4. Administrarea proceselor

Acest capitol este consacrat implementării noţiunilor *proces* şi *sincronizare* în cadrul unui sistem de operare. Mecanismele utilizate sunt bazate pe realizarea principiul excluderii mutuale, studiat în 4.1. Principiile directoare ale reprezentării şi gestiunii proceselor (contexte, alocarea procesorului) sunt prezentate în 4.2 şi ilustrate în 4.3 printr-un exemplu schematic de realizare a unui nucleu de sincronizare.

4.1. Realizarea excluderii mutuale

Mecanismele care realizează excluderea mutuală pentru un set de programe sunt bazate pe un principiu comun: utilizarea mecanismului de excludere mutuală existent deja la un nivel inferior. Drept rezultat, sunt utilizate variabile comune ale proceselor concurente, iar coerenţa acestor variabile trebuie ea însăşi să fie garantată. La nivelul de bază (cel al resurselor fizice) există două mecanisme elementare: excluderea mutuală prin accesarea unui amplasament de memorie (variabila de blocare sau eveniment memorizat) şi masca întreruperilor. Aceste două mecanisme sunt, în principiu, suficiente pentru toate necesităţile. Dar, din considerente de eficacitate, la nivelul resurselor fizice sau microprogramelor există dispozitive mai sofisticate, cum ar fi instrucţiunea *Test and Set* sau *semafoarele.*

4.1.1. Specificarea problemei

Vom preciza mai întâi problema excluderii mutuale. Fie $\{p_1, p_2,...,p_n\}$ o mulţime de procese pe care le vom considera ciclice; programul fiecărui proces conţine o secţiune critică. Excluderea mutuală este asigurată prin două fragmente de program (prolog şi epilog), care încadrează secţiunea critică a fiecărui proces. Presupunem, că fiecare proces, care intră în secţiunea critică o părăseşte într-un interval de timp finit.

Soluţia trebuie să posede următoarele proprietăţi:

a) excludere mutuală: la fiecare moment de timp cel mult un proces execută secţiunea critică,

b) absenţa blocajelor intempestive (care nu sunt la timpul lor): dacă în secţiunea critică nu se află vreun proces, nici un proces nu trebuie să fie blocat de mecanismul excluderii mutuale,

c) toleranţă la defecte: soluţia trebuie să rămână validă şi în cazul unor defecte în unul sau în mai multe procese, care se află în afara secţiunii critice,

d) absenţa privaţiunilor: un proces, care a cerut intrarea într-o secţiune critică nu trebuie să fie privat de acest drept (adică, să aştepte un timp infinit (presupunând, că toate procesele au aceeaşi prioritate)),

e) simetrie: prologul şi epilogul trebuie să fie identice pentru toate procesele şi independente de numărul lor.

Ţinând cont de aceste specificaţii vom construi o soluţie de forma:

```
<iniţializare>                          -- comună tuturor proceselor
        <programul procesului pi>:
            ciclu
                <prolog>                -- intrare în secţiunea critică
                <secţiunea critică>
                <epilog>                -- ieşire din secţiunea critică
                <restul programului>
            endciclu
```

Trebuie să elaborăm fragmentele *iniţializare, prolog* şi *epilog*.

4.1.2. Excluderea mutuală prin aşteptare activă

Înainte de a descrie implementarea excluderii mutuale prin operaţii elementare de blocare şi deblocare a proceselor prezentăm un mecanism, care permite simularea efectului acestor operaţii, menţinând procesele în stare activă. Un proces în aşteptare activă simulează blocarea efectuând o testare repetată a condiţiei de depăşire, care poate fi actualizată de alte procese.

4.1.2.1. Algoritmul lui Dekker

Pentru început considerăm cazul a două procese p_0 şi p_1. O primă abordare constă în reprezentarea condiţiei de aşteptare (care este complementară condiţiei de depăşire) printr-o variabilă booleană *c*; vom avea atunci:

c = true când "un proces este în secţiunea critică", *false* – în caz contrar.

Putem propune următorul program:

iniţializare: c := false;

```
prolog    : test: if c = false then
                      go to test
                  else
                      c := true
                  endif;
        <secţiunea critică>
epilog    : c := false;
```

La o analiză mai atentă observăm (v.3.2.2.3), că dacă nu vom face nişte ipoteze suplimentare, acest program poate să nu rezolve problema pusă. Iată secvenţa de intrare în secţiunea critică, descompusă în instrucţiuni:

		procesul p_0				procesul p_1	
1)	*test0:*	**load** R_0	*c*	1^1)	*test1:*	**load** R_1	*c*
2)		**br** (R_0=*true*)	*test0*	2^1)		**br** (R_1=*true*)	*test1*
3)		**stz**	*c(=true)*	3^1)		**stz**	*c*

(**br** este operaţia de salt condiţionat iar **stz** pune în *true* amplasamentul lui *c*). Dacă ordinea de execuţie este *1, 1^1, 2, 2^1, 3, 3^1* vedem că excluderea mutuală este greşită. Problema provine de la faptul că procesul p_1 poate consulta *c* între momentul când p_0 a consultat *c* (găsindu-l *false*) şi momentul în care p_0 l-a pus pe *c* în *true*. Altfel spus, este necesar ca secvenţele de acţiuni *(1, 2, 3)* şi *(1^1, 2^1, 3^1)* să fie executate în mod atomar. Anume acest principiu stă la baza instrucţiunii *Test and Set.*

O soluţie (algoritmul lui Dekker) poate totuşi fi construită fără a folosi alte mecanisme de excludere mutuală, în afară de indivizibilitatea accesării în citire sau actualizarea unui amplasament de memorie. Prezentăm algoritmul pentru două procese, deşi el poate fi extins pentru un număr arbitrar de procese.

Programul foloseşte trei variabile comune celor două procese:

```
          var  c    :  array [0..1] of boolean;
               tur  :  0..1;
iniţializare: c[0]:=c[1]:=false;
               tur:=0;
prolog    :                          -- pentru procesul i; se va pune j=1-i (celălalt proces)
               c[i]:=true;
               tur:=j;
          test: if c[j] and tur=j then
                    go to test
               endif;
          ...
epilog    :                          -- pentru procesul i
               c[i]:=false;
```

Această soluţie, demonstrarea validităţii căreia este propusă ca exerciţiu, prezintă un interes pur teoretic. În practică sunt utilizate instrucţiuni speciale care asigură indivizibilitatea secvenţelor de testare şi modificare.

Aşteptarea activă poate fi în egală măsură utilizată ca mecanism elementar de sincronizare în cazul unor alte probleme, diferite de excluderea mutuală.

Exemplul 4.1. Reluăm problema exemplului din 3.3.1 (comunicarea a două procese printr-un segment comun). Reprezentăm printr-o variabilă booleană *c* condiţia de aşteptare:

*c="*procesul *p* a terminat scrierea în segmentul *a"*

Valoarea iniţială *c=false.* Programele se vor scrie astfel:

```
procesul p                          procesul q
                                    ...          ...
scriere(a);                    test: if c then
c:=true                                 go to test
                                    endif;
                                    citire(a)
```

Putem verifica, că soluţia este corectă: oricare ar fi ordinea de executare a acestor două procese, deblocarea lui *q* este, în cel mai rău caz, retardată cu un ciclu de aşteptare activă. Printre altele, această proprietate rămâne adevărată dacă vom presupune, că mai multe procese $q_1, q_2,..., q_n$ aşteaptă terminarea operaţiei de scriere. ◄

Proprietatea, care asigură validitatea schemei de mai sus, constă în faptul că modificarea variabilei *c* este efectuată de către un *singur* proces. Notăm, că această schemă a fost deja întâlnită în capitolul 2 la administrarea intrărilor-ieşirilor la CDC 6600: un cuplu de procese comunică prin intermediul unei perechi de indicatori, fiecare dintre care este

actualizat de un proces şi consultat de altul. Pentru aceasta este necesar să se poată executa în excludere mutuală instrucţiunile de testare şi modificare.

4.1.2.2. Aşteptarea activă în sistemele multiprocesorale: Test & Set

Pentru tratarea cu ajutorul aşteptării active a cazului în care mai multe procese actualizează şi consultă variabile comune, unele maşini au o instrucţiune, care realizează într-o manieră indivizibilă consultarea şi actualizarea unui amplasament de memorie. Această instrucţiune, adesea numită *Test And Set* (**tas**), este utilizată în sistemele multiprocesorale (în sistemele monoprocesor mascarea întreruperilor este suficientă pentru asigurarea excluderii mutuale).

Fie *m* adresa amplasamentului de memorie considerat, sau **lacătul**, iar *R* un registru al procesorului. Prin convenţie, dacă lacătul este în *0* (în locaţiunea cu adresa *m* avem *0*), secţiunea critică este liberă, iar dacă este *1* – ea este ocupată. Efectul lui *Test And Set* este descris mai jos (*Mp[m]* desemnează conținutul amplasamentului de memorie cu adresa *m*):

```
tas R, m : <blocare acces la Mp[m]>
           R:=Mp[m]
             Mp[m]:=1
           <eliberare acces la Mp[m]>
```

Excluderea mutuală prin aşteptare activă poate fi programată cu ajutorul următoarelor secvenţe:

```
iniţializare : stz     m          -- Mp[m]:=0
prolog       : tas     R, m
               br(R≠0) $-1        -- test iterat
               secţiunea critică
epilog       : stz     m
```

4.1.3. Semaforul – instrument elementar pentru excluderea mutuală

4.1.3.1. Definiţii

Un **semafor** *s* este constituit prin asocierea unui contor cu valori întregi, notat *c.s.*, şi a unui fir de aşteptare, notat *f.s.* La crearea semaforului contorului i se atribuie o valoare iniţială s_0 ($s_0 \geq 0$), şi firul de aşteptare *f.s.* este vid. Un semafor serveşte la blocarea proceselor aşteptând să se producă o condiţie pentru deblocarea lor. Procesele blocate sunt plasate în *f.s.* Mai multe procese pot fi sincronizate prin semafoare, care aparţin părţii comune a contextului lor. Un procesor poate fi manipulat doar cu ajutorul a două operaţii *P(s)* şi *V(s)*, numite primitive. Valoarea contorului şi starea firului de aşteptare sunt inaccesibile, chiar şi pentru citire. Fie *p* un proces care execută *P(s)* sau *V(s)*, iar *q* un proces care se află în firul de aşteptare *f.s.* Algoritmul primitivelor este următorul:

```
P(s):                          V(s):

c.s.:=c.s.-1;                  c.s.:=c.s.+1;
if c.s.<0 then                 if c.s.≤0 then
    stare(p):=blocat;              extragere(q,f.s.);
    introducere(p,f.s.)            stare(q):=activ
endif                          endif
```

Aceste operaţii sunt executate în excludere mutuală. Modalitatea de realizare efectivă a semafoarelor şi a primitivelor *P* şi *V* este descrisă în 4.3. Operaţiile *introducere* şi *extragere* permit inserarea unui proces într-un fir de aşteptare sau, respectiv, extragerea. Nu facem aici nici un fel de ipoteze despre politica de gestionare a firului de aşteptare: algoritmii de sincronizare trebuie să fie independenţi. Politica aleasă în cazul alocării unui procesor este discutată în 4.3, la fel şi detaliile legate de operaţiile de blocare şi deblocare.

Exemplul 4.2. Funcţionarea unui semafor poate fi comparată cu lucrul unui magazin accesul în care este admis doar dacă la intrare există coşuri libere (sau cărucioare) în care cumpărătorii îşi duc marfa (intrarea fără coş este interzisă). La deschiderea magazinului un anumit număr de coşuri libere sunt puse la dispoziţia cumpărătorilor la intrare. Un cumpărător ia un coş liber, intră în magazin şi îşi alege marfa, iar dacă la intrare nu există un coş liber, cumpărătorul este obligat să aştepte într-un fir de aşteptare (operaţia *P*). La ieşire cumpărătorul pune coşul de unde l-a luat (operaţia *V*). ◄

Doar executarea primitivei *P* poate bloca un proces. Acesta va putea fi deblocat doar de un alt proces, care a executat primitiva *V* pe acelaşi semafor. Executarea operaţiei *V* nu este blocantă.

4.1.3.2. Proprietăţi

Proprietăţile principale ale sincronizării cu ajutorul semafoarelor pot fi deduse din câteva relaţii invariante: relaţii verificate iniţial şi care rămân neschimbate după executarea primitivelor *P* şi *V* un număr arbitrar de ori.

1) Fie, pentru un semafor *s*:
   $n_p(s)$ – numărul total de execuţii a operaţiei *P(s)*,

$n_v(s)$ – numărul total de execuţii a operaţiei *V(s).*

Are loc relaţia:

$$c.s. = s_0 - n_p(s) + n_v(s) \qquad (1)$$

deoarece valoarea iniţială a lui *c.s.* este s_0, fiecare operaţie *P(s)* scade din această valoare o unitate, iar *V(s)* adaugă *1*. Aceste operaţii, fiind executate în excludere mutuală, nici o modificare nu este pierdută.

2) Fie $n_{bloc}(s)$ numărul proceselor blocate în *f.s.* Are loc relaţia:

$$n_{bloc}(s) = \textbf{if } c.s. \geq 0 \textbf{ then } 0 \textbf{ else } -c.s. \textbf{ endif} \qquad (2)$$

care poate fi de asemenea scrisă

$$n_{bloc}(s) = max(0, -c.s.) \qquad (2^1)$$

Relaţia (2) iniţial este adevărată. Tabelul de mai jos, care indică efectul operaţiilor *P(s)* şi *V(s)* asupra valorii variabilei n_{bloc}, arată că aceste operaţii lasă relaţia (2) invariantă.

	c.s.<0	*c.s.=0*	*c.s.>0*
Efectul operaţiei *P(s)*	*+1*	*+1*	--
Efectul operaţiei *V(s)*	*-1*	--	--

3) Relaţia (2) poate fi scrisă sub o altă formă, care ne va fi utilă mai departe. Fie $n_f(s)$ numărul de "treceri" de către procese a primitivei *P(s)*, adică suma numărului de executări a lui *P(s)* fără blocare şi a numărului de deblocări realizate de către *V(s)*. Vom avea în acest caz:

$$n_{bloc}(s) = n_p(s) - n_f(s).$$

Introducând această valoare în (2^1) obţinem:

$$- n_f(s) = max(-n_p(s), -c.s.-n_p(s)), \text{ sau}$$

$$n_f(s) = min(n_p(s), c.s.+n_p(s)).$$

În fine, utilizând valoarea lui *c.s.* din (1), avem:

$$n_f(s) = min(n_p(s), c.s.+n_v(s)). \qquad (3)$$

4.1.3.3. Realizarea excluderii mutuale cu ajutorul semafoarelor

Prezentăm o schemă, care rezolvă problema excluderii mutuale pentru *n* procese. În cazul în care nu se fac ipoteze speciale despre gestionarea firului de aşteptare, nu se garantează lipsa privaţiunilor.

iniţializare : **semafor** *mutex* **init** *1*
prolog : *P(mutex)*
secţiunea critică
epilog : *V(mutex).*

Să ne convingem, că soluţia prezentată este în conformitate cu specificaţiile din 4.1.1. Trebuie să verificăm dacă proprietăţile *a*, *b* şi *c* au loc.

Fie n_c numărul de procese, care se află în secţiunea critică la un moment concret de timp. Avem:

$$n_c = n_f(mutex) - n_v(mutex) \qquad (4)$$

Proprietăţile în cauză pot fi verificate aplicând semaforului *mutex* relaţia (3) din 4.1.3.2:

$$n_f(mutex) = min(n_p(mutex), 1+n_v(mutex)) \qquad (5)$$

a) *Excluderea mutuală*

Din (5) avem:

$$n_f(mutex) \leq 1+n_v(mutex)$$

şi, utilizând (4), obţinem $\boldsymbol{n_c \leq 1}$: excluderea mutuală este asigurată.

b) *Absenţa blocajelor*

Presupunem, că nici un proces nu se află în secţiunea critică. Vom avea în acest caz $n_c = 0$, sau

$$n_f(mutex) = n_v(mutex) \text{ sau încă}$$

$$n_f(mutex) \leq 1+n_v(mutex)$$

Vom avea conform relaţiei (5):

$$n_f(mutex) = n_p(mutex) \text{ sau}$$

$$n_{bloc}(mutex) = 0$$

Deci, proprietatea *b* este prezentă, ca şi proprietatea *c*, deoarece nu s-a făcut nici o ipoteză despre starea proceselor, dacă ele nu se află în secţiunea lor critică.

1). Secţiuni critice incorporate. Blocaje

Vom considera două procese *p* şi *q* pentru care programul conţine două secţiuni critice distincte, corespunzătoare utilizării a două resurse critice distincte.

procesul *p*	procesul *q*
...	...
(1) *P(mutex1)*	(1^1) *P(mutex2)*
...	...
(2) *P(mutex2)*	(2^1) *P(mutex1)*

...	...
V(mutex2)	*V(mutex1)*
...	...
V(mutex1)	*V(mutex2)*

Dacă traiectoria temporală de execuţie a proceselor *p* şi *q* începe cu (1, 1^1, 2, 2^1), se va ajunge la o situaţie în care ambele procese sunt blocate pentru un timp infinit, deoarece fiecare dintre procese poate fi deblocat doar de către celălalt. Această situaţie este numită **blocare** sau îmbrăţişare fatală (eng. deadlock, fr. étreinte fatale).

2). Aşteptare infinită în secţiunea critică sau impas

Validitatea soluţiei propuse se bazează pe presupunerea, că toate procesele părăsesc secţiunea critică în timp finit. Numai ce am stabilit, că această ipoteză poate fi infirmată dacă secţiunile critice se intersectează. Pot fi şi alte cauze, care conduc la o aşteptare infinită. Astfel, blocarea, incorectitudini sau ciclări infinite într-un proces, care se află în secţiunea critică, pot paraliza toate procesele concurente cu procesul dat. În cazul unor secţiuni critice globale (care prezintă interes pentru toţi utilizatorii), realizate pentru un sistem de operare, pot fi propuse următoarele soluţii:

- oricărui proces, care execută o secţiune critică globală, i se atribuie, pe toată durata acestei execuţii, un statut special, care îi conferă anumite drepturi particulare: prioritate înaltă, protecţie contra distrugerii, etc.
- un orologiu de gardă este armat la intrarea unui proces în secţiunea critică; dacă procesul nu părăseşte secţiunea critică după un interval de timp predefinit, sistemul de operare forţează ieşirea procesului şi eliberează astfel secţiunea critică. Această soluţie nu este cea mai bună, or datele globale, manipulate în secţiunea critică, pot să devină incoerente. Este necesar să se ofere posibilitatea restabilirii acestor date la o stare anterioară, considerată validă, ceea ce implică salvări periodice.

3). Privaţiune

Algoritmul excluderii mutuale garantează intrarea exact a unui proces în secţiunea critică, dacă mai multe procese încearcă acest lucru, când secţiunea critică este liberă. Se poate întâmpla ca un proces particular să fie reţinut pentru un interval de timp nedefinit: acest fenomen se numeşte **privaţiune**, pe care o vom reîntâlni atunci când vom studia alocarea resurselor. Întradevăr, procesele candidate la intrare, aşteaptă în firul de aşteptare *mutex.f* şi ordinea lor de deblocare (prin *V(mutex))* depinde de politica de gestionare a acestui fir, despre care nu s-a făcut nici o ipoteză.

Pentru cazul cel mai frecvent, când firele de aşteptare a semafoarelor sunt gestionate conform ordinii "prim sosit – prim servit" fără prioritate, riscul de privaţiune este eliminat. Întradevăr, dacă presupunem că execuţia unei secţiuni critice durează totdeauna un interval finit de timp, orice proces, după o perioadă finită de timp, va deveni cel mai vechi în firul său de aşteptare.

4.2. Funcţionarea şi structura unui nucleu de sincronizare

Noţiunea de proces şi operaţiile asociate nu fac, de obicei, parte din setul de bază al instrucţiunilor calculatorului. Ele vor fi implementate cu ajutorul unor programe şi/sau microprograme, care formează **nucleul** de administrare a proceselor. În cadrul descrierii unui sistem de operare cu ajutorul maşinilor abstracte ierarhice (v. cap.9), nucleul constituie nivelul cel mai inferior, realizat direct pe maşina fizică. Maşina abstractă, realizată astfel poate fi numită *o maşină a proceselor*, care posedă, în afara setului de instrucţiuni de bază, primitivele care permit crearea, distrugerea şi sincronizarea proceselor. Ca şi orice maşină abstractă, maşina realizată în acest mod ascunde unele proprietăţi ale maşinii fizice. Astfel:

- noţiunea de proces, care este echivalentă cu cea de proces virtual, ascunde utilizatorilor nucleului mecanismul de alocare a procesoarelor fizice. La un nivel superior nivelului nucleului chiar şi numărul procesoarelor nu intervine decât doar asupra performanţelor sistemului şi nici într-un fel asupra structurii sale logice,
- primitivele de sincronizare, realizate de nucleu, ascund mecanismele fizice de comutare a contextului, de exemplu, cele oferite de întreruperi.

Structura unui nucleu de sincronizare depinde, printre altele, de specificaţiile maşinii fizice (gestiunea întreruperilor, structura cuvântului de stare, sistem mono- sau multiprocesoral, etc.) şi de specificaţiile maşinii abstracte care trebuie realizate, îndeosebi de mecanismul de sincronizare ales. Este, totuşi, posibil de evidenţiat câteva caracteristici comune ale acestei structuri, pe care le vom prezenta înainte de a descrie o realizare concretă.

4.2.1. Stările unui proces. Fire de aşteptare

Am considerat până acuma că un proces se poate afla în două stări: activ sau blocat. Alocarrea fizică a unui procesor ne impune să descompunem starea activă în două stări noi. Un proces activ se numeşte **ales**, dacă el este în curs de execuţie pe un procesor fizic; el se numeşte **eligibil** dacă nu poate fi executat din cauza lipsei unui procesor disponibil. Această evidenţiere este legată doar de disponibilitatea procesorului şi nu are vre-un suport logic. Figura 4.1 descrie stările unui proces şi tranziţiile lui.

