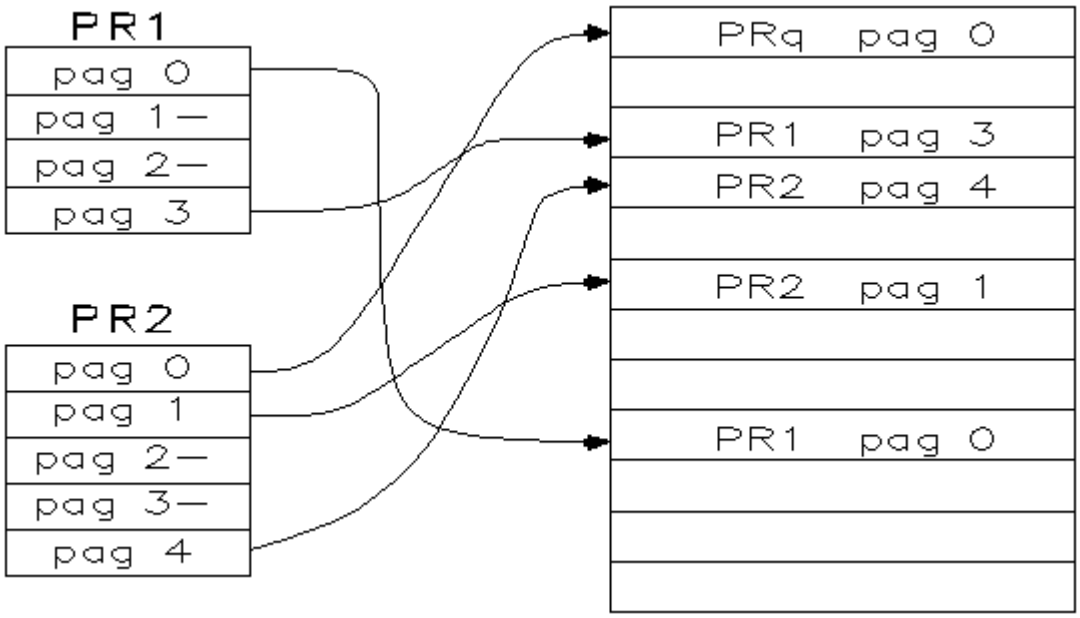

Figura 10.13 Două procese, într-o alocare paginată

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.

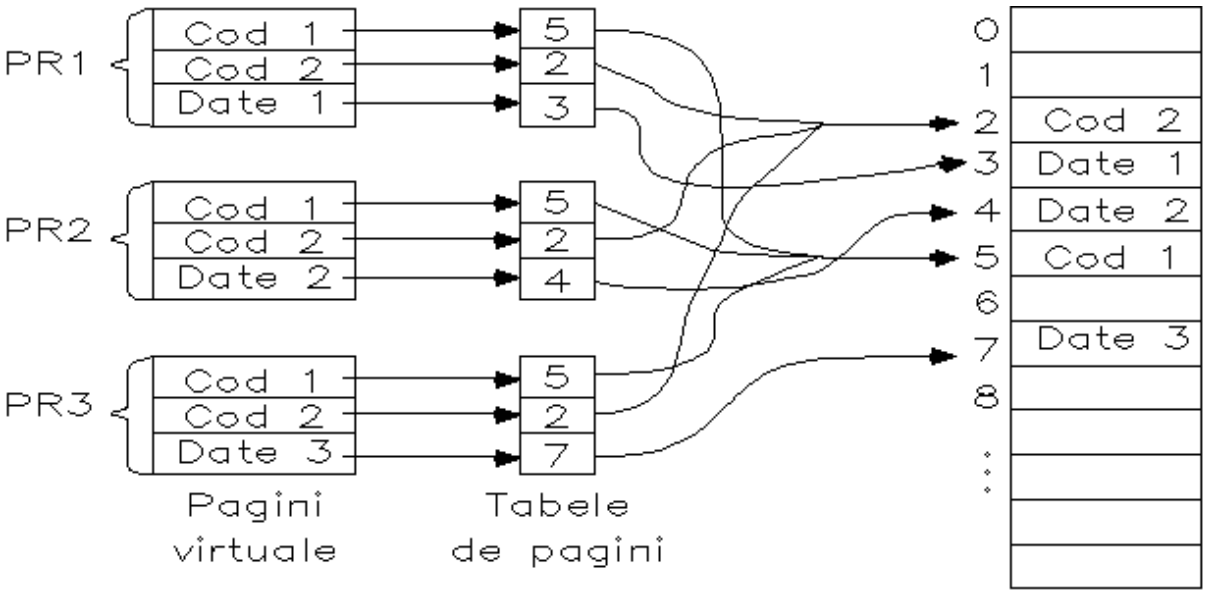

Figura 10.14 Folosirea în comun a unui cod

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].