Tranziţiile 3 şi 4 (blocare şi deblocare) sunt tranziţiile "interne", datorate sincronizării proceselor. Tranziţiile "tehnologice" 1 şi 2 sunt datorate alocării procesoarelor fizice proceselor. În particular, tranziţia 2 se produce atunci când algoritmul de alocare retrage procesorul de la un proces, care încă mai are nevoie de el. Această operaţie este numită **retragere** (eng. preemption, fr. réquisition) a procesorului.

Administrarea proceselor face apel la fire de aşteptare. Astfel, fiecărei cauze distincte de blocare (semafor, condiţie într-un monitor, etc.) i se asociază un fir de aşteptare pentru a stabili o ordine a proceselor blocate. Mai mult, procesele eligibile sunt menţinute într-un fir special de aşteptare, gestionarea căruia permite implementarea unei politici de alocare a procesoarelor fizice. Dacă presupunem, că viitorul proces ales este totdeauna primul din firul proceselor eligibile, algoritmul de alocare poate fi definit

- cu ajutorul algoritmului de inserare în firul proceselor eligibile,
- cu ajutorul algoritmului care determină retragerea procesoarelor fizice.

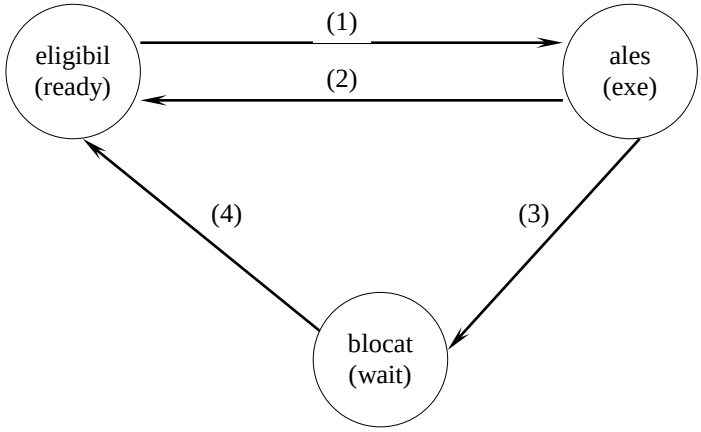


Fig.4.1. Stările unui proces

Mulţimea programelor, care realizează aceşti algoritmi se numeşte **planificator** (eng. scheduler, fr. ordonnanceur). Programul, care realizează alegerea propriu-zisă se numeşte **dispecer** (eng. dispatcher, fr. distributeur). Schema generală a firelor de aşteptare ale proceselor este prezentată în fig.4.2. Deplasarea proceselor între aceste fire corespunde schimbării stării.

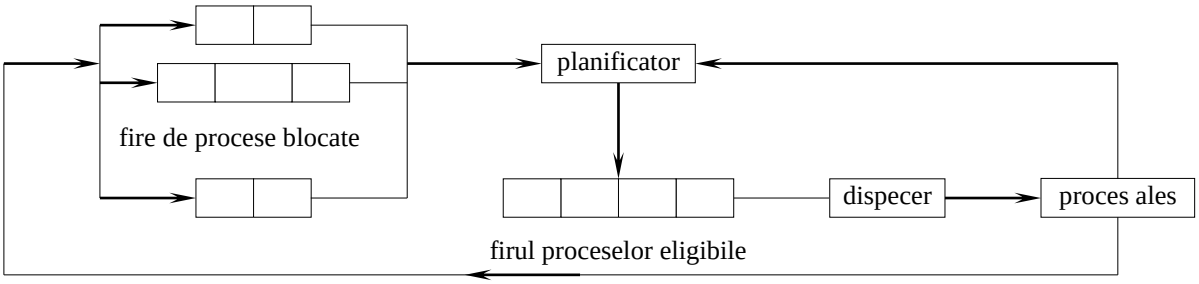


Fig.4.2. Fire de aşteptare ale proceselor, administrate de nucleul sistemului de operare

4.2.2. Administrarea contextelor şi schemele primitivelor

1) Conţinutul contextului

Operaţia de alocare a procesorului fizic impune păstrarea pentru fiecare proces a unei copii a contextului. Această copie a contextului descrie starea procesorului pentru procesul considerat. În acest scop fiecărui proces i se asociază o mulţime de informaţii rezidente în memorie şi numită **vector de stare, bloc de control al procesului** sau **blocul contextului**, care conţine:

- informaţiile despre starea procesorului, necesare la realocarea lui (conţinutul cuvântului de stare, registrelor),
- valorile atributelor procesului (prioritate, drept de acces),
- pointeri la spaţiul de lucru al procesului (segmentele procedură şi date, stiva de execuţie),
- informaţii de gestiune (starea, legăturile de înlănţuire).

O descriere mai detaliată a blocului contextului unui proces este dată în 4.3.

2) Organizarea nucleului

Execuţia programelor nucleului este declanşată în două moduri (fig.4.3):

- prin apelarea unei primitive de administrare a proceselor (creare, distrugere, sincronizare, etc.); aceste primitive sunt realizate sub formă de apel al supervizorului,
- printr-o întrerupere: programele de tratare a întreruperilor fac parte din nucleu, deoarece întreruperile sunt traduse în operaţii de sincronizare şi sunt invizibile la nivelurile superioare.

În ambele cazuri, mecanismul de intrare în nucleu conţine salvarea automată a cuvântului de stare şi, eventual, a registrelor procesorului, care execută apelul supervizorului sau care tratează întreruperea. În dependenţă de organizarea

calculatorului, aceste informaţii sunt salvate într-un amplasament fix (legat de nivelul întreruperii sau de apelul supervizorului) sau într-o stivă (îndeosebi în cazul sistemelor multiprocesorale). Părţile contextului, salvate în acest fel, sunt cele ale proceselor alese pentru procesorul în cauză, în momentul întreruperii sau apelării supervizorului.

Programele primitivelor şi cele de tratare a întreruperilor manipulează blocurile contextului şi firele proceselor. Pentru asigurarea coerenţei informaţiilor aceste programe trebuie executate în excludere mutuală. Executarea unui program al nucleului se termină în toate cazurile prin realocarea procesorului sau procesoarelor, adică prin apelarea dispecerului. Deoarece este posibil ca firul de aşteptare a proceselor alese să fi fost modificat de execuţia primitivelor, noul proces ales poate să difere de ultimul proces ales pentru procesorul întrerupt.

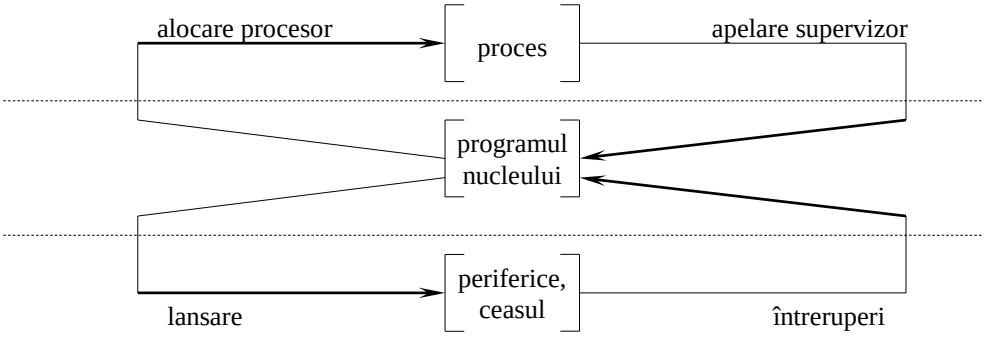


Fig.4.3. Comunicarea cu un nucleu de sincronizare

Programul unei primitive a nucleului are următoarea schemă generală:

```
prolog;                 -- salvarea contextului şi intrarea în secţiunea critică
control;                -- verificarea drepturilor şi a parametrilor
    <corpul programului> -- manipulează firele proceselor
alocare_procesor;       -- programul dispecer şi ieşirea din secţiunea critică
```

Secvenţa *prolog*, comună tuturor operaţiilor, salvează contextul procesului, care execută operaţia, şi asigură intrarea în secţiunea critică. Secvenţa *control* verifică drepturile procesului apelant de a executa primitiva şi validitatea parametrilor transmişi. Detaliile depind de primitivă. Secvenţa *alocare_procesor* este programul dispecerului: ea realizează realocarea procesorului şi ieşirea din secţiunea critică.

Diferenţa între sistemele mono- şi multiprocesorale se manifestă în mod esenţial la realizarea excluderii mutuale şi alocării procesorului (secvenţele *prolog* şi *alocare_procesor*).

Tratarea întreruperilor de către nucleu trebuie să fie coordonată cu mecanismele de sincronizare alese. De obicei sunt considerate două scheme de bază:

1) Asocierea unui proces, care tratează fiecare întrerupere. Doar acest proces se va afla în aşteptarea unei întreruperi anume, existând pentru aceasta o instrucţiune specială.
2) Asocierea unei operaţii de deblocare (*semnalizare* asociată unei condiţii, *V* la un semafor, etc.) la o întrerupere.

Problemele de mascare şi de prioritate a întreruperilor sunt, de obicei, tratate asociind priorităţi proceselor.

4.3. Realizarea unui nucleu de sincronizare

Vom prezenta în cele ce urmează schema realizării unui nucleu particular de sincronizare. Descrierea este cu titlu de ilustrare, fiind posibile multe ameliorări atât din punct de vedere al interfeţei, cât şi în ceea ce priveşte detaliile de realizare (v. exerciţiile).

4.3.1. Organizarea generală

4.3.1.1. Interfeţele

a) Gestionarea proceselor

Procesele pot fi create şi distruse în mod dinamic, şi sunt organizate ierarhic conform relaţiei de *legătură* (v.3.5). Un proces este creat cu ajutorul primitivei:

creare(p, context iniţial, atribute)

Atributele unui proces conţin prioritatea (exprimată printr-un întreg) şi drepturile de a executa anumite operaţii. Contextul iniţial specifică starea iniţială a cuvântului de stare şi a registrelor procesorului, şi a spaţiului de lucru asociat procesului (stiva, date proprii). Procesul este creat în starea *eligibil*. Numărul său *p* (numele) este returnat ca rezultat (valoarea *nil*, dacă crearea este imposibilă). O funcţie *determină_număr_propriu* permite unui proces să afle numărul său propriu.

Primitivele care urmează pot fi aplicate unui proces existent şi pot fi executate doar de procesul-părinte.

Procedura *distrugere* poate fi utilizată de către un proces pentru a se autodistruge. Deci, *distrugere(p)* va distruge toate procesele desemnate de procesul *p* şi toţi descendenţii acestora.

Procedura *suspendare(p)* întrerupe execuţia unui proces *p*, plasându-l într-un fir de aşteptare special. Execuţia lui *p* poate fi reluată doar cu ajutorul primitivei *reluare(p).* Primitivele *suspendare* şi *reluare* sunt introduse din considerente de securitate, în special pentru a permite monitorizarea unui proces de către procesul-părinte.

Utilizarea primitivelor *creare, distrugere, suspendare* şi *reluare* este condiţionată de un drept, care figurează în atributul *drepturi* al procesului.

b) Sincronizarea

Procesele sunt sincronizate cu ajutorul monitoarelor. Gestiunea întreruperilor este integrată în mecanismul monitoarelor: o întrerupere este asociată unei condiţii. Specificarea primitivelor, care diferă un pic de cea din 3.3, este precizată în 4.3.2. Monitoarele sunt declarate în programele proceselor; un monitor este creat la compilarea programului, unde el este declarat, şi este mai apoi utilizat conform regulilor, definite de limbajul de programare utilizat (care nu este aici specificat).

4.3.1.2. Structuri şi algoritmi

Din momentul creării sale unui proces i se asociază un număr fix (process handler), care serveşte la desemnarea lui şi permite accesarea blocului său de context. Blocul de context conţine următoarele câmpuri:

Csp	: zona de salvare a cuvântului de stare a procesorului,
Reg	: zona de salvare a registrelor generale ale procesorului,
Stare	: valoarea stării procesului (eligibil, blocat, ...),
Prio	: prioritatea procesului,
Drepturi	: drepturile procesului,
Fire, etc.	: legături de înlănţuire în ierarhia proceselor,
Suc, etc.	: legături de înlănţuire în firele de aşteptare (FA).

Administrarea proceselor utilizează fire de aşteptare, ordonate în ordinea de descreştere a priorităţilor şi comandate de un set de proceduri de accesare, specificate mai jos (*p* specifică un proces, *f* – un fir de aşteptare):

introducere(p, f) Introduce *p* în *f*, în elementul lui *f* care corespunde priorităţii procesului şi în ordinea sosirii pentru procesele cu aceeaşi prioritate.

primul(f) Întoarce numărul (numele) procesului din vârful lui *f* (*nil* dacă *f* este vid); nu modifică *f*.

ieşire(p, f) Extrage din firul *f* primul proces, numărul acestuia fiind pus în *p* (*nil* dacă *f* este vid).

extragere(p, f) Extrage din firul *f* procesul cu numărul *p* specificat, oricare ar fi elementul în care acesta se află; pune în *p* valoare *nil*, dacă procesul nu există în firul *f*.

vid(f) Funcţie booleană cu valoarea *adevărat*, dacă firul *f* este vid, *fals* în caz contrar.

Figura 4.4 prezintă schematic organizarea unui fir de aşteptare a proceselor.

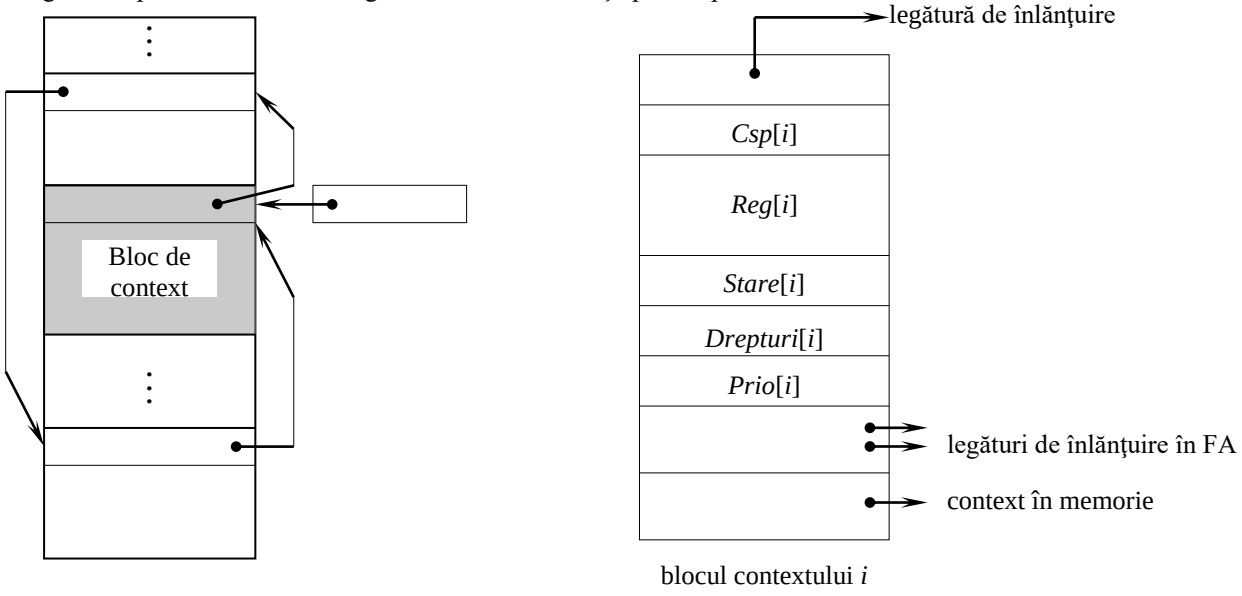


Fig.4.4. Organizarea unui FA de procese

4.3.2. Realizarea monitoarelor

Monitorul, descris mai jos, este inspirat de realizarea folosită în limbajul Mesa [12], care a fost validată de o vastă experienţă de utilizare. În raport cu noţiunea clasică de monitor (3.3.3), această realizare prezintă următoarele diferenţe:

1) *Semantica primitivei* ***semnalizare***. Specificarea iniţială a primitivei *c.semnalizare* precizează că unul din procesele care sunt în aşteptarea condiţiei *c* (dacă există) este imediat deblocat, ceea ce implică trecerea temporară în starea blocat a procesului care execută *semnalizare.* În specificarea prezentă procesul deblocat este simplu trecut în starea *eligibil* şi trebuie să intre în monitor; el se află, deci, în competiţie cu alte procese, care aşteaptă să intre în monitor, şi nu este garantat că va fi imediat ales. Condiţia, care provocase deblocarea, poate fi modificată înainte ca procesul deblocat să-şi reia execuţia în monitor. Pentru această nouă interpretare trebuie să fie modificată forma punerii în aşteptare. O construcţie de forma

if *continuare_non_posibilă* **then**
  c.aşteptare
endif

devine în acest caz

while *continuare_non_posibilă* **do**
  c.aşteptare
endwhile

Deşi această construcţie introduce un risc de privaţiune, ea prezintă o serie de avantaje de ordin practic. Ea evită o comutare a contextului, cu preţul unei evaluări suplimentare a condiţiei. Dar principalul este că ea permite definirea simplă a unor posibilităţi suplimentare utile (deblocare multiplă, întârzieri de control sau de gardă), precizate mai jos. În fine, trebuie să notăm, că verificarea validităţii monitorului este simplificată, deoarece condiţia de depăşire (*continuare_posibilă)* este consultată în timpul deblocării: procesul care execută *semnalizare* poate să se mulţumească cu garantarea unei condiţii mai slabe decât condiţia de depăşire.

2) *Deblocare multiplă.* Problema deblocării multiple poate fi rezolvată uşor introducând o primitivă nouă *c.difuzare_semnal* efectul căreia se exprimă astfel:

while *c.non_vid* **do**
  c.semnalizare
endwhile

Fiind dat, că toate procesele deblocate vor testa din nou condiţia şi cer din nou acces la monitor, această primitivă va avea evident efectul aşteptat.

3) *Întârziere de gardă.* Din probleme de securitate, în special pentru tratarea blocajelor, poate fi util să se asocieze o întârziere de gardă condiţiei unui monitor. Această întârziere este egală cu durata maximă de blocare a unui proces într-un fir de aşteptare asociat condiţiei date. La expirarea întârzierii de gardă va fi efectuată o tratare specificată. Această tratare poate consta în simpla deblocare a procesului (care va testa din nou condiţia de depăşire) sau transferul său într-un fir de aşteptare special (v.4.3.2.3).

Fie *M.c.întârziere* întârzierea de gardă asociată unei condiţii *c* în monitorul *M*. Se presupune disponibil un ceas h_{abs}, care pune la dispoziţie timpul absolut. Trebuie să adăugăm în programul primitivei *c.semnalizare* instrucţiunea următoare:

$$h_{deblocare}[p] := h_{abs} + M.c.întârziere$$

unde $h_{deblocare}[p]$ este un câmp nou al blocului de context al procesului *p*. Un proces, numit "*gardian*", deblocat la intervale regulate de timp parcurge mulţimea contextelor şi efectuează tratamentul specificat proceselor pentru care $h_{deblocare}[p] > h_{abs}$.

4.3.2.1. Algoritmi de bază

Programul monitorului trebuie să asigure două funcţii:
- excluderea mutuală pentru procedurile monitorului,
- blocarea şi deblocarea asociate primitivelor *aşteptare* şi *semnalizare*.

Fiecărui monitor *M* îi sunt asociate următoarele structuri de date:
- un dispozitiv de excludere mutuală *M.disp* (lacăt), care poate lua două valori *liber* şi *ocupat*, şi un fir de aşteptare *M.fir* asociat acestui dispozitiv. Iniţial *M.disp=liber, M.fir=<vid>*.
- fiecărei condiţii *c* de *M* îi este asociat un fir *M.c.fir*, un contor de gardă *M.c.întârziere* şi, pentru condiţiile asociate unei întreruperi, un indicator boolean *M.c.într_sosită.*

Firul proceselor eligibile este determinat de *f_eligibil.*

Pentru un monitor *M* vom cerceta programul a patru secvenţe *intrare, ieşire, c.aşteptare* şi *c.semnalizare* (secvenţele *intrare* şi *ieşire* sunt inserate de compilator şi încadrează execuţia procedurilor externe ale monitorului). Să definim mai întâi procedurile de gestiune a dispozitivului:

```
cerere_disp(M, p):
    if M.disp=ocupat then
        intrare(p, M.fir);
        stare[p]:=blocat
    else
        M.disp := ocupat;
        intrare(p, f_eligibil);
        stare[p]:=eligibil
    endif
```

```
eliberare_disp(M):
    if vid(M.fir) then
        M.disp:=liber
    else
        ieşire(q, M.fir);
        intrare(q, f_eligibil);
        stare[q]:=eligibil
    endif
```

Cele patru secvenţe se vor scrie utilizând următoarele proceduri:

```
intrare(M):
    prolog;
    p:=<proces apelant>;
    cerere_disp(M, p);
    alocare_procesor;
```

```
ieşire(M):
    prolog;
    p:=<proces apelant>;
    eliberare_disp(M);
    intrare(p, f_eligibil);
    alocare_procesor;
```

```
c.aşteptare:
    prolog;
    p:=<proces apelant>;
    intrare(p, M.c.fir);
    stare[p]:=blocat;
    eliberare_disp(M);
    alocare_procesor;
```

```
c.semnalizare:
    prolog;
    p:=<proces apelant>;
    if non_vid(M.c.fir) then
        ieşire(q, M.c.fir);
        cerere_disp(M, p);
        cerere_disp(M, q);
        eliberare_disp(M)
    else
        intrare(p, f_eligibil)
    endif
    alocare_procesor;
```

Să ne amintim, că secvenţa *prolog* asigură salvarea contextului şi intrarea în secţiunea critică, iar secvenţa *alocare_procesor* asigură alocarea procesorului şi părăsirea secţiunii critice (v.4.3.4).

Notăm, că în primitiva *semnalizare*, procesul apelant *p* şi procesul deblocat *q* sunt introduse (cu ajutorul primitivei *cerere_disp)* în firul de aşteptare pentru a intra în monitor. Procesul activat prin intermediul primitivei este primul proces din acest fir. Nu am încercat să reducem numărul transferurilor între fire pentru a realiza o implementare optimală.

4.3.2.2. Tratarea întreruperilor

Pentru asigurarea uniformităţii mecanismelor de sincronizare fiecărei întreruperi i se asociază:

- o condiţie într-un monitor,
- un proces ciclic care realizează tratarea întreruperilor, în stare de repaus acest proces este în aşteptarea condiţiei.

O condiţie poate fi asociată unui singur nivel de întrerupere. Sosirea unei întreruperi provoacă executarea funcţiei *semnalizare* pentru condiţia asociată. Prioritatea relativă a întreruperilor este tradusă în prioritatea proceselor, care tratează întreruperile.

Mecanismul descris mai sus nu este absolut perfect. De exemplu, excluderea procedurilor monitorului nu poate fi aplicată întreruperilor.* Se poate întâmpla ca o întrerupere să fie cerută atunci când procesul, care tratează întreruperile, este încă activ, din care cauză întreruperea va fi pierdută. Evitarea acestui fenomen se va face cu ajutorul unui indicator boolean, care memorizează sosirea unei întreruperi. Vom avea:

```
        <proces de prelucrare a întreruperii>

            ciclu
test        if nonM.c.într_sosită then
                c.aşteptare;                    -- evitarea pierderii unei întreruperi
                go to test
```

* Cu excepţia dacă se va utiliza pentru asigurarea blocării mascarea întreruperilor. Această soluţie (folosită în Modula-2) este aplicabilă doar pentru cazul monoprocesor şi nu poate fi acceptată, dacă există restricţii la întârzierea unei întreruperi.

endif;
<tratarea întreruperii>
endciclu
<sosirea unei întreruperi asociate lui *M.c*>
M.c.într_sosită := true;
c.semnalizare;

4.3.2.3. Tratarea erorilor

Principiul de tratare a erorilor constă în blocarea procesului care a provocat eroarea şi expedierea unui mesaj procesului părinte, care va putea lua măsurile necesare (corectarea erorii şi relansarea sau distrugerea procesului, care a generat eroare). Pentru aceasta este folosit un fir special *f_eroare* (în conformitate cu organizarea sistemului, poate fi prevăzut un fir unic sau un fir pentru fiecare utilizator, pentru fiecare subsistem, etc.).

Presupunem că o eroare care are loc în cursul execuţiei unui proces provoacă o deviere, tratarea căreia se va scrie astfel:

```
prolog;
p:=<proces apelant>;
intrare(p, f_eroare);
<tratare specifică>;
stare[p]:=suspendat;
alocare_procesor;
```

Am definit o stare nouă ("*suspendat*"), care se aplică unui proces activitatea căruia a fost întreruptă de un eveniment, considerat anormal (eroare de execuţie sau acţiunea primitivei *suspendare,* v.4.3.3).

Nu detaliem aici *<tratare specifică>*, care trebuie să fie specificat de către procesul părinte la momentul creării procesului descendent. Acest program conţine, evident, codul de diagnosticare (identitatea procesului generator de eroare, natura erorii), care trebuie transmis procesului părinte într-un mod special, conform gradului de urgenţă (actualizarea unui indicator, deblocare, etc.).

4.3.3. Operaţii asupra proceselor

4.3.3.1. Crearea şi distrugerea proceselor

Problema principală, condiţionată de gestiunea dinamică a proceselor, este alocarea contextelor şi numelor proceselor. Pentru aceasta sunt utilizate două metode principale:

- pentru blocurile contextelor sunt rezervate un număr fix de amplasamente; amplasamentele neutilizate sunt determinate de o valoare specială (*nil*) a câmpului *stare* al lor; fiecare bloc este desemnat printr-un număr, care este numărul utilizat pentru desemnarea procesului asociat;
- amplasamentele rezervate blocurilor de context sunt alocate dinamic în memorie; numerele sunt alocate proceselor de asemenea în mod dinamic şi un tabel de corespondenţă, asociază numărului fiecărui proces adresa în memorie a blocului său de context.

În ambele cazuri vom presupune disponibilă o procedură *alocare_context(p)*, care realizează alocarea contextului (blocul de context şi spaţiul de lucru) şi întoarce ca rezultat un număr *p* al procesului (*nil*, dacă crearea este imposibilă, de exemplu, din cauza lipsei de spaţiu suficient în memorie). Metodele de alocare a spaţiului de lucru nu sunt precizate aici. Numărul procesului creat este întors drept rezultat al primitivei:

```
creare(p, context iniţial):

    prolog;
    control;                        -- verificarea drepturilor
    alocare_context(p);
    if p ≠ nil then
        iniţializare_context(i);
        intrare(p, f_eligibil)
    endif;
    intrare(proces apelant, f_eligibil);
    alocare_procesor;               -- este întors p drept rezultat
```

Contextul iniţial este specificat de către procesul creator: el trebuie să definească valoarea iniţială a registrelor şi a cuvântului de stare a procesului creat, starea iniţială a spaţiului de lucru, atributele, cum ar fi prioritatea şi drepturile. Unele câmpuri ale cuvântului de stare sunt predefinite şi nu pot fi modificate (modul, mascarea întreruperilor, etc.). Pentru elementele legate de protecţie (drepturile de acces), procesul creat nu poate avea drepturi superioare drepturilor procesului creator; în caz contrar atributele de protecţie sunt declarate defecte.

Distrugerea unui proces trebuie să implice eliberarea resurselor, care îi fuseseră alocate. Printre aceste resurse, doar numele şi contextul sunt gestionate direct de nucleu; celelalte resurse, cum ar fi fişierele, sunt preluate de mecanisme specifice.

Distrugerea unui proces, care se află în secţiunea critică poate conduce la o blocare. Secţiunile critice ale monitoarelor sunt gestionate direct de nucleu. Este posibil să se asocieze unui proces numărul dispozitivului de blocare, care se află în posesia procesului dat (el poate fi angajat în mai multe apeluri incorporate), şi să diferenţiem distrugerea procesului până când valoarea acestui număr nu va fi *0*. O altă soluţie constă în examinarea periodică a fiecărui dispozitiv de blocare şi să eliberăm dispozitivul de blocare, dacă procesul care îl posedă a fost distrus.

Principiul primitivei *distrugere* este dat în schema de mai jos:

```
distrugere (p):

    prolog;
    control;                              -- verificarea drepturilor
    eliberare_context(p);
    intrare(proces apelant, f_eligibil);
    alocare_procesor;
```

Procedura *eliberare_context* trebuie să asigure eliberarea resurselor ocupate de procesul distrus şi de descendenţii acestuia:

```
eliberare_context(p):

    listă:=<lista firelor procesului p>;
    restituire_bloc_context(p);
    restituire_memorie(p);
    for q∈ listă do
            eliberare_context(q)
    endfor;
```

4.3.3.2. Suspendarea şi reluarea

Primitiva *suspendare* permite procesului-părinte să controleze activitatea unui proces descendent, întrerupând în mod forţat execuţia acestuia. O utilizare curentă este suspendarea unui proces, angajat într-o buclă infinită. Procesul întrerupt în acest mod este transferat într-un fir de aşteptare special, care poate fi firul *f_eroare*, utilizat pentru devieri.

Efectul primitivei *suspendare* poate fi ca şi al unei devieri şi programul de tratare poate fi analogic. Suspendarea unui proces pune o problemă analogică celei de distrugere, dacă procesul se află în secţiunea critică într-un monitor.

```
suspendare(p):

    prolog;
    control;
    < tratare secţiune critică>;
    f:=<fir care conţine p>;
    extragere(p, f);
    intrare(p, f_eroare);
    stare[p]:=suspendat;
    intrare(proces apelant, f_eligibil);
    alocare_procesor;
```

Primitiva *reluare* permite unui proces să deblocheze un fir suspendat, după modificarea eventuală a contextului său.

```
reluare(p):

    prolog;
    control;
    extragere(p, f_eroare);
    stare[p]:=eligibil;
    intrare(proces apelant, f_eligibil);
    intrare(p, f_eligibil);
    alocare_procesor;
```

4.3.4. Excluderea mutuală şi alocarea procesorului

4.3.4.1. Realizarea pentru cazul monoprocesor

În acest caz excluderea mutuală este realizată prin mascarea întreruperilor. Pentru aceasta trebuie pregătită masca întreruperii în cuvântul de stare, care ar specifica programele asociate primitivelor de tratare a întreruperilor. Dacă notăm prin *proces_ales* o variabilă globală, care conţine numărul procesului ales, iar prin *salv_csp* locaţiunea în care a fost salvat cuvântul de stare a procesorului la apelarea supervizorului sau la întrerupere, prologul va fi de forma:

```
prolog:
    <mascarea întreruperilor>        -- masca în cuvântul de stare
    csp[proces_ales] := salv_csp;
    salv_registre(Reg[proc_al]);
```

Programul dispecerului, care de asemenea realizează ieşirea din secţiunea critică, are grijă să aloce procesorul primului proces din firul de procese eligibile. Pentru simplificarea manipulării acestui fir este binevenit să fie introdus aici un proces special cu prioritate joasă, care rămâne tot timpul în coada firului şi nu poate fi blocat. Acest proces, care poate fi ales doar atunci când el este unicul eligibil, execută o activitate de fond, care nu este urgentă sau o simplă buclă de aşteptare. El garantează, deci, că firul proceselor eligibile nu este niciodată vid.

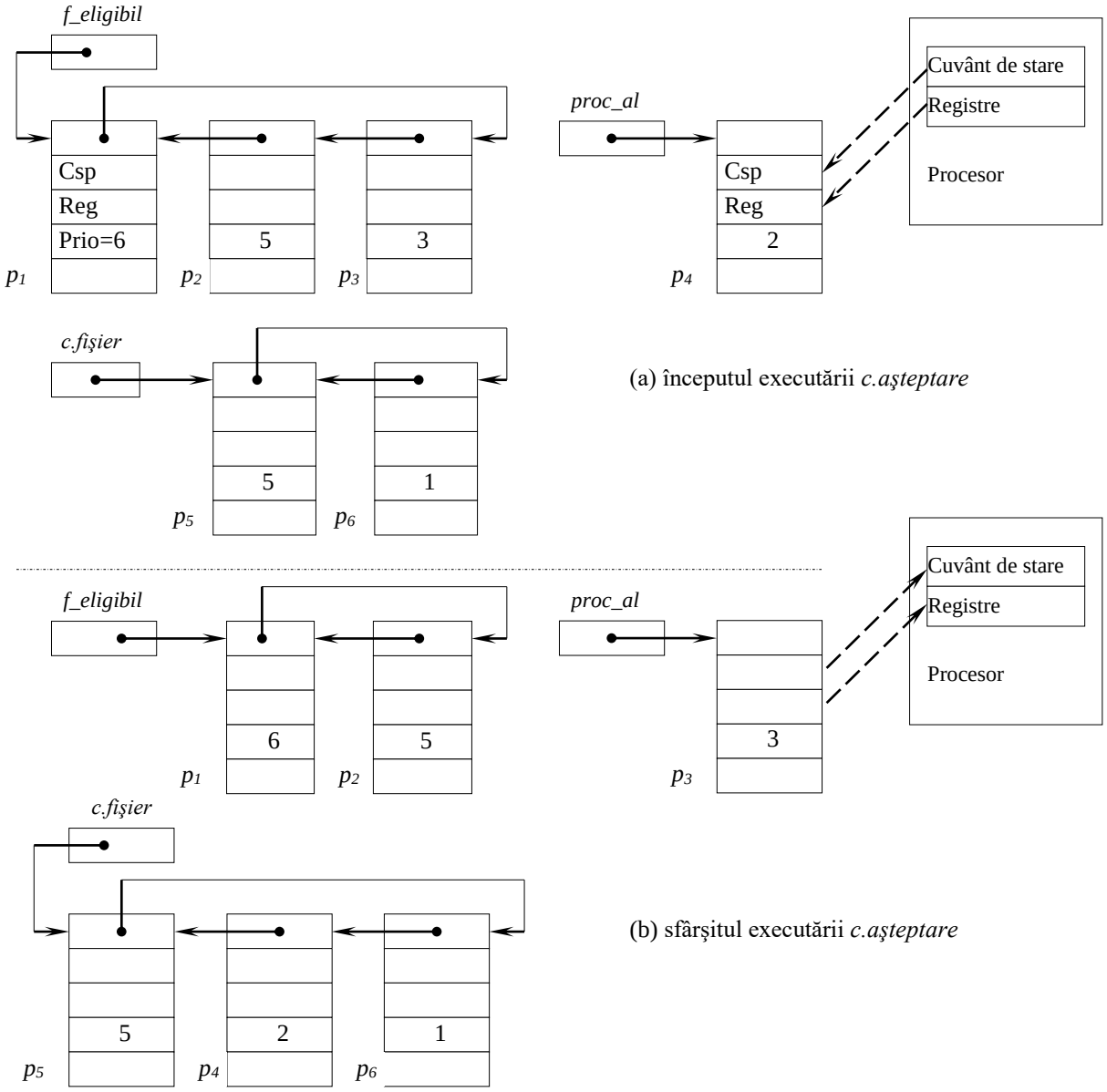


Fig.4.5. Alocarea procesorului

Programul dispecerului este de forma:

```
alocare_procesor:
    ieşire(proces_ales, f_eligibil);
```

```
încărcare_registre(Reg[proc_al]);
încărcare_csp(csp[proces_ales]);
```

Figura 4.5 ilustrează principiul de funcţionare a nucleului, exemplificând efectul global al unei realocări a procesorului după blocarea procesului ales.

4.3.4.2. Realizarea pentru cazul unui sistem multiprocesoral

Descrierea, care urmează este inspirată din [13]. Vom specifica mai întâi organizarea fizică a sistemului. Avem *n* procesoare identice, care accesează o memorie comună. Fiecare procesor, desemnat printr-un număr (de la *0* la *n-1*), îşi cunoaşte numărul propriu. Regimului de funcţionare multiprocesor îi sunt specifice două instrucţiuni:

Test and Set(R, m)	*:*	asigură excluderea mutuală (4.1.1.2)
Întrerupere(k)	*:*	provoacă o întrerupere a procesorului *k*.

O întrerupere sau un apel de supervizor provoacă introducerea cuvântului de stare şi a registrelor procesorului în cauză într-o stivă în memorie, proprie acestui procesor.

Orice primitivă a nucleului poate fi executată de oricare din procesoare. Vom împărţi întreruperile în întreruperi, destinate unui procesor specific (ceasul procesorului, instrucţia *Întrerupere(k)*) şi întreruperi banalizate (intrări-ieşiri, de exemplu), care sunt tratate de orice procesor. În acest caz pentru a fi retras se va alege procesorul, care execută procesul cu cea mai mică prioritate. Numărul acestui proces se află într-un amplasament rezervat de memorie k_{min} şi vom considera, că un dispozitiv fizic îndreaptă întreruperile banalizate spre procesorul cu numărul k_{min}. Vom utiliza acelaşi principiu de alegere şi în cazul unei alocări a procesoarelor.

Pentru ca firul proceselor eligibile să nu fie vid vom adăuga aici *n* procese de prioritate joasă. Numărul procesului ales pentru a fi executat de procesorul *k* vom nota *proc_al[k]*.

Excluderea mutuală a primitivelor nucleului utilizează un dispozitiv de blocare, consultat şi actualizat cu ajutorul instrucţiunii *Test And Set.*

```
prolog:                                  -- procesorul k
        <mascare întreruperi>;
        Csp[proc_al[k]]:=top(stivă_csp);
        Reg[proc_al[k]]:=top(stivă_reg);
test:   Test And Set(R, dispozitiv de blocare);
        if R ≠ 0 then
            go to test                   -- aşteptare activă
        endif
```

Dispecerul va asigura ca cele *n* procese alese să fie la orice moment de timp cu prioritatea cea mai înaltă printre procesele eligibile. Deoarece executarea primitivei curente poate modifica firul proceselor eligibile, prioritatea primului proces eligibil este comparată cu prioritatea procesului ales pentru procesorul k_{min}. Dacă aceasta este superioară procesorul dat este retras cu ajutorul instrucţiunii *Întrerupere*.

```
alocare_procesor:                        -- pentru procesorul cu numărul k
    ieşire(proces_ales[k], f_eligibil);
    pr1:=Prio[primul(f_eligibil)];
    pr2:=Prio[proc_al[kmin]];
    if Prio[proc_al[k]] < pr2 then -- actualizarea lui kmin
        kmin:=k
    endif;
    disp_de_blocare:=0;              -- terminarea secţiunii critice
    if pr1 > pr2 then
        Întrerupere(kmin)            -- retragere
    endif;
    încărcare_registre(Reg[proc_al[k]];
    încărcare_csp(Csp[proc_al[k]];       -- demascarea întreruperilor
```

Tratarea întreruperii de retragere (instrucţiunea *Întrerupere*) a procesorului se reduce la realocarea procesorului întrerupt:

```
<tratarea întreruperii de retragere a procesorului k>
    prolog;
    intrare(proc_al[k], f_eligibil);
    alocare_procesor;
```

4.4. Procese şi fire în Linux

4.4.1. Crearea proceselor

În Linux procesele „se înmulţesc” prin *clonare*: apelul de sistem, care crează un proces nou, se numeşte *clone*, iar procesul fiu este o copie aproape exactă a procesului părinte, doar că mai departe va executa codul său, iar procesul părinte – ceea ce este scris după apelarea lui *clone*. Diferenţele pot deveni foarte mari şi dacă dorim să evităm diferenţierea, apelarea lui *clone* permite să definim următorii indicatori (*flags*), care vor specifica momentele comune ale fiului şi părintelui:

- Spaţiul de adrese (*Clone_VM*);
- Informaţiile despre sistemul de fişiere (*Clone_FS*);
- Tabelul fişierelor deschise (*Clone_FILES*);
- Tabelul programelor de tratare a semnalelor (*Clone_SIGHAND*);
- Părintele (*Clone_PARENT*) – în acest caz, evident, va fi creat un proces – frate.

Firele sunt realizate în biblioteca standard de susţinere a programelor cu mai multe fire ca şi procesele, generate cu indicatorul *Clone_VM*, şi, din punctul de vedere al nucleului sistemului, nu se deosebesc de alte procese. Însă în unele biblioteci de alternativă pot exista diferenţe.

Mai există fire, numite *„handicapate”*, generate de funcţia *kernel_thread* pentru necesităţi interne ale sistemului. Acestea nu au parametri pentru linia de comandă, de obicei nu au fişiere deschise, etc. Deoarece aceste fire (procese) de asemenea figurează în lista lucrărilor, adesea în literatură poate fi întâlnită noţiunea de proces propriu-zis, creat din „spaţiul utilizatorului” (*userspace*), şi noţiunea de lucrare, prin aceasta înţelegându-se toate procesele, inclusiv procesele interne ale nucleului.

Procesele sunt create prin utilizarea funcţiilor din familia *exec* ale bibliotecii Linux standard: *execl, execlp, execle, execv, execve, execvp*. Deşi formatul de apelare este diferit, în ultimă instanţă execută acelaşi lucru: înlocuiesc codul din procesul curent cu codul, care se află în fişierul indicat. Fişierul poate fi un fişier binar executabil Linux, un script al interpretorului limbajului de comandă, un fişier binar de un alt format (de exemplu, o clasă *java*, un fişier executabil DOS). În ultimul caz modalitatea de prelucrare va fi determinată de modulul de adaptare a nucleului *binfmt_misc*. Din această cauză, operaţia de lansare a unui program, care în DOS şi Windows formează un tot întreg, în Linux (şi în Unix, în general) este împărţită în două: mai întâi are loc lansarea propriu-zisă, iar apoi se va determina care program va fi executat. Are oare aceasta sens şi care sunt cheltuielile suplimentare? Doar crearea copiei unui proces presupune copierea unui volum semnificativ de informaţii!

În mod sigur putem afirma că are sens această abordare. Foarte frecvent un program trebuie să îndeplinească unele acţiuni înainte de începerea propriu-zisă a execuţiei lui. De exemplu, lansăm două programe, care-şi transmit reciproc date prin intermediul unui canal fără nume. Astfel de canale sunt create prin apelarea de sistem *pipe*, care returnează o pereche de descriptori de fişiere de care pot fi legate, de exemplu, fluxul standard de intrare (*stdin*) al unui program şi de ieşire (*stdout*) al celuilalt program. Aceasta este posibil deoarece mai întâi sunt create procesele, apoi vor fi executate manipulările necesare cu descriptorii fişierelor şi doar după toate acestea este apelată funcţia *exec*.

Aceleaşi rezultate pot fi obţinute în Windows NT într-un singur pas, dar într-un mod mult mai complicat. Cât despre cheltuielile suplimentare ele sunt în majoritatea cazurilor minime, deoarece dacă este creată copia unui proces datele proprii ale acestuia nu sunt fizic copiate. Şi aceasta din cauza că este utilizat procedeul *copy-on-write*: paginile datelor ambelor procese sunt marcate într-un anumit mod şi doar atunci când un proces va încerca să modifice conţinutul uneia din paginile sale aceasta va fi duplicată.

Primul proces este lansat în sistem la iniţializarea nucleului. Fragmentul de cod de mai jos este partea de încheiere a procedurii de iniţializare a nucleului sistemului de operare Linux:

```
if (execute_command)
execve(execute_command,argv_init, envp_init);
execve("/sbin/init",argv_init,envp_init);
execve("/etc/init",argv_init,envp_init);
execve("/bin/init",argv_init,envp_init);
execve("/bin/sh",argv_init,envp_init);
panic("No init found.  Try passing init= option to kernel.");}
```

Este făcută încercarea de comutare a procesului la fişierul, indicat în linia de comandă a nucleului, apoi la fişierele */sbin/init, /etc/init, /bin/init* şi, în sfârşit, la fişierele din */bin/sh*.