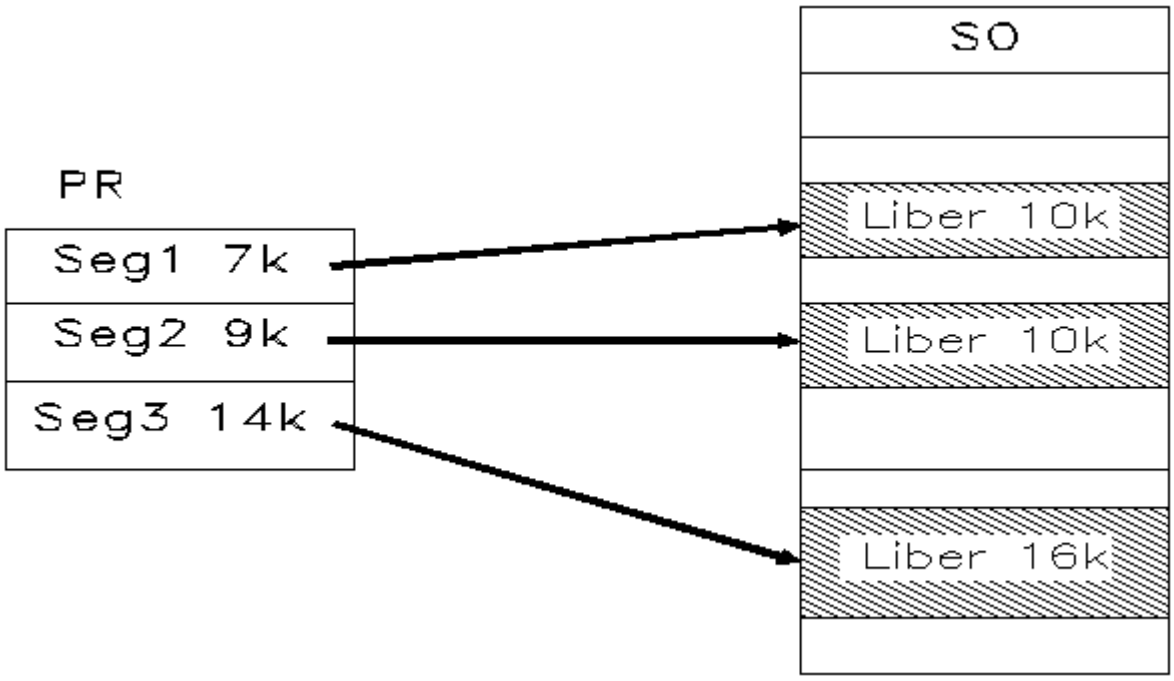

Figura 10.15 Alocare necontiguă prin segmentare

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.