4.4.2. Distrugerea proceselor

La terminarea execuţieia unui proces (normal, forţat sau accidental), el este distrus eliberând toate resursele, care fusese alocate anterior. Dacă procesul părinte se termină înaintea procesului descendent, ultimul devine “*orfan*”

(*orphaned process*). Toţi "*orfanii*" sunt "*înfiaţi*" în mod automat de programul *init*, executat de procesul cu numărul 1, care duce evidenţa terminării execuţiei lor.

Dacă a fost terminată deja execuţia procesului descendent, iar procesul părinte nu este gata să recepţioneze de la sistem semnalul despre acest eveniment, descendentul nu dispare total, ci este transformat în *Zombie*; în câmpul *Stat* aceste procese sunt notate cu litera *Z*. Procesele *Zombi* nu cer timp de procesor, dar în tabelul proceselor este păstrată linia lor şi structurile respective ale nucleului nu sunt eliberate. După terminarea execuţiei procesului părinte, procesul *Zombi orfan* devine pentru o perioadă scurtă de timp descendentul lui *init*, ca mai apoi să "*moară*" definitiv.

Un process poate să "*cadă în hibernare*", fără a putea fi scos din această stare: în câmpul *Stat* acest eveniment se va nota prin litera *D*. Procesele aflate în hibernare nu reacţionează la cererile de sistem şi pot fi distruse doar prin reîncărcarea sistemului.

4.4.3. "Demoni" în Linux

Demon (*daemon*) în Linux este numit procesul predestinat să lucreze în regim de fond fără terminal şi care execută anumite operaţii pentru alte procese (nu obligator pe calculatorul Dumneavoastră). De obicei, demonii îşi îndeplinesc în linişte lucrul şi ne amintim de ei doar în cazul unor situaţii ieşite din comun: spaţiu insuficient – demonul singur informând utilizatorul despre aceasta, sau refuz să lucreze şi sunteţi întrebat de şef când se vor termina problemele cu imprimantă ☺.

Pentru multe calculatoare demonii, care servesc procesele altor calculatoare, sunt rar utilizaţi din care cauză nu trebuiesc păstraţi constant în memorie cu cheltuieli neraţionale ale resurselor sistemului. Pentru coordonarea lucrului acestora a fost creat un superdemon – *inetd* (*Internet daemon*).

În fişierul de configurare *inetd* (*/etc/inetd.conf*) este indicat care demon accesează un serviciu anume de Internet. De obicei, cu ajutorul lui *inetd* sunt apelate programele *pop3d, imap4d, ftpd, telnetd* (exerciţiu - determinaţi serviciul pus la dispoziţie), etc. Aceste programe nu sunt în mod constant active, în rezultat, ele nu pot fi considerate demoni în adevăratul sens al cuvântului, dar, deoarece ele sunt create de un demon adevărat, sunt numite demoni.

4.4.4. Obţinerea informaţiilor despre procese

Pentru obţinerea informaţiilor despre procese, vizualizate de programele *ps* şi *top*, Linux-ul utilizează un sistem special de fişiere, numit *procfs*. În majoritatea distributivelor el este iniţializat la lansarea sistemului de operare cu titlul de catalog */proc*. Datele despre procesul cu numărul 1 (de obicei */sbin/init*) se află în subcatalogul */proc/1*, despre procesul cu numărul 182 - în */proc/182*, etc. Toate fişierele, deschise de un proces, sunt reprezentate sub forma unor referinţe simbolice în catalogul /proc/<pid>/fd, iar referinţa la catalogul rădăcină este păstrată ca /proc/<pid>/root.

Sistemului de gestiune a fişierelor *procfs* îi sunt asociate şi alte funcţii. De exemplu, cu ajutorul comenzii *echo 100000>/proc/sys/fs/file-max* un *superuser* poate indica, că se permite deschiderea unui număr de până la 100000 de fişiere, iar comanda *echo 0>/proc/sys/kernel/cap-bound* va retrage proceselor din sistem toate drepturile suplimentare, adică va priva sistemul de noţiunea *superuser*.

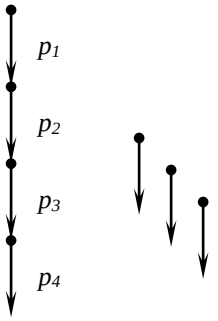
Informaţii utile pune la dispoziţie programul *lsof*. Acesta returnează lista tuturor fişierelor, utilizate la momentul curent de către procese, inclusiv cataloagele folosite de către unele procese în calitate de catalog curent sau catalog rădăcină, bibliotecile dinamice, încărcate în memorie, etc.

4.4. Exerciţii la capitolul 4

Exerciţiul 1 [14]. O mulţime de procese, care cooperează pentru executarea unei lucrări comune, poate fi reprezentată printr-un graf orientat în care fiecare arc reprezintă evoluţia completă a unui proces. Graful obţinut în aşa mod este numit **graful proceselor** lucrării considerate. Introducem două reguli de compunere *S(a, b)* şi *P(a, b)*, în care *a* şi *b* desemnează procese, astfel încât:

S(a, b) reprezintă execuţia secvenţială a proceselor *a* şi *b*,
P(a, b) reprezintă execuţia lor paralelă.

Folosind doar funcţiile *S* şi *P*, descrieţi atunci când este posibil, următoarele grafuri:



(1)

Exerciţiul 2. În cazul când nu se cere respectarea strictă a ordinii executării unor subexpresii în cadrul unei expresii aritmetice, valorile subexpresiilor pot fi calculate într-o ordine relativ arbitrară, fiind posibilă evaluarea paralelă, dacă există un număr suficient de procesoare.
Fie expresia: *(a+b)*(c+d) – (e/f)*

1) Elaboraţi structura arborelui corespunzător.
2) Folosind convenţiile exerciţiului 1, elaboraţi un graf al proceselor corespunzătoare unei evaluări paralele a acestei expresii. Se va încerca să se exploateze la maximum paralelismul.
3) Aduceţi o descriere a acestui graf, utilizând în exclusivitate funcţiile *S* şi *P*, introduse în exerciţiul 1.

Exerciţiul 3. *Problema excluderii mutuale pentru procesele paralele cu ajutorul variabilelor comune* [15]. Acest exerciţiu are drept scop demonstrarea dificultăţii programării excluderii mutuale pentru cazul proceselor paralele, dacă unicele operaţii de care dispunem sunt modificarea şi testarea valorii unei variabile (operaţii de tipul *TAS,* ca şi semafoarele sunt excluse). Principiul soluţiei căutate constă în definirea unei mulţimi de variabile de stare, comune contextelor diferitor procese; autorizarea intrării în secţiunea critică se va face prin teste asupra acestor variabile, iar aşteptarea eventuală va fi programată ca o aşteptare activă, adică prin repetarea ciclică a testelor.

1) Ne vom limita pentru început la cazul a două procese, notate p_1 şi p_2 şi vom încerca să construim o soluţie, folosind o variabilă booleană unică *c,* cu valoarea **true**, dacă unul din procese se află în secţiunea sa critică şi **false** în caz contrar.

   Arătaţi, că excluderea mutuală nu poate fi programată utilizând numai această unică variabilă.

2) Construiţi o soluţie, utilizând o variabilă comună unică *t*, cu următoarea condiţie: *t = i (i=1, 2)*, atunci şi numai atunci când p_i este autorizat să intre în secţiunea sa critică.
   - Scrieţi programul procesului p_i,
   - Verificaţi, care din caracteristicile excluderii mutuale sunt prezente.

3) Pentru a fi îndeplinite toate caracteristicile excluderii mutuale, introducem câte o variabilă booleană pentru proces; fie variabila *c[i]* ataşată procesului p_i cu următorul sens:
   - ♦ *c[i]=true* dacă procesul p_i este în secţiunea sa critică sau cere să intre acolo,
   - ♦ *c[i]=false* dacă procesul p_i este în afara secţiunii sale critice.

Procesul p_i poate să citească şi să modifice *c[i]* şi numai să citească *c[j] (j≠i)*.
   - Scrieţi programul procesului p_i,
   - Verificaţi, care din caracteristicile excluderii mutuale sunt prezente.

4) Combinând 2) cu 3) putem obţine o soluţie corectă, adică, completând soluţia 3) prin introducerea unei variabile suplimentare *t,* care va servi la reglarea conflictelor la intrarea în secţiunea critică. Această variabilă poate fi modificată doar la ieşirea din secţiunea critică. Principiul rezolvării este următorul: în caz de conflict (adică, dacă *c[1]=true* şi *c[2]=true*) cele două procese trec într-o secvenţă de aşteptare în care *t* păstrează o valoare constantă. Procesul p_i cu $i \neq t$ anulează cererea sa (*c[i]=false*), permiţând celuilalt proces să intre în secţiunea sa critică; p_i aşteaptă într-o buclă ca *t* să fie pus în *i* înainte de repetarea cererii sale de intrare prin *c[1]=true*.
   - Scrieţi programul procesului p_i,
   - Verificaţi, care din caracteristicile excluderii mutuale sunt prezente.

5) Generalizaţi soluţia 4) pentru *i>2*.

Exerciţiul 4. *Descrierea unui sistem simplu de întrerupere.*

Considerăm un sistem de întrerupere, utilizând două bistabile: o mască *m* şi un indicator de întrerupere *t.* Întreruperea este mascată, dacă *m=0* şi autorizată, dacă *m=1*. Sosirea unui semnal de întrerupere se manifestă prin tentativa de a pune pe *t* în *1: t := 1*. Dacă întreruperea este demascată, *t* trece imediat în *1;* în caz contrar, *t* trece în *1* în momentul demascării. Punerea lui *t* în *1* conduce la deblocarea unui proces ciclic de tratare a întreruperii.
Descrieţi, cu ajutorul semafoarelor, logica acestui dispozitiv cablat şi a procesului ciclic de tratare.

5. Gestiunea informaţiei

Acest capitol este consacrat studierii principiilor de gestiune a informaţiei într-un sistem de operare. Noţiunea de obiect formează suportul director al studiului dat. Vom începe cu prezentarea unor concepte de bază, care permit o mai bună înţelegere a mecanismelor de administrare a informaţiei într-un sistem: nume sau identificator, cale de acces, legare, protecţia obiectelor ş.a.. Aceste noţiuni vor fi utilizate în două domenii importante: desemnarea şi legarea fişierelor (5.2) şi legarea programelor şi a datelor (5.3). Vom mai prezenta în 5.4 mecanismele fizice şi logice de bază, care facilitează administrarea obiectelor (segmente, capacităţi) şi exemple de implementare a acestora în cadrul sistemelor de operare.

Se consideră că informaţia, care circulă într-un sistem de calcul constă din obiecte; obiectele sunt entităţile asupra cărora sunt efectuate anumite operaţii. Toate operaţiile pot fi clasificate în patru categorii:

- de creare,
- de modificare,
- de căutare,
- de distrugere a obiectelor.

Fiecare obiect are o reprezentare **externă** (în afara calculatorului) şi una **internă**, determinată de suportul fizic. Un obiect poate fi accesat cu ajutorul **funcţiilor de acces**.

Problema centrală a administrării informaţiei constă în conversia reprezentării externe şi a funcţiilor de acces asociate în reprezentarea internă şi funcţiile de acces corespunzătoare fiecărui obiect al sistemului.

5.1. Principiile gestiunii informaţiei

Noţiunile de identificator, valoare, tip, desemnare, reprezentare sunt utilizate în informatică în forme extrem de diverse. Elaborarea unui model general pentru aceste noţiuni mai este încă o problemă deschisă. Ne vom limita aici la prezentarea unui model simplificat, care se conturează în cadrul limbajelor de programare şi care permite să ţinem cont de cele mai frecvente situaţii.

5.1.1. Definiţii generale

Programul unui sistem informatic descrie acest sistem ca o mulţime de obiecte. Obiectele sistemului şi operaţiile asociate sunt cele specificate de limbajul utilizat. Pentru implementarea sistemului descris de un sistem informatic va trebui să definim pentru fiecare obiect o reprezentare concretă, care are forma unei mulţimi de informaţii în memorie, pentru unităţile periferice, etc. Implementarea sistemului se traduce prin acţiuni, care modifică starea acestor reprezentări.

Procesul de reprezentare utilizează două scheme de bază. În schema **compilării** obiectele abstracte, specificate de program, sunt înlocuite, în faza preliminară de translatare, prin reprezentările lor. Aceste reprezentări sunt obiecte executabile direct interpretate de un dispozitiv fizic. În schema **interpretării** un sistem logic (interpretorul) este alimentat direct din program; el asociază (în mod static sau dinamic) o reprezentare internă fiecărui obiect şi stabileşte corespondenţa între obiectul abstract şi reprezentarea la fiecare accesare. Schema interpretării este de regulă mai puţin eficace în comparaţie cu schema compilării, deoarece corespondenţa obiectelor trebuie realizată la fiecare accesare, însă ea convine mai mult în cazul unei gestionări dinamice a obiectelor (înlocuirea unui obiect prin altul, modificarea reprezentării unui obiect). În practică, este bine de combinat aceste două scheme în funcţie de restricţiile concrete.

Programul unui sistem utilizează **nume** pentru a desemna obiectele acestui sistem. Numele unui obiect este o informaţie cu funcţie dublă: pe de o parte permite să se facă distincţia obiectului dat de alte obiecte; pe de altă parte, el serveşte ca şi cale de acces la obiect, adică el poate fi interpretat în scopul efectuării unor acţiuni asupra obiectului. Numele respectă anumite reguli proprii limbajului de programare. Nume sunt identificatorii, care desemnează variabilele şi procedurile într-un limbaj de programare sau fişierele într-un limbaj de comandă.

În cazul reprezentării externe un identificator desemnează un anumit obiect, care poate fi o constantă sau o informaţie ce permite accesul la un alt obiect (obiectul *permite referirea* unui alt obiect, fig. 5.1).

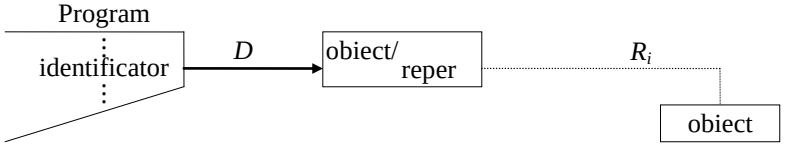


Fig. 5.1. Relaţia identificator - obiect

Trecerea de la identificator la obiectul propriu-zis se realizează prin compunerea funcţiilor de acces D şi R_i, obţinând **calea de acces** la un obiect.

Suportul fizic al informaţiei este memoria. La acest nivel memoria poate fi considerată un şir de amplasamente caracterizate printr-un conţinut. Un proces desemnează un amplasament printr-un nume folosit de unitatea centrală pentru a citi şi la necesitate a modifica conţinutul acestui amplasament. Conţinutul unui amplasament poate fi interpretat ca un întreg, o instrucţiune sau ca un nume care desemnează alt amplasament.

Reprezentarea externă este convertită în cuplul (*amplasament, conţinut*), numele amplasamentului devenind numele obiectului. De exemplu, o constantă devine un cuplu (*amplasament, conţinut invariabil*), iar un obiect care repereză altul - (*amplasament, conţinut variabil*): amplasamentul conţine reprezentarea unui obiect care este fie reprezentarea internă a unei constante, fie un nume.*

Vom face o deosebire între celulele memoriei fizice şi amplasamente prin utilizarea noţiunii de **adresă** pentru primele şi **nume** pentru amplasamente. Obiectele definite de perechea (*amplasament, conţinut*) pot fi deplasate în memoria fizică, schimbarea adresei fizice a unui amplasament nu afectează, în general, numele lui.

Acestea au fost **obiecte simple**. Obiectele constituite din mai multe obiecte de acelaşi tip sau de tip diferit (masive, fişiere, structuri) se numesc **obiecte compuse**. Numele obiectului compus se află într-un amplasament al cărui conţinut este **descriptorul** obiectului. Descriptorul, fiind reprezentarea internă a unui obiect compus, defineşte tipul, numărul componentelor obiectului, o eventuală ordine a acestora, precum şi amplasamentele în care se află aceste componente. Funcţia de acces asociată descriptorului are parametri şi furnizează un conţinut sau un nume de amplasament.

Numim **obiect accesibil** un obiect căruia i s-a asociat o cale de acces. Numim **obiect partajat** orice obiect care este accesibil mai multor utilizatori (eventual cu drepturi de acces diferite). Folosirea numelor care iniţiază calea de acces la un obiect partajat poate fi imaginată în mai multe feluri: fie utilizarea aceluiaşi nume sau a unor nume diferite pentru a avea acces la acelaşi obiect, fie folosirea aceluiaşi nume pentru a desemna reprezentări diferite.

Partajarea obiectelor la nivelul aceluiaşi nume (fig. 5.2, a) presupune rezervarea numelor tuturor obiectelor partajabile în mod potenţial. Rezervarea acestor nume rămâne efectivă chiar dacă obiectele în cauză nu mai sunt folosite, ansamblul obiectelor accesibile unui proces împărţindu-se în două părţi disjuncte: **obiectele private** ale procesului şi cele **potenţial partajabile**.

În cazul partajării unui obiect folosind nume diferite (fig. 5.2, b), fiecare proces posedă un nume care după stabilirea căii de acces îi permite accesul la o reprezentare unică.

În unele situaţii acelaşi nume permite accesul la reprezentări diferite; este cazul procedurilor reenterabile în care un nume fixat la compilare conduce în timpul execuţiei la reprezentări diferite, proprii proceselor care utilizează această procedură (fig. 5.2, c).

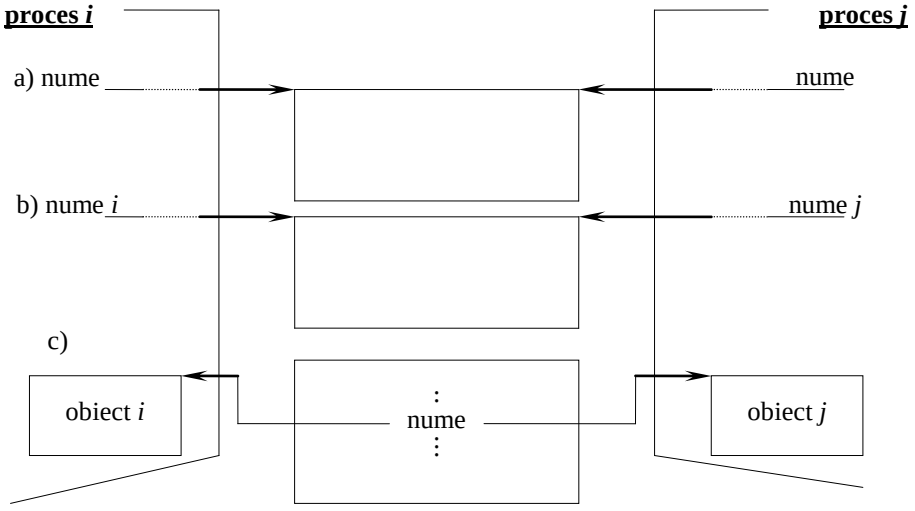


Fig. 5.2. Accesarea obiectelor partajate

Numim **durată de existenţă** sau **de viaţă** a unui obiect perioada de timp în care acesta este accesibil. Sistemul de operare şi tipul obiectului determină durata de existenţă a obiectului. Un masiv creat de un program într-un limbaj de programare este distrus la terminarea execuţiei programului, amplasamentele pe care le ocupă putând fi reutilizate pentru reprezentarea altor obiecte, în timp ce un fişier poate supravieţui programul care l-a creat.

5.1.2. Interpretarea numelor

La nivelul sistemului de operare obiectele sunt memorate în amplasamente, iar procesele le pot accesa prin nume. Stabilirea căii de acces la un obiect prin compunerea funcţiilor de acces asociate se numeşte **legare**. Se spune că un obiect este **legat** atunci când pentru el este stabilită calea de acces. În cazul operaţiilor aritmetice calea de acces asociată

* În calculatoarele care permit adresarea imediată, o constantă poate fi reprezentată în instrucţiune.

unui obiect conduce la o constantă; în cazul listelor sau parametrilor - la un nume. Legarea se traduce, în ultimă instanţă, prin stabilirea corespondenţei între identificatori şi adrese. Această corespondenţă este stabilită parcurgând un şir de etape consecutive: se trece de la identificator la adresă conform unui set de relaţii, care se numeşte cale de acces (avem, deci, încă o definiţie a noţiunii cale de acces, care este echivalentă celei introduse anterior).

5.1.2.1. Construirea căii de acces

Fie F_1 şi F_2 doua funcţii de acces, F_1 permiţând accesul de la O_1 la O_2, iar F_2 - de la O_2 la O_3: $O_1F_1O_2$ şi $O_2F_2O_3$, accesul de la O_1 la $_OO_3$ fiind realizat prin compunerea acestor funcţii. Calea de acces de la O_1 la O_3 poate fi construită prin metoda **substituţiei** sau prin metoda **înlănţuirii**.

Metoda substituţiei stabileşte o nouă funcţie de acces F_3, obiectul O_3 fiind accesat direct de la O_1: $O_1F_3O_3$. Metoda are avantajul că accesul este rapid, dar şi dezavantajul că O_2 este iremediabil pierdut.

Metoda înlănţuirii cere ca la fiecare accesare a obiectului O_3 pornind de la O_1 să fie parcursă calea $O_1F_1O_2F_2O_3$. Nu se pierde nici o informaţie, dar accesul este mai lent.

Unele obiecte pot fi legate la faza de compilare (de exemplu, obiectele private ale unui program), pentru altele calea de acces este stabilită într-o fază ulterioară (obiectele externe şi parametrii sunt obiecte **libere** după compilare). Pentru un obiect extern compilatorul creează un obiect legătură al cărui conţinut este un şir de caractere ce alcătuiesc identificatorul plus o informaţie care permite regăsirea tuturor numelor ce desemnează această legătură. Operaţia de legare a externilor se numeşte **editare de legături**. Ea poate fi realizată într-o fază distinctă, premergătoare fazei de execuţie (static), sau în faza de execuţie (dinamic), când se face prima referinţă la obiectul extern. Editarea legăturilor se face prin înlănţuire sau prin metoda substituţiei. În primul caz numele externului este la legătura prevăzută de compilator şi această legătură este păstrată în continuare, iar în cel de-al doilea - numele externului este plasat în toate amplasamentele indicate de legătură, după care aceasta este distrusă.

La acest nivel prin **segment** subînţelegem un ansamblu de amplasamente consecutive în care sunt reprezentate obiecte de acelaşi tip, cu aceeaşi durată de existenţă şi cu acelaşi grad de protecţie. Segmentul este cea mai mică unitate care poate fi partajată şi poate conţine obiecte compuse - un masiv, un fişier, o stivă sau o procedură - accesibile unui proces la un anumit moment. Un descriptor, reprezentat într-un amplasament al segmentului, defineşte segmentul, iar numele amplasamentului care conţine descriptorul este chiar numele segmentului.

Obiectului **procedură** îi sunt asociate mai multe noţiuni: **modul sursă, modul obiect** sau **modul executabil**. Modulul sursă al unei proceduri este textul acesteia scris de către programator într-un limbaj de programare şi care va fi tratat de către compilator. Modulul obiect al procedurii este obţinut la ieşirea compilatorului, deci este un produs al compilatorului. Modulul obiect este reprezentat într-un segment sau într-un fişier, destinat interpretării (după editarea legăturilor, la necesitate) de către procesor ca instrucţiuni, valori etc., fiind manipulat în consecinţă. Pentru a separa gestiunea resurselor de administrarea informaţiei, interferenţă ce are loc din mai multe motive (memorii limitate din punctul de vedere al capacităţii, timpi de acces foarte variaţi, execuţia instrucţiunilor numai atunci când se află în memoria operativă, ceea ce impune un grad de mobilitate al obiectelor pe suporturile fizice etc.), s-a introdus noţiunea de **memorie fictivă**: memorie operativă ipotetică suficient de mare pentru a conţine toate obiectele sistemului. Memoria fictivă este asociată sistemului, iar **memoria virtuală** este proprie procesului.

5.1.2.2. Structura reprezentărilor. Descriptori

Schema de mai jos poate fi utilizată pentru obiecte elementare, cum ar fi valori întregi, reale sau caractere, reprezentarea cărora cere un singur amplasament şi pentru care funcţiile de acces sunt reprezentate direct prin instrucţiunile maşinii fizice. Se va mai ţine cont de următoarele două aspecte:

1) pot exista obiecte compuse, structura internă a cărora poate fi complexă, lungimea poate varia pe perioada timpului de existenţă a obiectului,
2) poate fi necesar să se realizeze unele funcţii complexe de accesare a obiectelor compuse.

Cum a fost menţionat, numele obiectului compus se află într-un amplasament al cărui conţinut este **descriptorul** obiectului. Utilizarea unui descriptor pentru a accesa un obiect impune trecerea printr-un cod de accesare care va interpreta acest descriptor şi prezintă următoarele avantaje:

1) în cazul în care obiectul este pasat ca parametru unei proceduri este suficient să fie transmis descriptorul sau adresa acestuia: este mult mai simplu să administrezi informaţii de volum fix şi cunoscut,
2) descriptorul constituie un "punct de trecere" impus pentru accesarea reperată a obiectului şi, ca rezultat, poate servi la implementarea unor controale de accesare, de măsurare, etc.,
3) descriptorul asigură un acces indirect la obiectul reperat, ceea ce permite modificarea dinamică a căii de acces (substituirea unui obiect printr-un alt obiect) sau deplasarea unui obiect în memorie fără recompilarea modulului sursă,
4) partajarea obiectelor între mai mulţi utilizatori cu drepturi sau proceduri de accesare diferite poate fi realizată construind tot atâţia descriptori, câţi utilizatori există; aceşti descriptori reperează acelaşi obiect şi au aceleaşi informaţii de localizare fizică.

Exemplul 5.1. Un tablou este de obicei reperat de un descriptor, care conţine adresa de început a implantării sale în memorie, numărul de amplasamente, ocupate de un element, numărul dimensiunilor şi pentru fiecare dimensiune, valorile maxime ale indicilor. Aceste informaţii permit calcularea adresei de implantare a fiecărui element al tabloului; ele mai permit să se controleze dacă indicii introduşi aparţin intervalului permis. ◄

Noţiunea de descriptor este larg folosită în sistemele de gestiune a fişierelor (v. 5.2 şi cap. 7). O altă utilizare este realizarea mecanismului de adresare segmentată. Segmentarea este o tehnică elementară de structurare a softului. Putem spune, că **segmentul** este un obiect compus, de lungime variabilă, reprezentarea în memorie a căruia ocupă o suită de amplasamente consecutive; el permite ordonarea obiectelor, care se află într-o relaţie logică. Segmentarea permite utilizatorului să-şi organizeze programele şi datele sub forma unui ansamblu de segmente, fără să se preocupe de implantarea lor fizică. De exemplu, fiecare procedură a unui program complex poate ocupa un segment distinct. În calculatoarele cu adresare segmentată fiecare segment este reperat de un descriptor, numele acestor descriptori fiind direct interpretate de procesor.

5.1.2.3. Contexte şi medii

Mulţimea obiectelor accesibile unui proces variază în timp. Iată câteva considerente:

1. *Decompoziţia aplicaţiilor.* Metodele de decompoziţie, utilizate pentru structurarea unei aplicaţii complexe, definesc componentele (module, proceduri, etc.). Fiecărei componente i se asociază o mulţime distinctă de obiecte accesibile.
2. *Gestiunea dinamică*. Mulţimea obiectelor accesibile unui proces îşi poate modifica compoziţia din considerente, legate chiar de natura aplicaţiei: obiectele pot fi create sau distruse în timpul execuţiei.
3. *Protecţia*. O modalitate simplă de a împiedica un proces să aştepte un obiect, accesul la care îi este interzis, este de a suprima toate căile de acces ale procesului spre acest obiect pentru durata interdicţiei. Vor fi evitate în acest fel cheltuieli exagerate la execuţie.
4. *Eficacitatea*. Dacă un obiect este căutat într-o mulţime de alte obiecte, căutarea este cu atât mai eficace cu cât mulţimea are mai puţine elemente.

Trebuie, deci, să luăm în consideraţie atât posibilitatea evoluţiei dinamice a mulţimii obiectelor, cât şi a căilor de acces la aceste obiecte. Introducem pentru aceasta noţiunile care urmează.

Vom numi **lexică** o mulţime de identificatori. Mulţimea obiectelor, desemnate de identificatorii lexicii la un moment de timp dat, se numeşte **context** asociat la această lexică (adică mulţimea obiectelor pentru care există o cale de acces pornind de la unul dintre aceşti identificatori). Starea de execuţie a unui context este starea mulţimii obiectelor, care constituie acest context.

Fiind dată doar lexica nu putem defini un context: mai trebuie să fie specificate regulile de interpretare, care vor fi aplicate identificatorilor din cadrul lexicii. Vom numi **mediu** mulţimea formată dintr-o lexică şi informaţiile (programe, date, reguli de interpretare) necesare la utilizarea acestei lexici. Aceste informaţii pot lua diferite forme în dependenţă de limbajul utilizat (limbaj de comandă, limbaj de programare).

Vom numi **accesibilitate** a unui identificator într-un program regiunea programului în care acest identificator este valid, adică poate fi utilizat ca origine a unei căi de acces. Altfel spus, un proces poate utiliza acest identificator pentru a desemna un obiect atunci când el execută partea în cauză a programului.

Noţiunea de **durată de existenţă** sau **de viaţă** a unui obiect poate fi extinsă şi pentru căile de acces, înţelegând prin aceasta perioada de timp în care acestea există (intervalul de timp care separă crearea de distrugere).

Atunci când un proces execută un program, mulţimea obiectelor la care procesul are acces este definită pentru orice moment de timp, aplicând identificatorilor valizi în aceste momente de timp regulile de interpretare, specificate de mediul curent: regăsim noţiunea de context pentru execuţia unui proces, introdusă în capitolul 3.

Exemplul 5.2. Fie procesul (presupus unic) asociat utilizatorului unui sistem interactiv. În mediul, definit de interpretorul limbajului de comandă, lexica conţine numele fişierelor accesibile utilizatorului. Atunci când utilizatorul comandă execuţia unei proceduri, mediul se modifică: lexica conţine identificatorii definiţi în interiorul procedurii de regulile de accesibilitate ale limbajului şi interpretaţi conform regulilor proprii acestui limbaj.

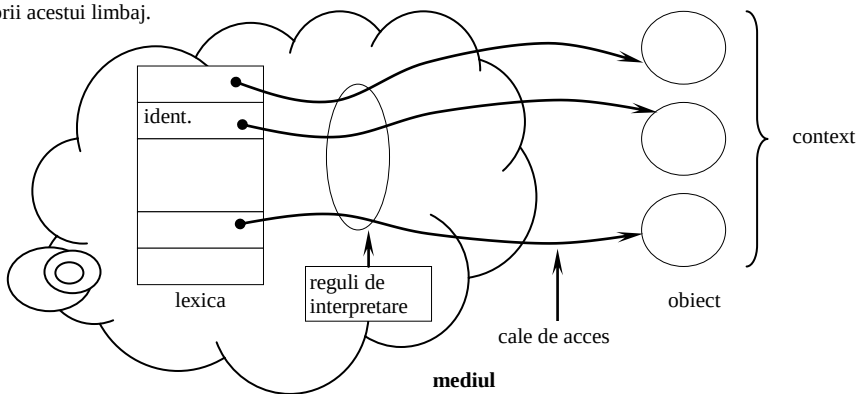


Fig.5.3. Contextul de execuţie a unui proces

Conform specificaţiilor sistemului de operare, identificatorii fişierelor pot sau nu pot continua a fi utilizabili (conform regulilor proprii de interpretare). ◄

Starea de execuţie a unui proces (“valoarea” obiectelor contextului său) se poate modifica la execuţia fiecărei instrucţiuni, însă conţinutul contextului său (identitatea obiectelor care-l formează), se schimbă cu o frecvenţă mai mică. Iată evenimentele principale care pot modifica conţinutul contextului unui proces:

1) Schimbarea mediului, implicând o modificare a compoziţiei lexicii şi, eventual, aplicarea unor reguli de interpretare noi: apel de procedură, intrarea într-un bloc nou (într-un limbaj cu structură de blocuri), schimbarea catalogului curent (într-un limbaj de comandă).
2) Modificarea explicită a căii de acces, pornind de la un identificator al lexicii: asocierea unui fişier sau unui periferic unui flux de intrare-ieşire.
3) Crearea sau distrugerea explicită a unui obiect desemnat de un identificator din lexică: crearea sau distrugerea unui fişier, alocarea sau eliberarea unei variabile administrate dinamic.

Examinând aceste cazuri putem diferenţia durata de viaţă a unui obiect, a unui identificator, care desemnează acest obiect şi cea a unei căi de acces, care conduce la obiectul în cauză. Sunt posibile diferite situaţii: un identificator poate fi legat succesiv de diferite obiecte; reciproc, un obiect poate succesiv (sau simultan) să fie desemnat de mai mulţi identificatori diferiţi; un obiect poate deveni inaccesibil (nici o cale de acces nu conduce la el). Existenţa obiectelor inaccesibile pune problema recuperării spaţiului ocupat de acestea: tehnici speciale de “adunare a fărămiturilor”, permit rezolvarea acestei probleme.

Exemplul 5.3. Cu titlu de exemplu vom indica diferite clase de obiecte, accesibile unui proces în cursul execuţiei unei proceduri, exprimată într-un limbaj de programare de nivel înalt. Aceste clase diferă din punct de vedere a duratei de viaţă, duratei legăturii şi modului de partajare a obiectelor.

1) *Obiecte interne:* acestea sunt instrucţiunile, care compun textul procedurii. Ele sunt desemnate de etichete, utilizate pentru instrucţiunile de ramificare. Durată lor de viaţă coincide cu durată de viaţă a procedurii.
2) *Obiecte locale:* acestea sunt variabilele, declarate în interiorul procedurii. Aceste obiecte sunt create la fiecare apel al procedurii şi distruse la retur. În cazul unui apel recursiv, un exemplar nou al fiecărui obiect local este creat la fiecare apel şi identificatorul său desemnează ultimul exemplar creat (celelalte rămânând inaccesibile până la returul la nivelul corespunzător).
3) *Obiecte remanente şi obiecte globale:* acestea sunt obiectele care existau deja la apelul procedurii şi care vor supravieţui la retur; durata lor de viaţă este fie cea a procesului (obiecte remanente), fie cea a unei proceduri, care înglobează procedura dată (obiecte globale).
4) *Obiecte externe:* sunt obiectele construite şi păstrate independent de procedura şi procesul considerat (alte proceduri, fişiere, etc.). Durata lor de viaţă nu depinde de cea a procedurii sau a procesului; ele pot fi create sau distruse în mod dinamic în timpul execuţiei procedurii.
5) *Parametri:* parametrii formali sunt identificatori, utilizaţi în interiorul procedurii şi care sunt legaţi doar în momentul apelării. Obiectele legate de acestea sunt numite parametri efectivi sau activi; parametrii efectivi sunt furnizaţi de către procedura apelantă sau sunt obiecte externe. Legătura dintre parametrii formali şi cei efectivi poate lua diferite forme în dependenţă de regulile definite în limbajul de programare: apelare prin nume, prin valoare, prin referinţă. Aceste forme diferă prin momentul stabilirii şi permanenţei legăturii. Legătura dispare la returul din procedură. ◄

În cazul în care mai multe procese partajează o procedură, fiecare proces posedă un set propriu de obiecte locale, remanente, globale. Obiectele externe sunt în exemplar unic, ca şi textul (invariant) al procedurii.

5.1.3. Legarea

Numim **legare** procesul construirii unei căi de acces. Acest proces acoperă o varietate mare de situaţii, care vor fi analizate din mai multe puncte de vedere: natura relaţiei de desemnare, momentul legării, permanenţa legăturii. Vom prezenta mai jos şi tehnicile principale utilizate.

Legarea obiectelor unui program poate fi efectuată la diferite momente de viaţă a programului în sistem:

1) *În momentul scrierii programului*. Este cazul unui program scris direct în cod binar când fiecare obiect este desemnat prin adresa absolută a amplasamentului, care-l conţine. Un atare program poate fi imediat executat, dar orice modificare este dificilă şi conţine un risc ridicat de eroare. În practică, unicele obiecte legate la etapa scrierii programului sunt notaţiile universale, care desemnează constantele.

2) *La una din fazele de translatare* (asamblare sau compilare). Legătura este definitivă şi identificatorii sunt înlocuiţi prin adrese absolute. Dezavantajul este că programul nu poate fi deplasat în memorie fără a fi recompilat (dacă nu există mecanisme de translatare a adreselor). De asemenea, legătura stabilită la translatare nu este decât parţială[*]: identificatorii nu sunt înlocuiţi de adrese absolute, ci relative începând cu originea programului (deplasare).

3) *La o fază de încărcare şi editare a legăturilor*. Faza încărcării are drept scop înlocuirea adreselor relative prin adrese absolute, fixând originea programelor în memorie. Faza editării legăturilor are ca scop stabilirea legăturii referinţelor externe (să ne amintim, că vorbim despre identificatorii, care desemnează obiecte construite sau păstrate independent de programul în cauză). Încărcarea şi editarea legăturilor pot fi combinate într-o singură operaţie sau realizate separat. Algoritmii utilizaţi sunt descrişi în 5.3.

O operaţie analogică editării legăturilor este cea a legării fluxurilor de intrare-ieşire cu fişierele sau perifericele. Ca şi editarea legăturilor, această operaţie, descrisă în 5.2.3.2 poate fi realizată în prealabil sau în timpul execuţiei.

[*] Cu excepţia sistemelor de înlănţuire a programelor, numite “compile and go” în care programele sunt traduse şi executate imediat; aceste sisteme sunt prevăzute pentru programe scurte, destinate unei singure execuţii.

4) *În timpul execuţiei; legătura este numită în acest caz **dinamică***. Există mai multe considerente de retardare a legăturii până la faza de execuţie:

- informaţiile necesare pot fi cunoscute doar la această fază, în particular dacă obiectele desemnate sunt create dinamic, fiind, deci necunoscute la momentul compilării,
- calea de acces trebuie modificată în timpul execuţiei: este cazul când un identificator este reutilizat pentru a desemna succesiv mai multe obiecte (de exemplu, flux de intrare-ieşire reasociat mai multor fişiere diferite),
- mecanismul interpretării impune o legătură dinamică: poate fi cazul, de exemplu, al variabilelor locale a unei proceduri recursive, adresa cărora poate fi stabilită doar în cursul execuţiei.

Legarea dinamică constă, cel mai frecvent, în completarea unei căi de acces existente deja parţial.

Tabelul, care urmează, aduce un rezumat al caracteristicilor legăturii în dependenţă de momentul stabilirii acesteia.

Tabelul 5.1. Caracteristicile legăturilor

Legătura	Condiţii	Caracteristici
Devreme	Informaţiile sunt cunoscute din timp Condiţii de execuţie invariante	Eficienţă în execuţie Cale de acces nemodificabilă
Târzie	Informaţii cunoscute parţial Condiţii de execuţie variabile	Necesitatea păstrării informaţiilor de legătură Timp mare de execuţie, programe adaptabile

Dacă criteriul principal este comoditatea utilizării unui sistem şi adaptabilitatea lui la condiţii diverse de execuţie (ceea ce este cazul sistemelor interactive), vom fi cointeresaţi să întârziem momentul stabilirii legăturii ("delay binding time"). Motivaţia principală a unei legături, stabilite la etapa iniţială de elaborare a programului, este eficacitatea execuţiei.

5.1.4. Protecţia

Vom înţelege prin termenul **protecţie** mulţimea metodelor şi mecanismelor, care vizează specificarea regulilor de utilizare a obiectelor şi garantează respectarea acestor reguli. Protecţia este asigurată de o combinaţie de dispozitive fizice şi logice.

Există legături strânse între desemnarea obiectelor şi protecţia lor. Să ne reamintim trei observaţii, legate de protecţie:

1) O modalitate simplă de a interzice unui proces orice acces la un obiect constă în suprimarea tuturor căilor de acces la acest obiect, altfel spus, de a le retrage din contextul procesului,
2) Atunci când operaţiile permise asupra unui obiect sunt specificate prin apartenenţa la o clasă sau un tip, este posibilă verificarea prealabilă execuţiei (adică în momentul compilării sau editării legăturilor) dacă obiectul este utilizat conform regulilor specificate,
3) În cazul în care verificarea este făcută în momentul execuţiei, o procedură de acces facilitează implementarea verificării şi reduce riscul unor erori. Această procedură poate fi la nivel logic (un interpretor, de exemplu) sau fizic (un dispozitiv va aproba trecerea mai departe).

Prezentăm mai întâi un model simplu, care va permite să introducem noţiunile principale, necesare studierii protecţiei. Mecanismele, care permit implementarea protecţiei într-un sistem informatic, sunt descrise în 5.4.

5.1.4.1. Domenii şi drepturi de acces

Printre obiectele, care formează un sistem informatic cele care pot acţiona asupra altor obiecte, modificându-le starea, vor fi numite "active". Regulile de utilizare pot fi exprimate specificând **drepturile de acces** ale fiecărui obiect activ, adică mulţimea de operaţii pe care obiectele active sunt autorizate să le execute asupra altor obiecte.

Să precizăm acum noţiunea de obiect "activ", adică să definim entităţile cărora sunt ataşate drepturile. Am putea să legăm drepturile direct de procese, însă această alegere nu permite exprimarea simplă:

1) a posibilităţii evoluţiei dinamice a drepturilor unui proces,
2) a faptului, că mai multe procese pot avea, în aceleaşi circumstanţe, acelaşi set de drepturi.

Pentru a ţine cont de aceste două aspecte a fost introdusă noţiunea de **domeniu** de protecţie. Această noţiune este un caz particular al noţiunii de mediu - un domeniu defineşte:

- o mulţime de obiecte accesibile sau context,
- pentru fiecare dintre aceste obiecte, o mulţime de operaţii permise (drepturi),
- un mecanism, care asigură accesul la aceste obiecte, respectând restricţiile de mai sus.

Un proces este întotdeauna executat într-un domeniu bine definit; contextul său este cel ataşat domeniului, procesul posedând drepturile specificate asupra tuturor obiectelor acestui context. Un proces poate schimba domeniul cu ajutorul unei operaţii particulare (apelare domeniu). Un domeniu este el însuşi un obiect asupra căruia poate fi executată operaţia de apel; alte operaţii sunt definite mai departe.

Exemplul 5.4. Fie o mulţime de procese executate pe un procesor. Pot fi definite două domenii, supervizor şi sclav, conform modului de execuţie, specificat în cuvântul de stare. În modul supervizor, procesul poate executa toate instrucţiunile procesorului; în modul sclav el poate executa doar instrucţiunile neprivilegiate. Un proces poate schimba domeniul, executând instrucţiuni speciale: atunci când procesul este în modul supervizor este suficient să fie încărcat un cuvânt de stare care conţine modul sclave; dacă procesul este în modul sclave el nu poate trece la modul supervizor decât în urma unei devieri sau a unui apel al supervizorului, care conţine modificarea modului. Remarcăm, că în acest caz el poate executa doar programul asociat devierii sau apelului supervizorului şi nu orice program. Anume acest mecanism asigură eficacitatea dispozitivului de protecţie. ◄

Domeniile de protecţie pot fi definite prin mai multe modalităţi; exemplificăm câteva mai jos.

- un domeniu pentru sistemul de operare, unul pentru fiecare utilizator,
- un domeniu pentru fiecare subsistem, care realizează o funcţie particulară,
- un domeniu pentru fiecare mediu (definit, de exemplu, de cuplul (procedură, catalog curent)).