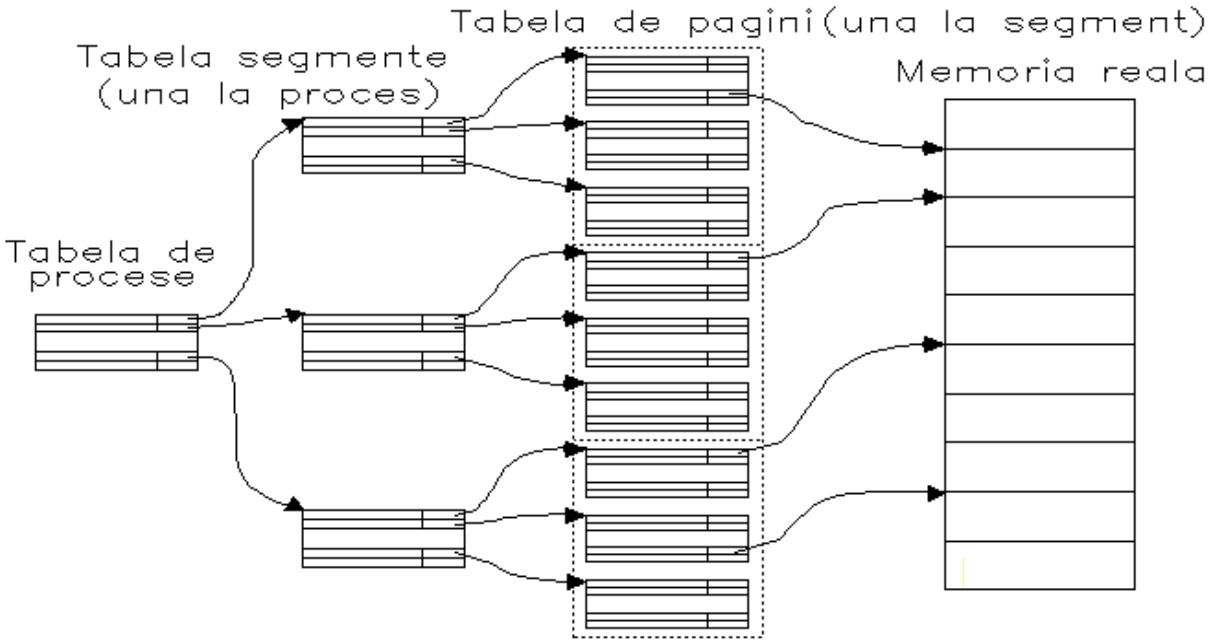

Figura 10.16 Alocare segmentată şi paginată

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.

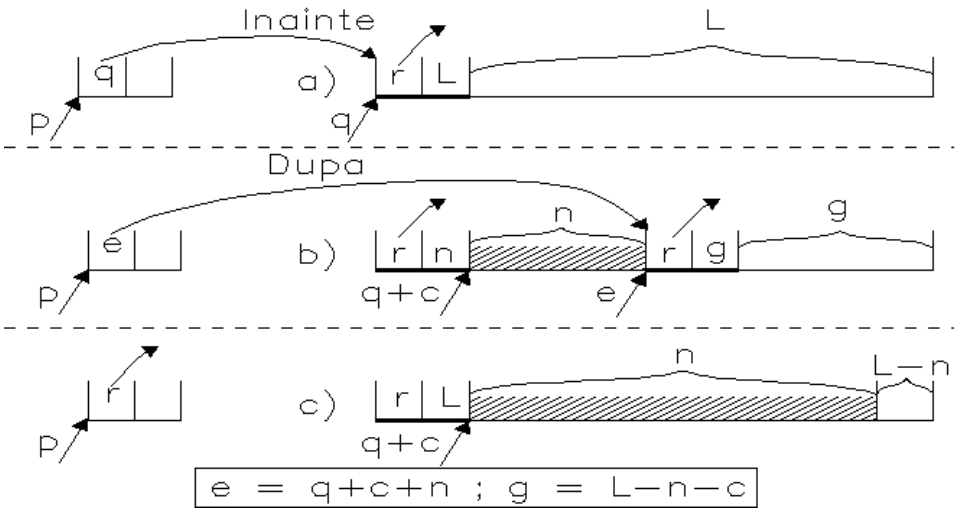

Figura 10.17 Alocarea de octeţi într-o zonă liberă

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.

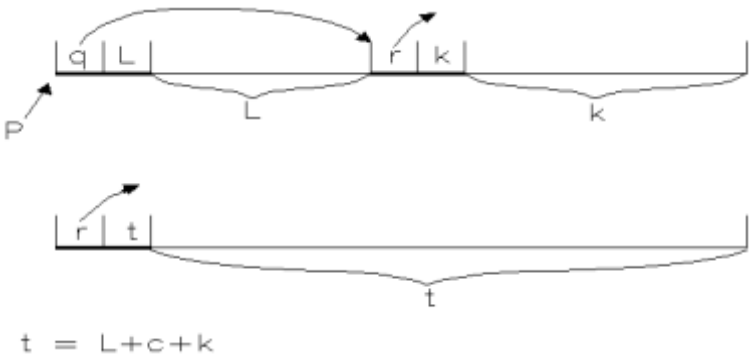

Figura 10.18 Comasarea a două zone libere adiacente

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.