Alegerea depinde de funcţiile cerute şi, cel mai important, de mecanismele disponibile (v. 5.4).

Presupunem pentru început, că există un număr constant de obiecte. Regulile de protecţie pot fi reprezentate sub forma unui tablou bidimensional, numit **matricea drepturilor**. Acest tablou conţine câte o linie pentru fiecare domeniu D_i şi câte o coloană pentru fiecare obiect O_j (notăm, că domeniile, fiind obiecte particulare, apar de asemenea şi în coloane). Caseta (i, j) conţine drepturile pe care le are un proces, care se execută în domeniul D_i, asupra unui obiect O_j.

Tabelul 5.2. Exemplu de matrice de drepturi

	fişierul 1	*fişierul 2*	*periferic*	D_1	D_2	D_3
D_1	<*citire, scriere, executare*>	<*citire, scriere, executare*>	<*alocare, retragere*>	<>	<*apelare*>	<*schimbare drepturi*>
D_2	<*citire, scriere, executare*>	<*citire, scriere, executare*>	<*cerere, eliberare*>	<*apelare*>	<>	<*apelare*>
D_3	<*citire, scriere, executare*>	<*citire, scriere, executare*>	<>	<*nil*>	<*apelare*>	<>

Notaţia *nil* semnifică faptul, că obiectul O_j nu figurează în contextul domeniului D_i, iar notaţia <> defineşte o listă vidă. În ambele cazuri O_j este inaccesibil în D_i; diferenţa apare dacă să vrea să se ofere lui D_i drepturi asupra lui O_j: în primul caz este necesar să se introducă O_j în contextul lui D_i, adică să-l legăm; în cel de-al doilea caz este suficient să extindem lista existentă.

În practică, matricea drepturilor este foarte rarefiată, adică foarte multe casete conţin *nil* sau <>. Din această cauză sunt utilizate alte forme de reprezentare a matricei drepturilor:

1) *Reprezentarea coloanelor*: lista de acces

Lista de acces, asociată unui obiect este o listă $(D_i, <d_i>)$, unde D_i este un domeniu, care conţine obiectul, iar $<d_i>$ este mulţimea drepturilor acestui domeniu asupra lui. De exemplu, dacă un domeniu este asociat fiecărui utilizator al unui sistem în timp partajat, lista de acces a unui fişier conţine pentru fiecare utilizator lista operaţiilor pe care el este autorizat să le întreprindă asupra fişierului.

O metodă frecvent utilizată pentru a reprezenta mai compact listele de acces constă în specificarea pentru un obiect a unor drepturi implicite (default) pe care le posedă fiecare domeniu. De exemplu, putem specifica implicit, că orice fişier este accesibil doar pentru lectură fiecărui utilizator. Lista de acces va conţine doar cuplurile $(D_i, <d_i>)$ pentru care drepturile diferă de cele implicite.

2) *Reprezentarea liniilor*: lista de drepturi şi capacităţi

Lista drepturilor asociată unui domeniu este o listă $(O_j, <d_j>)$ în care O_j desemnează un obiect, care figurează în contextul domeniului, iar $<d_j>$ este mulţimea drepturilor domeniului asupra lui O_j. Un proces, care este executat în domeniul considerat, primeşte această listă de drepturi; la fiecare accesare a unui obiect mecanismul de accesare trebuie să verifice că operaţia curentă este licită, adică este în $<d_j>$. Din considerente de eficacitate, este de dorit ca acest mecanism să fie cablat. Forma cea mai primitivă în acest sens este bitul supervizor-sclave a cuvântului de stare a unui procesor. O formă mai evoluată este **capacitatea** (v. 5.4.3), care reuneşte într-o structură unică de date, interpretată la fiecare accesare, adresa unui obiect şi mulţimea drepturilor permise. În acest fel, lista drepturilor unui domeniu este lista capacităţilor.

O operaţie importantă este cea de schimbare a domeniului, care permite unui proces să-şi modifice mediul şi drepturile asupra obiectelor. Această schimbare a domeniului ia adesea forma unui apel de procedură, atunci când procesul trebuie să revină în domeniul său iniţial. Pentru a garanta respectarea regulilor de protecţie, trebuie luate măsuri de precauţie de fiecare dată, când are loc extinderea drepturilor. Această circumstanţă se poate produce în egală măsură atât la apel, cât şi la retur. Pentru controlarea operaţiei de schimbare, se cere ca apelul şi returul domeniului să se facă în mod exclusiv prin execuţia unor proceduri speciale (**ghişeu** de apel sau de retur), programele cărora garantează respectarea regulilor specificate. Schimbarea domeniului pe o altă cale decât cea a ghişeului este interzisă de mecanismele de protecţie a domeniului.

Exemplul 5.5. În cazul domeniilor, definite de modurile supervizor şi sclav, ghişeele de apel, în modul sclav, sunt definite prin devieri şi apelări a regimului supervizor, care realizează primitivele interfeţei sistemului de operare. Programul acestor primitive, care este protejat, verifică validitatea parametrilor transmişi. Deoarece drepturile în modul supervizor sunt totdeauna mai mari decât cele din modul sclav, ghişeul de retur este inutil şi primitivele sunt executate în totalitate în modul supervizor; ghişeul devine necesar, dacă aceste primitive trebuie să apeleze o procedură în regimul sclav, deoarece are loc o extensie de drepturi la retur. ◄

Vom găsi în 5.4.2 şi 5.4.3 o cercetare mai detaliată a mecanismelor de protecţie şi câteva exemple de utilizare a acestora în sistemele de operare.

5.1.4.3. Problemele protecţiei

Prezentăm succint câteva probleme, legate de implementarea protecţiei în sistemele informatice.

1) Protecţie ierarhizată

În această situaţie simplă vrem să protejăm un subsistem *A* contra erorilor sau acţiunilor prohibitive, provenite dintr-un subsistem *B*, care utilizează serviciile subsistemului *A*, dar nu are vre-un drept asupra lui *A*. În acelaşi timp *A* poate accesa fără restricţii toate informaţiile lui *B*. Subsistemul *A* este, de exemplu, un sistem de operare, iar *B* - o aplicaţie. Această problemă este rezolvată prin mecanisme ierarhice cum ar fi regimurile supervizor/sclav, cu trecere obligatorie printr-un apel la regimul supervizor pentru comunicarea între *A* şi *B*. Schema inelelor de protecţie (v.5.4.2.5) este o generalizare a acestui principiu.

2) Subsisteme reciproc suspicioase

Există o suspiciune reciprocă între două subsisteme în cazul în care fiecare specifică, că unele din informaţiile sale trebuie să fie protejate contra erorilor sau acţiunilor prohibitive ale celuilalt subsistem. Această situaţie nu poate fi tratată cu ajutorul unui mecanism de protecţie ierarhic.

3) Acordarea şi retragerea drepturilor de acces

Problema pusă aici este problema modificării dinamice a drepturilor de acces. În particular, este posibil să se dorească să avem posibilitatea de a extinde sau restrânge drepturile unui domeniu. Când transmiterea drepturilor este tranzitivă (extinderea drepturilor unui domeniu permiţându-i la rândul său să transmită drepturi), poate fi foarte dificil să se cunoască la un moment de timp dat mulţimea drepturilor ataşate unui obiect anume. Apare o problemă, dacă se va decide restrângerea sau suprimarea acestor drepturi pentru toate domeniile sau doar pentru o parte a lor, care pot accede obiectul. O soluţie posibilă constă în introducerea unui pasaj unic (descriptor) la traversarea căruia se vor permite accesările obiectului. Însă această metodă nu permite tratarea cazului unei retrageri selective, pentru care poate fi necesar să fie parcurse legăturile inverse între obiect şi domeniile, care au acces la obiect.

5.2. Desemnarea şi legarea fişierelor şi intrărilor-ieşirilor

Organizarea fişierelor şi realizarea funcţiilor lor de acces vor face obiectul de studiu al capitolului 6. Vom examina aici modul de desemnare a fişierelor, legarea lor cu programele, care le utilizează şi relaţiile lor cu intrările-ieşirile. Nu avem nevoie pentru aceasta să cunoaştem organizarea internă a unui fişier, interesându-ne doar organizarea lui ca un tot întreg în cazul unui limbaj de comandă sau al unui program.

5.2.1. Căi de acces la un fişier

Un fişier este un obiect compus: el posedă un descriptor, care conţine informaţiile, necesare localizării sale fizice şi realizării funcţiilor de acces. Pentru sistemul de operare, numele descriptorului unei fişier (adresa fizică sau indicile într-un tabel de descriptori) permite accesul la fişier. Acest nume al descriptorului, de obicei necunoscut de utilizatori şi rezervat doar pentru sistemul de operare, este numit **nume intern** al fişierului. Descriptorul şi numele intern al fişierului sunt unice. Conţinutul unui descriptor de fişier este detaliat în 6.4.

Un fişier este desemnat de către utilizatorii externi cu ajutorul identificatorilor – nume externe. Aceste nume externe sunt definite într-un mediu comun mai multor utilizatori. Structurile de date, care permit construirea căii de acces la un fişier pornind de la unul din numele sale externe, sunt numite **cataloage** sau directorii (eng., directory). Structura cataloagelor şi interpretarea numelor externe sunt descrise în 5.2.2.

În afara numelor interne şi externe, adesea mai este definit un nume, zis **local** sau **temporar**. Un nume local este definit într-un mediu propriu unui utilizator şi are o existenţă doar temporară (de exemplu, durata de execuţie a unui program). Necesitatea unui asemenea nume rezidă în următoarele:

- Eficacitate: numele locale sunt definite într-un mediu mai restrâns decât numele externe şi interpretarea lor este, deci mai rapidă (cataloagele nu sunt parcurse la fiecare accesare a fişierului),
- Comoditatea utilizării şi adaptabilitatea programelor: acelaşi nume poate fi refolosit pentru a desemna fişiere diferite pentru instanţe de timp distincte, ceea ce permite reutilizarea unui program cu fişiere diferite fără a fi necesar să se modifice textul programului,
- Acces selectiv: interpretarea numelor locale permite introducerea unui acces selectiv la un fişier în dependenţă de utilizator (funcţii de acces sau drepturi de acces diferite).

Legarea numelor locale este realizată prin înlănţuire: un nume local desemnează un descriptor local, care la rândul său, direct sau indirect, reperează descriptorul unic al fişierului.

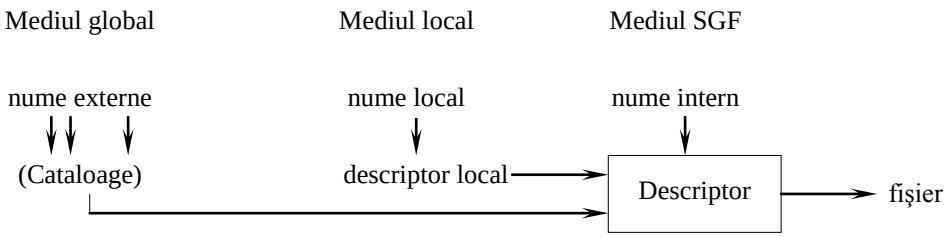


Fig.5.4. Căi de acces la un fişier

5.2.2. Desemnarea externă a fişierelor. Cataloage

Vom examina modul în care un utilizator poate nota fişierele cu ajutorul identificatorilor sau nume externe, şi structurile de date sau cataloagele, care permit determinarea descriptorului fişierului, pornind de la un nume extern, Pentru simplificarea expunerii vom presupune, că descriptorii se conţin direct în catalog; în realitate doar o parte a descriptorului se poate afla în catalog, acompaniată de un nume intern, care permite determinarea restului.

5.2.2.1. Introducere

Un catalog defineşte în sensul 5.1.2, un mediu, adică o mulţime de identificatori (sau o lexică) şi regulile de interpretare a acestor identificatori. Organizarea cea mai simplă poate fi schematic reprezentată de un tabel, care asociază unui identificator descriptorul fişierului pe care acesta îl desemnează. Acest tabel este administrat prin una din tehnicile cunoscute (organizare secvenţială, adresare dispersată, etc.). O astfel de organizare a fost descrisă în 3.3. Ea tratează la acelaşi nivel toate numele, ceea ce implică următoarele inconveniente:

- într-un sistem mono-utilizator ar fi bine să avem la dispoziţie posibilitatea de a clasifica fişierele pe rubrici,
- într-un sistem multi-utilizator conflictele datorate omonimiei a două fişiere de utilizatori diferiţi, restricţionează libera alegere a numelor,
- în toate cazurile, căutarea unui fişier cu numele dat, se efectuează pe toată mulţimea numelor, în timp ce informaţii referitoare la natura sau apartenenţa fişierului ar putea accelera această căutare.

Reieşind din aceste considerente, organizarea pe un singur nivel (fig.5.5, (a)) nu este folosită, cu excepţia unor sisteme mono-utilizator foarte mici. Este preferată cea ierarhică în care fişierele şi cataloagele sunt organizate conform unei structuri arborescente. Această structură se reflectă în structura identificatorilor. Adâncimea arborescenţei poate fi limitată: de exemplu, o structură cu două nivele (fig.5.5 (b)) permite definirea a câte un catalog pentru fiecare utilizator a unui sistem, aceste cataloage fiind şi ele grupate într-un catalog general.

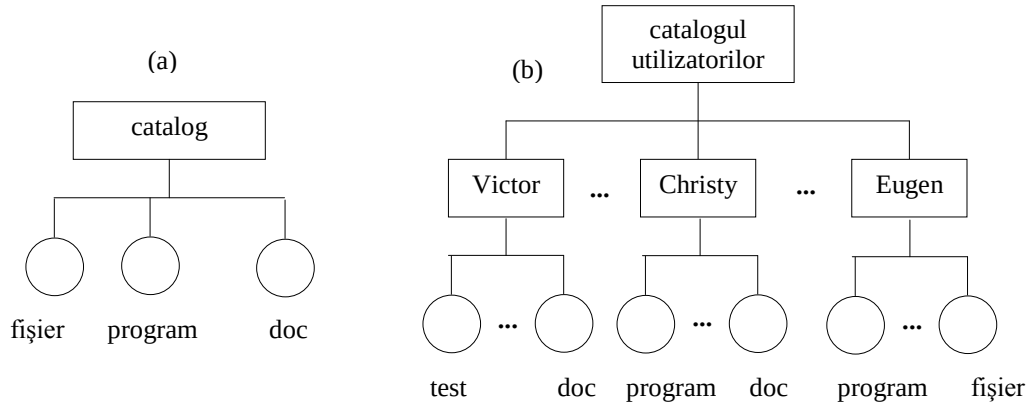


Fig. 5.5. Organizarea fişierelor pe niveluri

Modelul general al organizării ierarhice, prezentat mai jos, a fost propus în sistemul Multics; multe sisteme s-au inspirat de acest model.

5.2.2.2. Organizarea arborescentă

O organizare arborescentă este definită după cum urmează:

- Unui catalog i se asociază o mulţime (care poate fi vidă) de alte cataloage şi fişiere; aceste obiecte se numesc incluse în catalog; ele sunt desemnate aici printr-un identificator zis **nume simple**. Relaţia între un catalog şi un alt catalog, inclus în primul, se numeşte relaţie de legătură; ea permite să se definească un **fiu** (catalogul inclus) şi un tată.
- Există un catalog, şi numai unul singur, numit **rădăcină**, care nu are tată. Orice alt catalog are un tată, şi doar unul singur.

Relaţia de legătură defineşte o arborescenţă de cataloage cu rădăcina în catalogul rădăcină. Sunt utilizaţi termenii de predecesori sau descendenţi pentru a desemna cataloagele la care poate ajunge de la un catalog dat prin intermediul unei suite de relaţii tată sau fiu.

Plecând de la un catalog dat, există o cale unică într-o arborescenţă pentru a ajunge la oricare din descendenţii săi. Această proprietate este utilizată pentru desemnare: numele unui catalog este construit prin concatenarea numelor simple succesive, care desemnează predecesorii săi, începând cu catalogul iniţial; numele unui fişier este concatenarea catalogului, care îl include şi a numelui său simplu. În identificatorii construiţi astfel (zise nume calificate sau compuse) este utilizat un simbol special (“>” în Multics, “/” în Unix sau “/” în MS DOS) pentru separarea numelor simple succesive.

În acest mod putem asocia un mediu fiecărui catalog; lexica acestui mediu este mulţimea numelor, simple sau compuse, construite cum a fost indicat; contextul este mulţimea formată din catalogul considerat, descendenţii săi şi toate fişierele, incluse în aceste cataloage. Mediul şi numele asociate catalogului rădăcină se numesc **universale.**

De exemplu, în sistemul Multics, simbolul ">" este utilizat ca separator pentru construirea numelor compuse; utilizarea separată a acestui simbol desemnează prin convenţie, catalogul rădăcină. La orice moment de timp un catalog curent este asociat fiecărui utilizator. Prin convenţie, mediul utilizatorului este reuniunea mediului definit de catalogul curent şi de mediul universal. Utilizatorul poate desemna orice obiect, utilizând numele universal (care începe cu ">"); cu numele simplu, dar numai obiectele din catalogul curent, sau cu un nume calificat – obiectele incluse în descendenţii acestui catalog.

Acest mod de desemnare permite determinarea oricărui fişier; totuşi, numele universal poate fi prea lung pentru obiectele situate la o mare adâncime în ierarhie. Pentru a permite o desemnare mai simplă a obiectelor, care nu sunt incluse în relaţia de descendenţă a catalogului curent, sunt definite două extensii pentru construcţia numelor:

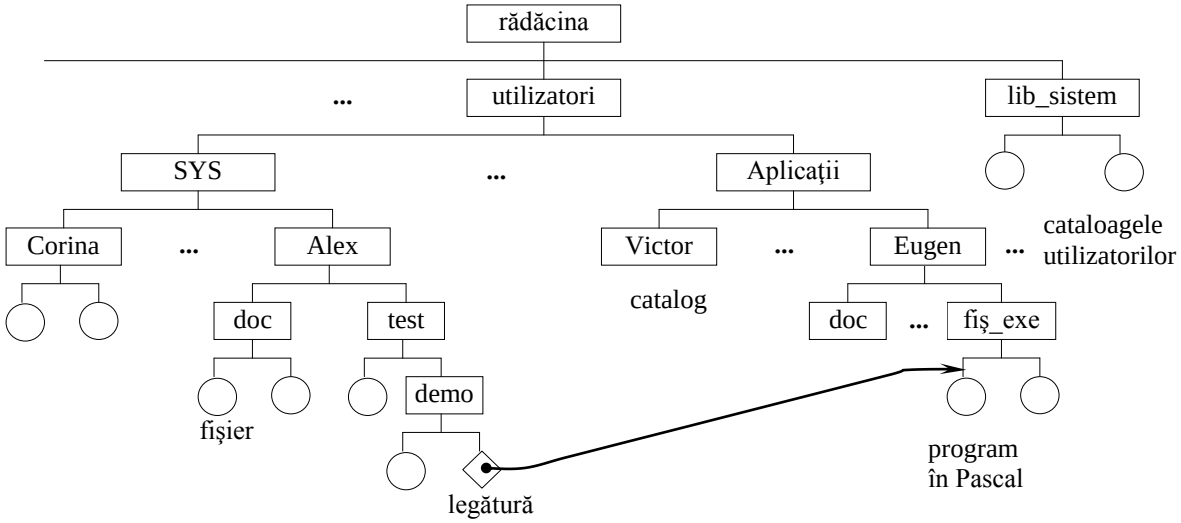


Fig.5.6. Organizare arborescentă a unui sistem de gestiune a fişierelor

1) Desemnarea părintelui. Prin convenţie, un simbol special ("<" în Multics, "**..**" în Unix) desemnează în orice catalog diferit de rădăcină, tatăl catalogului dat. Utilizarea poate fi iterată ("<<" desemnează bunelul, etc.). Pot fi de asemenea desemnate simplu obiecte incluse în cataloagele "frate" sau "verişor" ai catalogului curent.
2) Creare de legături. Numim **legătură** asocierea unui nume simplu (numele legăturii) şi a unui alt nume (nume obiectiv sau scop). Crearea unei legături într-un catalog introduce aici numele simplu (numele legăturii), care trebuie să fie unic în cadrul catalogului dat. Această operaţie este o legare prin înlănţuire: când numele legăturii este interpretat, în mediul catalogului unde a fost creat, el este înlocuit prin numele obiectiv.

Posibilitatea creării legăturilor modifică caracterul pur arborescent al desemnării, ceea ce poate crea probleme delicate necesităţii de a avea mai multe căi de acces la un obiect.

5.2.3. Legarea fişierelor cu fluxurile de intrare-ieşire

Un program schimbă informaţii cu mediul exterior prin intermediul unor operaţii de intrare-ieşire. care permit comunicarea cu un fişier sau un periferic. În momentul elaborării programului nu este încă cunoscut cu care fişier sau periferic se vor produce intrările-ieşirile; adesea este necesar să se utilizeze fişiere sau periferice, care diferă de la o execuţie la altă. Din aceste considerente este util să se poată întârzia legătura unui program cu fişierele sau perifericele pe care acesta le utilizează. În acest scop se introduce noţiunea de **flux** de intrare-ieşire.

Un flux de intrare-ieşire este un obiect, care posedă toate caracteristicile unui periferic de intrare-ieşire (nume, operaţii de acces), dar care nu are o existenţă reală. Pentru a fi efectiv utilizat în transferul informaţiilor, un flux trebuie să fie în prealabil legat, cu ajutorul unei operaţii, numite **asociere**, cu un fişier sau periferic. Transferul informaţiei, descris de operaţiile asupra fluxului, vor fi executate asupra fişierului sau perifericului, asociat fluxului. Asocierea poate fi modificată de la o execuţie la alta, însă textul programului, care face referinţă doar la numele fluxului, rămâne invariant.

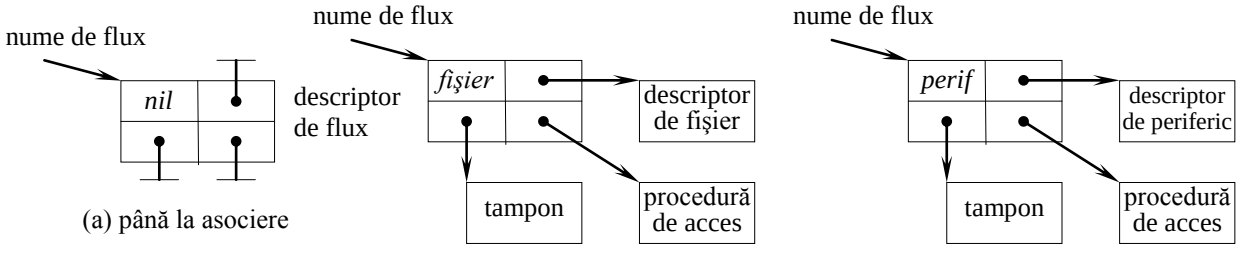


Fig.5.7. Asocierea unui flux de intrare-ieşire

La realizarea asocierii este folosită legarea prin înlănţuire. Fiecărui flux i se asociază un descriptor. Operaţia de asociere cere ca acest descriptor să conţină un pointer spre descriptorul fişierului sau perifericului, pe care îl asociază fluxului. La execuţia unei operaţii de intrare-ieşire asupra fluxului, o direcţionare permite obţinerea suportului corespunzător (fig.5.7).

În schema, prezentată în fig.5.7 am presupus, că descriptorul fluxului permite să se ajungă până la procedura şi zona tampoanelor, utilizate pentru intrări-ieşiri. Legătura respectivă poate fi realizată, în dependenţă de sistem, la asocierea fluxului sau la o fază ulterioară deschiderii.

Asemeni editării legăturilor, asocierea fluxurilor utilizate de un program poate fi realizată la două dintre etapele de derulare:

- înaintea execuţiei programului, cu ajutorul unei instrucţiuni a limbajului de comandă,
- în timpul execuţiei, cu ajutorul unei primitive de sistem, implementată printr-un apel de supervizor.

Sistemele de operare oferă utilizatorilor un set de fluxuri predefinite şi asociate iniţial, în mod implicit, la periferice determinate. De exemplu, un sistem interactiv utilizează un flux standard de intrare şi un flux standard de ieşire; în momentul în care un utilizator este admis în sistem aceste fluxuri sunt asociate, implicit, la tastatură şi ecran. Ele pot fi temporar reasociate fişierelor; în caz de eroare, asocierea implicită este în mod automat restabilită pentru a permite utilizatorului să intervină.

Sistemele oferă de asemenea de obicei posibilitatea de a crea fluxuri noi, care se adaugă celor predefinite. Ca şi asocierea, această creare poate fi cerută de o instrucţiune sau de un apel al regimului supervizor.

Exemplul 5.6. Fluxuri în sistemul Unix. În acest sistem sunt utilizate două tipuri de nume: nume externe, care sunt identificatori construiţi în mod ierarhic, şi nume interne, care sunt numere întregi. Numele externe sunt interpretate într-un mediu global tuturor proceselor, numele locale într-un mediu propriu fiecărui proces. Crearea şi interpretarea numelor locale sunt descrise în 6.6.
Numele locale sunt utilizate şi pentru desemnarea fluxurilor. Prin convenţie, numele locale *0, 1* şi *2* desemnează fluxul standard de intrare, fluxul standard de ieşire şi fluxul mesajelor de eroare, respectiv. Orice instrucţiune este executată de un proces creat în acest scop şi care foloseşte aceste trei fluxuri. Iniţial, fluxurile *0* şi *1* sunt în mod implicit asociate respectiv la claviatura şi ecranul utilizatorului, care a creat procesul.
Sistemul pune la dispoziţie următoarele operaţii asupra fluxurilor:

1) Reasocierea fluxurilor. Dacă *flux_in* şi *flux_out* desemnează, respectiv, un flux de intrare şi un flux de ieşire, iar *id_fişier* este un identificator de fişier, instrucţiunile
   flux_in < id_fişier
   şi *flux_out > id_fişier*
   reasociază, respectiv, *flux_in* şi *flux_out* la *id_fişier*. Dacă *flux_in* şi *flux_out* nu sunt specificate în instrucţiune, ele iau în mod implicit valorile *0* şi *1* respectiv, adică cele corespunzătoare fluxurilor standard. Astfel, instrucţiunea:
   a < sursă > destinaţie
   provoacă execuţia instrucţiunii *a*, după ce în prealabil au fost reasociate fluxurile standard de intrare şi ieşire la fişierele *sursă* şi *destinaţie*.
2) Crearea mecanismelor *pipe*. Un **tub** (**pipe** în Unix) este un tampon, care permite ca două procese să comunice conform schemei producător-consumator. El conţine un flux de intrare (depozit) şi unul de ieşire (retragere) şi poate fi creat în două feluri:
   a) în mod implicit, prin intermediul limbajului de comandă. O instrucţiune de forma
      a | b
      în care *a* şi *b* sunt instrucţiuni elementare cu următorul efect:
      i) să se creeze două procese, fie *proc_a* şi *proc_b*, împuternicite să execute *a* şi *b*, respectiv,
      ii) să se creeze un tub,
      iii) să se reasocieze fluxul standard a *proc_a* la intrarea tubului, iar fluxul standard de ieşire la ieşirea tubului.
   Cele două procese vor funcţiona acum în modul producător-consumator. Pot fi create mai multe tuburi în serie (*a*|*b*|*c*...); este de asemenea posibil înlănţuirea unei suite de instrucţiuni în "**pipe-line**". Această construcţie poate fi combinată cu reasocierea fluxurilor (exemplu: <*sursă a*|*b*...>*destinaţie*).
   b) în mod explicit, prin apelarea regimului supervizor. Un apel al supervizorului permite crearea unui tub şi obţinerea numelor locale pentru intrarea şi ieşirea sa; aceste nume pot apoi fi asociate fişierelor. Aceste nume locale sunt cunoscute procesului apelant şi tuturor descendenţilor lui. ◄

5.3. Legarea programelor şi datelor

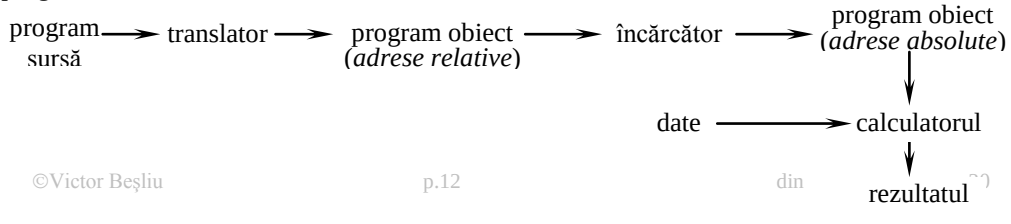
5.3.1. Etapele de viaţă a unui program

Am prezentat mai sus diferite momente în care poate fi stabilită legătura dintre instrucţiunile şi datele unui program. Vom prezenta în rezumat cele mai frecvente scheme înainte de a trece la realizarea lor. Această prezentare vizează, în principal, expunerea folosirii noţiunii de legare, fără a detalia aspectele tehnice ale funcţionării unui încărcător sau editor de legături.

1) Program interpretat

2) Program compilat

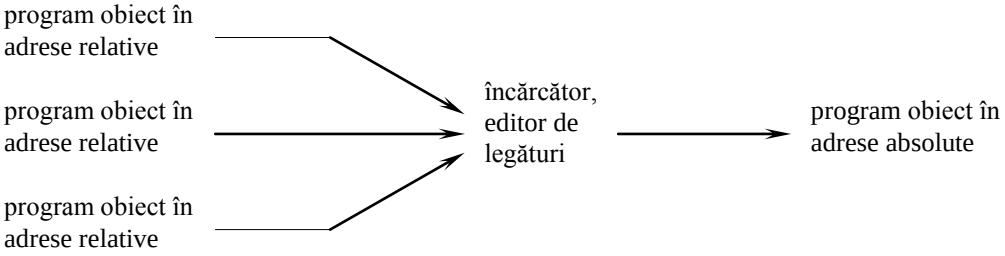
a) program unic



Compararea acestor două scheme pune în evidenţă diferenţa importantă între interpretare şi compilare: modificarea unui program interpretat are efect imediat, în timp ce în cazul compilării suntem nevoiţi să parcurgem toate etapele, începând cu translatarea.

b) program compus

Constituirea unui program unic, pornind de la programe construite independent, se realizează pornind de la programe obiect în adrese relative (deplasabile), adică obţinute după translatare, prin legarea referinţelor externe. Această operaţie poate fi combinată cu încărcarea (ceea ce nu este obligator). O condiţie este ca formatul modulelor obiect în adrese relative să respecte o convenţie comună la care trebuie să se conformeze translatoarele; anume sub această formă programele comune se vor păstra în memorie.



5.3.2. Funcţionarea unui încărcător

Un **încărcător** este destinat să pună în formă absolută un program (sau modul) obiect în adrese relative. Această operaţie constă în înlocuirea tuturor adreselor relative la originea modulului prin adrese absolute. Pentru aceasta se va efectua un lucru preliminar în faza de translatare: înregistrărilor modulului obiect, care conţin o adresă translatabilă, li se va ataşa un indicator, care va fixa poziţia acestei adrese în interiorul înregistrării (dacă poziţia nu este specificată în mod implicit). Adresele translatabile pot să apară:

- sau în câmpul de adresă al instrucţiunii,
- sau în cazul datelor, în "expresii pentru calcularea adresei", destinate a fi utilizate ca relaţii de direcţionare sau să fie încărcate în registrele de bază.

Încărcătorul utilizează metoda substituţiei: orice adresă relativă *a* este înlocuită prin adresa absolută *a+originea*, unde *originea* este adresa absolută începând cu care este încărcat modulul. Pentru un calculator cu registre de bază acest lucru este simplificat considerabil; reamplasarea este realizată la execuţie prin calcularea adresei, cu condiţia că registrele de bază sunt încărcate corect.

Dacă programul încărcat trebuie imediat executat, o funcţie suplimentară a încărcătorului va determina adresa absolută de la care trebuie să înceapă execuţia. Această adresă este adesea fixată prin convenţie (de exemplu, se va porni de la primul amplasament al modulului). Pentru cazuri mai generale, modulul poate avea mai multe puncte de intrare, desemnate cu ajutorul identificatorilor; el va conţine în acest caz un tabel al punctelor de intrare, construit de translator, care asociază o adresă relativă fiecărui identificator. Aceste adrese sunt transformate în adrese absolute; adresa punctului de intrare este determinată pornind de la identificator prin intermediul unei căutări în tabel.

Pentru a ilustra prezentăm un format posibil al unui modul obiect translatabil şi programul corespunzător al încărcătorului.

en-tête	<*id_modul, lungime,...*>
	...
corpul modulului	<*adr, n, r, cod*>
	...
	...
tabelul punctelor de intrare	<*identificator, adresă relativă*>
	...

O înregistrare a corpului modulului este de forma <*adr, n, r, cod*> cu

adr : adresa relativă a codului <*cod*> în modul
n : lungimea lui <*cod*> în octeţi
r : *1* sau *0* (cod translatabil sau nu)
cod : *n* octeţi ai programului.

Algoritmul încărcătorului poate fi următorul:

Parametrii de apel : identitatea modulului (numele fişierului)
: *adr_încărcare* (adresa absolută de încărcare)
: *id_început* (identificatorul punctului de intrare)

```
citire(en-tête);
<verificare dacă lungimea permite încărcarea>

repeat
    citire înregistrare;                -- <adr, n, r, cod>
    if r = 1 then
        translatare(cod)
    endif;
    adr_implantare := adr + adr_încărcare;
    <ordonarea codului  pornind de la adr_implantare>
until corp de modul epuizat

citire(tabel de puncte de intrare);
căutare(id_început,adr_exe);
if eşec then
    <ieşire eroare>                 -- sau alegerea unei adrese implicite
else                                -- implicit
    adr_exe := adr_exe + adr_încărcare
endif;
<ramificaţie la adresa adr_exe>
```

Procedura *translatare(cod)* modifică *<cod>*-ul, translatând adresele relative ale acestuia: fiecare adresă relativă este incrementată cu *adr_încărcare.* Poziţia acestor adrese în interiorul înregistrării trebuie, deci, să fie cunoscută (cel mai des, fiecare înregistrare conţine o instrucţiune unică în care adresa ocupă o poziţie fixă).

Utilizarea unei adresări cu registre de bază simplifică lucrul încărcătorului, translatarea fiind realizată automat, înaintea execuţiei, încărcând într-un registru de bază adresa de implantare a programului. Folosirea a mai multor registre de bază permite reimplantarea independentă a mai multor fragmente de program (de exemplu, procedurile de o parte, datele de alta). Unicele informaţii care trebuie să fie translatate de către încărcător sunt expresiile adreselor, utilizate pentru încărcarea registrelor de bază.

5.3.3. Funcţionarea unui editor de legături

Vom descrie mai jos principiile de funcţionare a unui editor de legături presupunând, că el mai îndeplineşte şi încărcarea. Editorul de legături primeşte la intrare un set de module obiect translatabile şi construieşte la ieşire un modul obiect absolut. El este obligat, deci, pentru fiecare modul obiect:

1) să determine adresa de implantare a acestuia,
2) să îndeplinească modificările informaţiilor translatabile,
3) să realizeze legarea referinţelor externe.

5.3.3.1. Legarea prin substituţie

În cazul editării legăturilor prin substituţie fiecare referinţă la un obiect extern în cadrul unui modul este înlocuită prin adresa absolută a acestui obiect.

Prin definiţie, un modul *A* **utilizează** un modul *B* dacă programul lui *A* face referinţe la obiecte, conţinute în *B.* Deoarece graful relaţiei *utilizează* poate conţine circuite, editarea legăturilor foloseşte un algoritm în doi paşi. Primul pas construieşte planul implantării modulelor şi determină adresa absolută a tuturor obiectelor externe; al doilea pas soluţionează referinţele la aceste obiecte.

Formatul modulelor obiect, indicate în 5.3.2, este completat de un tabel de referinţe externe, care conţine identificatorii tuturor obiectelor externe, folosite de modul; o referinţă externă din program este înlocuită printr-o referinţă la intrarea corespunzătoare a tabelului. Această ordonare permite să fie păstrat doar un exemplar a fiecărui identificator extern şi evită dispersarea acestor identificatori în corpul modulului.

Pentru prezentarea unui algoritm al editorului de legături indicăm un format posibil al modulelor translatabile.

en-tête	*<id_modul, lungime,...>*
	...
tabelul referinţelor externe	*<identificator extern>*
	...
	...
corpul modulului (program)	*<adr, n, r, cod>*

	...
	...
tabelul definiţiilor externe	*<identificator, adresă relativă>*
	...

O înregistrare din corpul modulului are acum următoarea semnificaţie:

adr : adresa relativă a codului *<cod>*
n : lungimea lui *<cod>* în octeţi
r : indicator de reimplantare a adresei
cod : fragment de program sau date
0 : informaţie absolută
1 : informaţie translatabilă (internă)
< 0 : informaţie externă, referinţa n^0-*r* în tabelul referinţelor externe.

Comunicarea între cei doi paşi ai editorului de legături este asigurată de un tabel global de identificatori externi, construit la primul pas, şi care pentru un obiect extern conţine cuplul

(identificator, adresă absolută)

Specificăm două proceduri de acces la acest tabel:

intrare(id, val)	-- introduce cuplul *(id, val)* în tabel
căutare(id, val)	-- caută un cuplu cu identificatorul *id*; în caz de *eşec* (nu a fost găsit) *val* este fără semnificaţie, altfel returnează valoarea asociată lui *val*.

Parametrii de intrare ai unui editor de legături sunt:

- adresa de încărcare a modulului executabil (*adr_încărcare*),
- numele fişierelor modulelor, care se vor lega,
- bibliotecile în care vor fi căutate referinţele nesatisfăcute.

Programul primului pas va fi de forma:

```
adr_init := adr_încărcare          -- adresa începutului modulului
repeat                             -- iteraţii asupra modulului
    citire(en-tête);               -- (id_modul,lungime)
    <tratare en-tête>;
    repeat                         -- iteraţii asupra referinţelor externe
        citire(id_ref);
        <tratare ref_externe>;
    until tabel de referinţe epuizat
    trece(corpul modulului);       -- pas de tratare
    repeat                         -- iteraţii asupra definiţiilor externe
        citire(id_def,adr_def);
        <tratare_def_externe>
    until tabel def epuizat
until mai există module de tratat;
<sfârşit pas 1>
```

În timpul constituirii tabelului externilor valoarea *val=0*, asociată unui identificator semnifică, prin convenţie, că aceasta a fost întâlnită în cel puţin o referinţă, dar nu încă şi într-o definiţie. Avem, deci, o referinţă (provizoriu) nesatisfăcută. Precizăm procedurile:

```
<tratare_ en-tête>:
    căutare(id_modul,val);         -- numele modulului
    if eşec or val=0 then          -- referinţă nouă
        intrare(id_modul,adr_init)
    else
        eroare("definiţie dublă")
    endif;
        baza:=adr_init;            -- începutul modulului curent
    adr_init:=adr_init+lungime     -- începutul modulului următor
    <verificare dacă lungimea permite încărcare>

<tratare_def_externe> :
    căutare(id_def,val);
```

```
    if eşec or val=0 then
        intrare(id_def,baza+adr_def)        -- adresă absolută a definiţiei externe
    else
        eroare("definiţie dublă")
    endif

<tratare_ref_externe> :
    căutare(id_ref,val);
    if eşec or val=0 then
        intrare(id_ref,val)                 -- acţiune vidă, dacă referinţa este deja prezentă
    endif

<sfârşit pas 1>:

pentru toate(id_ref,val) pentru care val=0  -- referinţă nesatisfăcută
    <să se caute în biblioteca specificată modulul, care conţine id_ref şi să se determine
      adresa sa de implantare (acest modul va fi încărcat în memorie cu programul)>
    <să se calculeze val şi intrare(id_ref,val)
```

La trecerea pasului 1 tabelul global al externilor este construit. Cu excepţia cazurilor de eroare, orice identificator al acestui tabel este asociat unei adrese absolute de implantare.

Rolul pasului 2 este de a efectua translatarea adreselor şi încărcarea în memorie, utilizând informaţiile găsite în tabelul externilor.

```
<pasul 2>:
repeat                                      -- iteraţii asupra modulului
    citire(en-tête);
    căutare(id_modul,val);
    baza:=val;
    citire(tabel de referinţe externe);
    repeat                                  -- iteraţii asupra tabelului de referinţe
        <tratare_înregistrare>
    until corpul mudulului epuizat
until mai există module de tratat;
<sfârşit pas 2>;
```

Precizăm procedurile fazei a doua:

```
<tratare_înregistrare>:

adr_implantare:=adr+baza;
if r=1 then                                 -- referinţă internă translatabilă
    <translatare_înregistrare(baza)>
else
    if r<0 then                             -- referinţă externă
        id_ref:=<intrarea n^0 – r în tabelul referinţelor externe>;
        căutare(id_ref,val);
        translatare_înregistrare(val);
    endif
endif;                                      -- terminat, dacă r=0
<încărcare înregistrare de la adr_implantare>

<sfârşit pas 2>:

căutare(id_intrare,val);
if eşec or val=0 then
    eroare("adresă necunoscută de execuţie")    -- sau alegere implicită (default)
else
    adr_exe:=val
endif;
<imprimarea conţinutului tabelului externilor>  -- "diagrama implantării map"
<ramificaţie la adresa adr_exe>
```

Imprimarea tabelului externilor sub forma diagramei de implantare (eng. map) permite semnalizarea erorilor (definiţii duble, referinţe nesatisfăcute) şi facilitarea punerii la punct şi localizării informaţiilor în memorie. Diagrama de implantare este o secvenţă de linii de forma:

<identificator extern> <adresă absolută> <o diagnoză eventuală>

5.3.3.2. Legare prin înlănţuire

1) Vector de transfer

Metoda vectorului de transfer este relativ puţin utilizată pentru programele compilate, totuşi descriem principiile acestei tehnici, deoarece:

- ea este aplicabilă în cazul programelor interpretate sau celor combinate,
- ea permite introducerea într-o formă primitivă a noţiunii de segment de legătură, noţiune dezvoltată în continuare.

Fiecare modul conţine un tabel, numit **vector de transfer**, care are câte o intrare pentru o referinţă externă. În corpul modulului fiecare referinţă la un obiect extern este înlocuită printr-o referinţă *indirectă* la intrarea respectivă a vectorului de transfer. Prima trecere a editării legăturilor are loc ca şi în 5.3.3.1; a doua trecere constă în asocierea intrărilor vectorilor de transfer din diferite module cu adresele corespunzătoare a obiectelor externe. La execuţie, accesul la un obiect extern este, deci, realizat prin direcţionarea via un cuvânt al vectorului de transfer. Această metodă este costisitoare în spaţiu (vectorii de transfer trebuie păstraţi) şi în timp (referire indirectă). Ea convine pentru cazul înlocuirii unui modul într-un program deja legat; anume din această cauză metoda vectorilor de transfer prezintă interes pentru programele interpretate sau pentru sistemele de depanare.

2) Registre de bază

Ca şi în cazul încărcătorului, adresarea cu ajutorul registrelor de bază simplifică lucrul editorului de legături. Există două moduri distincte de utilizare a acestor registre:

1. Registrele de bază sunt accesibile utilizatorilor: în acest caz sunt definite convenţii de utilizare care permit, de exemplu, folosirea registrelor de bază specificaţi pentru a adresa un subprogram, o zonă de date, un bloc de parametri etc. Aceste convenţii sunt implementate de către translatoare sau direct de către utilizatori, dacă aceştia programează în limbaj de asamblare.
2. Registrele de bază sunt administrate de sistemul de operare şi sunt inaccesibili utilizatorilor. În acest caz, sistemul asociază un registru de bază fiecărui fragment independent (program sau zonă de date) şi asigură realocarea acestor registre, dacă numărul lor este insuficient.