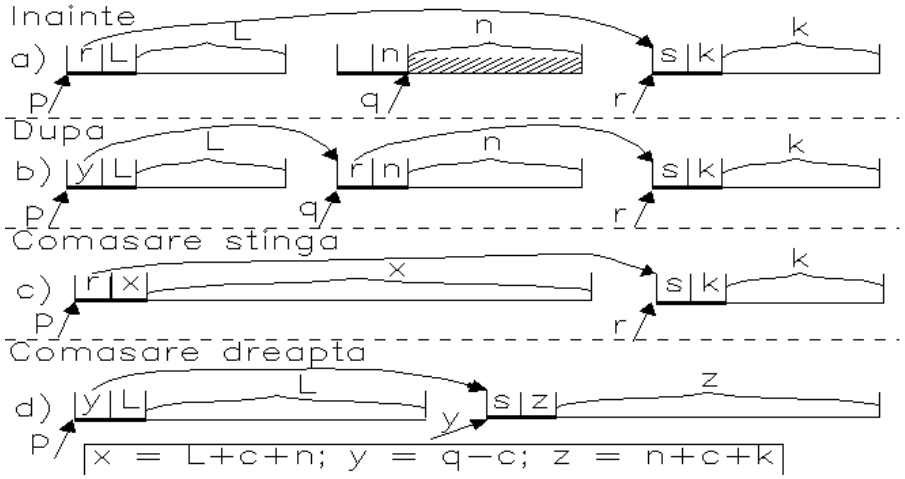

Figura 10.19 Eliberarea unei zone ocupate

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 duală 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 şi A1 mod 2^(k+1) = 0

sau

A2 < A1 , A1 = A2 + 2^k şi A2 mod 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.

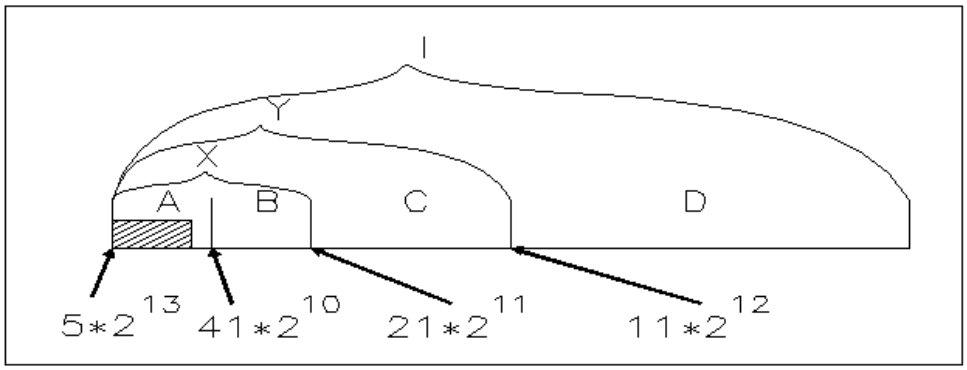

Figura 10.20 Alocare în sistem Buddy

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:  1010000000000000000
B:  1010010000000000000
          ^
```

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.

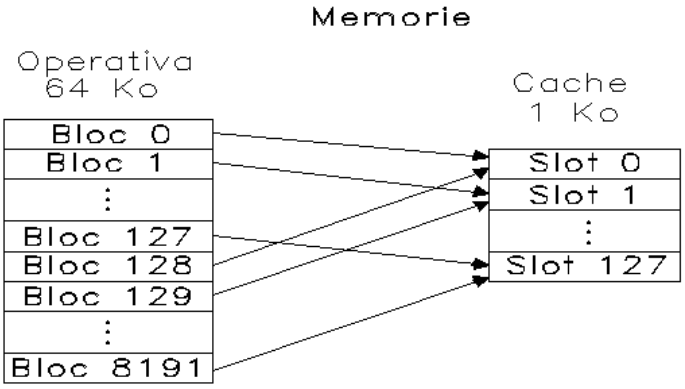

Figura 10.23 Proiectare directă pe memoria cache

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**.

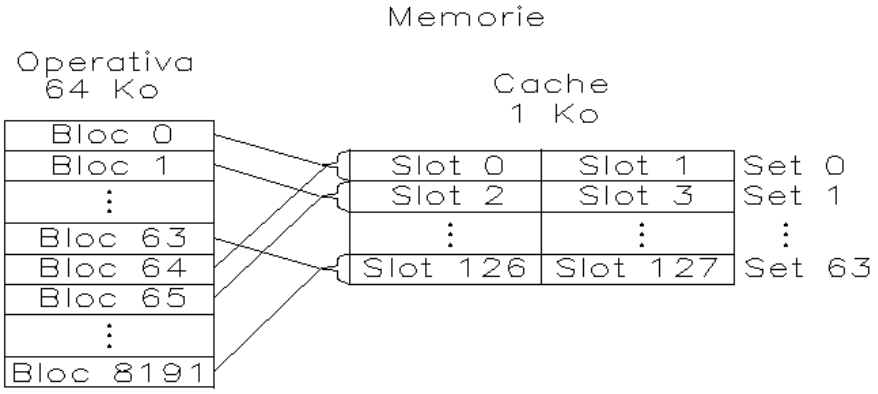

Figura 10.24 Proiectarea set - asociativă

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.