În ambele cazuri, lucrul unui editor de legături constă în calcularea expresiilor adreselor, care vor fi încărcate în registrele de bază. În al doilea caz, printre altele, este necesar să fie introduse instrucţiunile de încărcare a acestor registre (sub formă de apel al regimului supervizor) în modulele obiect.

5.4. Mecanisme de gestiune a obiectelor

5.4.1. Segmentarea

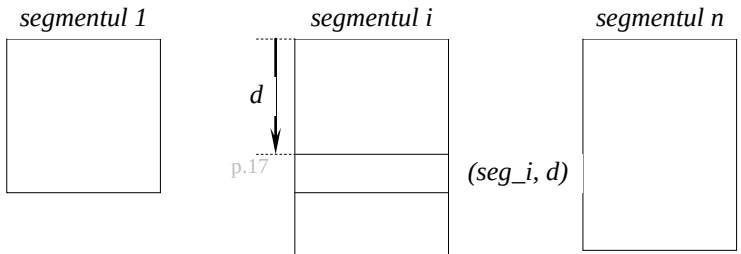
Utilizarea registrelor de bază permite reimplantarea independentă în memorie a procedurilor şi datelor unui program. Totuşi, acest mecanism are două restricţii:

- pentru utilizator: necesitatea de a se conforma convenţiilor de folosire a registrelor de bază,
- pentru sistemul de operare: necesitatea gestiunii alocării registrelor de bază, dacă acestea sunt în număr insuficient.

Aceste restricţii vor fi eliminate, dacă utilizatorul ar putea numi cu un nume la alegere proprie "elementele de informaţie" şi dacă sistemul ar dispune de un mecanism de plasare în memorie şi de legare pentru a administra astfel de elemente. Anume la aceste două întrebări încearcă să răspundă segmentarea.

Un **segment** este p mulţime de informaţii considerată ca o unitate logică şi desemnată de un nume; reprezentarea sa ocupă o mulţime de amplasamente adiacente (contigue). Un segment poate fi de lungime variabilă. În interiorul unui segment informaţiile sunt desemnate cu ajutorul unei **deplasări**, care este o adresă relativă în raport cu începutul segmentului. O informaţie este desemnată de un cuplu (*nume de segment, deplasare*), care se numeşte **adresă segmentată** (fig. 5.8).

Este important să se noteze, că segmentele sunt reciproc independente atât din punct de vedere fizic, cât şi logic. Fizic, un segment poate fi implantat la o adresă oarecare, cu condiţia să fie ocupate adrese adiacente. Logic, diferite segmente, administrate de un sistem de operare, sunt independente şi trebuie considerate ca tot atâtea spaţii de adresare liniare distincte. Chiar dacă numele segmentelor sunt frecvent reprezentate de numere întregi consecutive, nu există o relaţie de adiacenţă între ultimul amplasament al segmentului *i* şi primul amplasament al segmentului *i+1*. Folosirea unei deplasări negative sau mai mare decât lungimea segmentului utilizat este o eroare de adresare.



Segmentele sunt utilizate:

- ca unităţi de secţionare logică a unui program, pentru reprezentarea, de exemplu, a diferitor proceduri, module, date legate logic (tabele, structuri),
- ca unitate de *partajare* între mai mulţi utilizatori,
- ca unitate de *protecţie* (consecinţă a partajării): segmentul este entitatea la care sunt ataşate drepturile de acces.

Realizarea adresării segmentate utilizează principiul, descris în figura 5.9.

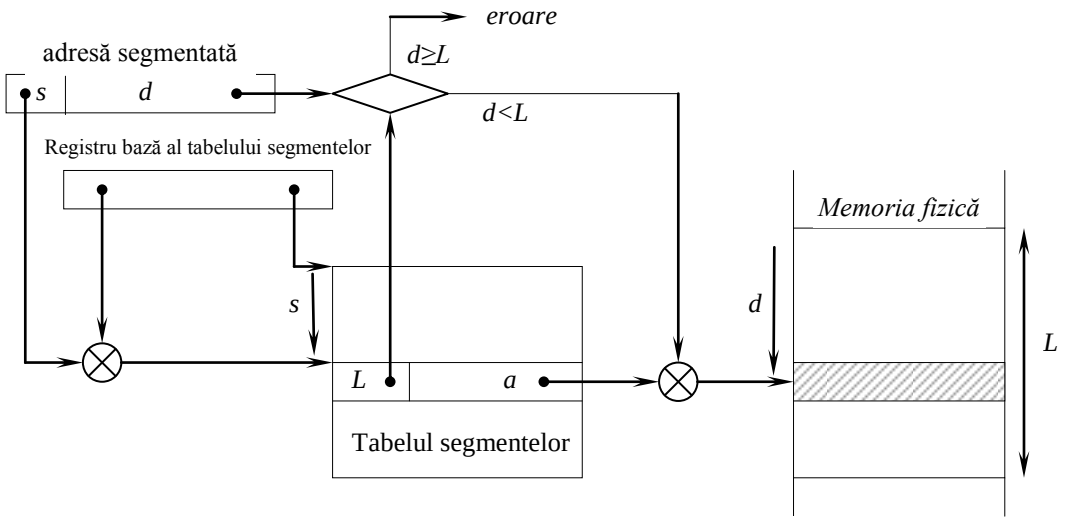


Fig.5.9. Realizarea adresării segmentate

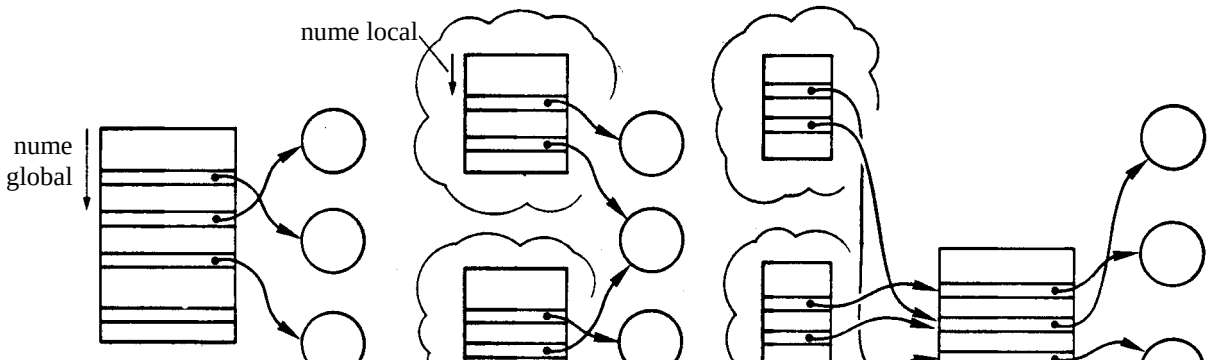
Fiecărui segment îi este asociat un descriptor; el conţine (cel puţin) adresa de implantare a segmentului, lungimea sa şi drepturile de acces. Descriptorii se află în tabelul segmentelor; un segment este desemnat de indicele descriptorului său în acest tabel. Există mai multe posibilităţi de realizare, care sunt dictate de organizarea tabelelor descriptorilor:

1) Tabel unic. Toţi descriptorii sunt păstraţi într-un singur tabel; numele unic al unui segment este indicile descriptorului său din acest tabel. Descriptorul conţine o listă de acces, care defineşte drepturile fiecărui utilizator al segmentului. Această metodă permite partajarea segmentelor, însă toţi utilizatorii sunt obligaţi să utilizeze unul şi acelaşi nume unic, ceea ce adesea nu este comod.

2) Tabele multiple. Există un tabel distinct pentru fiecare mediu; într-un mediu oarecare numele segmentului (dacă este accesibil în acest mediu) este indicele descriptorului său în acest tabel local. Un tabel accesibil în mai multe medii are mai mulţi descriptori şi mai multe nume distincte.

3) Organizare mixtă. Fiecare segment posedă un descriptor central, care conţine caracteristicile sale de implantare fizică (lungimea, adresa). El mai posedă un descriptor local în fiecare mediu în care este accesibil; acest descriptor conţine informaţiile proprii mediului (subliniem drepturile de acces) şi punctează pe descriptorul central. Un segment are în acest mod un nume distinct pentru fiecare mediu, iar caracteristicile fizice sunt situate într-un loc unic, descriptorul central.

Figura 5.10 prezintă aceste moduri de organizare.



5.5. Exerciţii la capitolul 5

Exerciţiul 5.1. Examinaţi problemele, ce ţin de retragerea drepturilor de acces la un obiect într-un mecanism de protecţie. Se doreşte să se permită unui proces proprietar al unui obiect posibilitatea de a acorda şi a retrage selectiv drepturile de acces la acest obiect. Examinaţi separat cazul în care acordarea drepturilor este tranzitivă (un proces poate la rândul său să retransmită drepturile pe care el le-a primit). **Indicaţii**: două mecanisme necesare sunt redirecţionarea şi crearea unor legături inverse.

Exerciţiul 5.2. În schema integrării unui volum amovibil în cadrul unui catalog al unui sistem de gestiune a fişierelor examinaţi următoarele probleme:

- posibilitatea de a crea legături, care să nu treacă prin rădăcina volumului,
- posibilitatea de a crea fişiere multivolum.

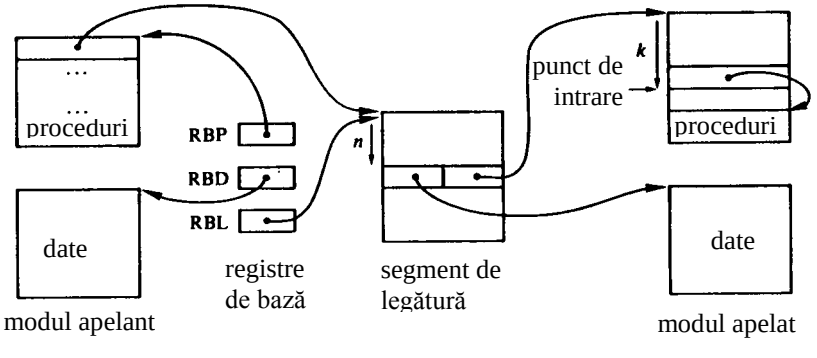
Exerciţiul 5.3. Specificând structura datelor necesare, precizaţi realizarea procedurilor de montare şi demontare a unui volum amovibil.

Exerciţiul 5.4. Dacă vom reprezenta catalogul unui sistem de gestiune a fişierelor printr-un graf nodurile căruia sunt cataloage şi fişiere, iar arcele – relaţiile de desemnare, sistemul de fişiere din Multics (fără a ţine seama de legături) poate fi reprezentat printr-o arborescenţă şi cel din Unix (în care nu există diferenţă între legături şi nume simple) printr-un graf fără circuite (subgraful catalogului rămânând o arborescenţă).
Examinaţi avantajele, inconvenienţele şi problemele de realizare, care vor apare pentru structuri mai generale.

Exerciţiul 5.4. Propuneţi o schemă care ar permite editarea legăturilor într-o singură fază conform principiului lanţului de reluare, utilizat în asambloarele cu o singură fază.

Exerciţiul 5.5. Se propune să se realizeze un mecanism pentru executarea programelor modulare pe un calculator cu adresare segmentată. Un modul conţine o mulţime de variabile globale şi proceduri, care accesează aceste variabile. Procedurile unui modul *M* pot fi apelate dintr-un alt modul, care **utilizează** *M*. Ne vom limita la un proces unic.
Un segment este identificat prin indicele descriptorului său într-un tabel general al segmentelor. Unui modul îi sunt asociate două segmente: unul conţine codul procedurilor sale, altul – variabilele sale globale. Două registre de bază RBP şi RBD conţin numerele acestor două segmente pentru modulul, care este în curs de execuţie. Contorul ordinal CO conţine deplasarea instrucţiunii curente în segmentul procedură. Un segment special serveşte ca stivă de execuţie; parametrii procedurilor sunt pasaţi prin valori în stivă.
Procedurile modulelor, utilizate de un modul *M*, sunt numite externe pentru *M*. Desemnarea lor face apel la un segment de legătură, fiecare amplasament *n* al căruia conţine numerele segmentelor procedură şi date ale unui modul *E*, folosit de *M*. Un registru de bază RBL conţine numărul segmentului de legătură a modulului curent. Segmentul procedură a unui modul *M* este organizat după cum urmează:

- primul amplasament (deplasarea *0*) conţine adresa de bază a segmentului de legătură a modulului *M*,
- următoarele *m* amplasamente conţin deplasările punctelor de intrare a celor *m* proceduri ale modulului,
- restul segmentului conţine codul executabil al procedurilor.



În programul unui modul *M* apelul procedurii cu numărul *k* a modulului, descris de amplasamentul cu numărul *n* al segmentului de legătură este realizat cu ajutorul unei instrucţiuni **call** *n, k.*

1) Permite oare această schemă realizarea mai multor module care ar avea proceduri comune şi variabile globale proprii? Care sunt avantajele unei atare posibilităţi?
2) Descrieţi detaliat secvenţa operaţiilor, care vor fi executate la apelul unei proceduri externe a unui modul şi la returul din pr**call** *n, k*
3) Permite oare această schemă editarea dinamică a legăturilor? Descrieţi structurile de date necesare şi principiul de realizare.

6. Gestiunea fişierelor

Acest capitol este consacrat studierii detaliate a gestiunii fişierelor în sistemele de operare: organizarea logică, reprezentarea informaţiei, realizarea funcţiilor de acces, protecţia şi securitatea. Noţiunile menţionate sunt ilustrate cu exemple de gestionare a fişierelor în sistemele de operare Windows şi Unix.

6.1. Noţiuni generale

6.1.1. Funcţiile unui sistem de gestiune a fişierelor

Numim **fişier** o mulţime de informaţii, formate cu scopul păstrării şi utilizării lor în cadrul unui sistem informatic. Fişierele au de obicei o durată de viaţă superioară timpului de execuţie a unui program sau duratei unei sesiuni de lucru: ca rezultat, suportul lor permanent este memoria secundară. Fişierul este un obiect: el posedă un nume, care permite desemnarea sa, are asociate funcţii de acces, adică operaţii, care permit crearea sau distrugerea, consultarea sau modificarea informaţiilor, etc. Componentele unui sistem de operare, care asigură posibilitatea păstrării fişierelor şi realizează funcţiile de acces se numeşte **sistem de gestiune a fişierelor (SGF).**

Utilizatorul unui sistem informatic organizează informaţiile conform necesităţilor proprii, impunându-le o structură sau organizare, numită **logică**; funcţiile de acces sunt exprimate cu ajutorul acestei structuri. Reprezentarea informaţiilor fişierelor în memoria secundară (adresa de implantare, codificarea informaţiei) determină organizarea **fizică** a fişierelor. **Reprezentarea structurii logice a fişierelor printr-o organizare fizică este sarcina SGF.**

Adesea un fişier este definit ca o colecţie de informaţii elementare de aceeaşi natură, numite **înregistrări** sau **articole** (eng. “items” sau “records”). O înregistrare poate ea însăşi avea o structură mai complicată (este introdusă noţiunea de **câmpuri** pentru desemnarea unor componente). Exemple de organizare logică sunt date în 6.2.

Funcţiile principale, asigurate de un sistem de gestiune a fişierelor sunt:

- crearea unui fişier (definirea numelui, alocarea eventuală a spaţiului),
- distrugerea unui fişier (eliberarea numelui şi a spaţiului, alocat fişierului),
- deschiderea unui fişier (declararea intenţiei de a folosi fişierul cu ajutorul funcţiilor de acces şi a drepturilor specificate),
- închiderea unui fişier deschis (interzicerea oricărui acces ulterior),
- diverse funcţii de consultare şi modificare: citire, scriere, modificarea lungimii, etc. (detaliile legate de aceste funcţii depind de organizarea logică a fişierului).

Aceste funcţii sunt materializate prin operaţiile de bază, prezente în toate SGF. Trebuie să adăugăm aici diverse funcţii detaliile cărora depind de organizarea sistemului de gestiune a fişierelor: specificarea şi consultarea drepturilor de acces, consultarea caracteristicilor (data creării, modificării, etc.), operaţii asupra numelui (schimbarea numelui, etc.).

Păstrarea fişierelor şi realizarea funcţiilor de acces impune SGF să aibă în şarjă:

- gestiunea suporturilor fizice ale informaţiei, ascunzând unui utilizator obişnuit detaliile organizării fizice;
- securitatea şi protecţia fişierelor, adică garantarea integrităţii,confidenţialităţii şi accesibilităţii în caz de accident sau de rea voinţă şi respectarea regulilor stabilite de utilizare (drepturi de acces, condiţii de partajare).

Într-un sistem de operare SGF joacă rolul central, deoarece el trebuie să comande cea mai mare parte a informaţiilor utilizatorilor, cât şi a sistemului propriu-zis. SGF are legături strânse cu sistemul de intrare-ieşire, este adesea convenabil conceptual să nu se facă distincţie între fişiere şi unităţile periferice ca suport sursă sau destinaţie a informaţiei în timpul execuţiei unui program. În sistemele cu dispozitive speciale de adresare şi desemnare a informaţiei, cum ar fi memoria virtuală segmentată sau paginată, SGF trebuie să fie legat de aceste mecanisme, din care cauză pot avea loc confuzii.

6.1.2. Organizarea generală a unui sistem de gestiune a fişierelor

SGF realizează corespondenţa între organizarea logică şi organizarea fizică a fişierelor. Organizarea logică, unică cunoscută de utilizatorii obişnuiţi, este determinată de consideraţii de comoditate şi universalitate. Organizarea fizică, legată de suporturile de memorie utilizate, este determinată de considerente de economisire a spaţiului şi eficacitatea accesului. Aceste două organizări sunt, în genere, diferite fiecare fiind definite prin structuri de date proprii, controlate de SGF. Drept rezultat, SGF sunt construite conform unei structuri ierarhice, care are cel puţin două niveluri corespunzătoare celor două organizări, menţionate mai sus: orice acţiune, definită la nivelul logic este interpretată de un set de acţiuni la nivelul fizic. Pentru facilitarea concepţiei sau pentru a răspunde unor necesităţi specifice (portabilitate, de exemplu) pot fi definite şi unele nivele intermediare.

Pentru separarea organizării fizice de organizarea logică este recomandabil să fie introdusă o organizare intermediară, care joacă rol de interfaţă. Această organizare intermediară poate fi un segment, adică o suită de amplasamente adiacente,

desemnate de valori întregi pozitive, numite adrese logice sau deplasări. Putem separa, în acest caz, în două etape stabilirea corespondenţei între organizarea logică şi cea fizică (fig.6.1):

- traducerea numelor, specificate de funcţiile de acces la fişier, în adrese logice (un atare nume desemnează o înregistrare sau un câmp al acesteia),
- traducerea adreselor logice în adrese fizice

.

Fig.6.1. Adresare logică într-un fişier

Această schemă reprezintă doar un model explicativ, adesea etapa intermediară este scurtcircuitată, funcţiile de acces fiind traduse direct în adrese fizice. Mai notăm, că în cazul unui calculator cu memorie segmentată, reprezentarea fişierelor se reduce la prima etapă: implantarea fizică a segmentelor este în şarja sistemului de operare.

Completând modelul de mai sus cu elementele, legate de desemnarea fişierelor din 5.2, construirea organizării SGF poate fi reprezentată conform schemei din figura 6.2.

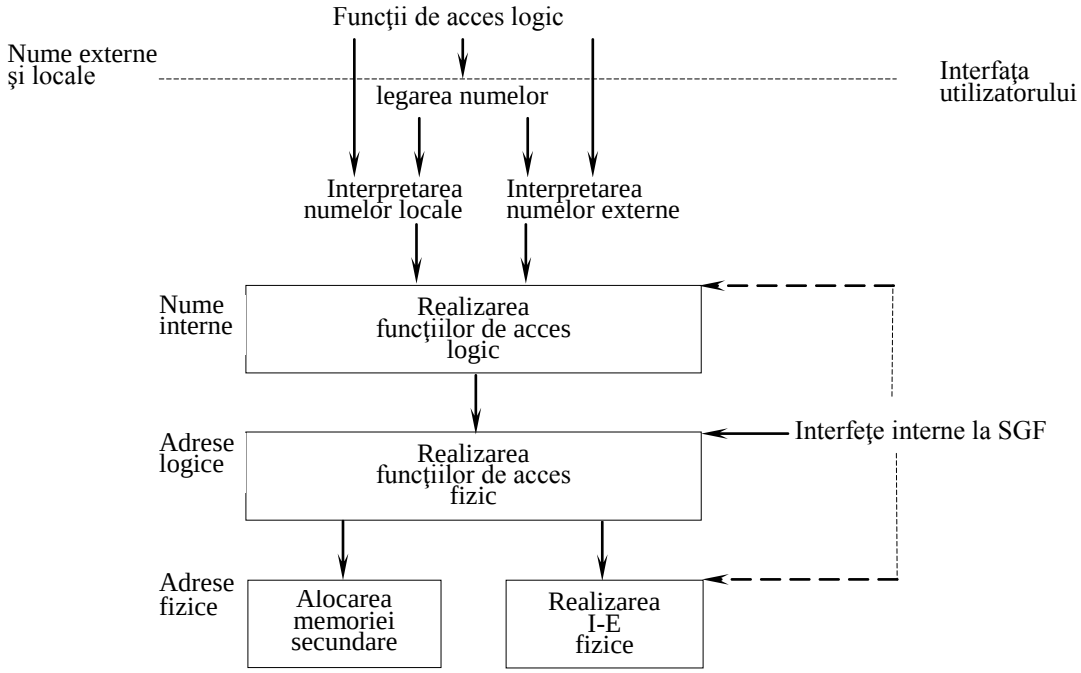


Fig.6.2. Organizarea unui sistem de gestiune a fişierelor

Această schemă nu trebuie să fie considerată drept un cadru rigid, care ţine cont de toate modurile de organizare a SGF, ci doar ca un ghid pentru stabilirea funcţiilor şi structurilor de date mai frecvente. Planul adoptat pentru restul capitolului va urma, într-un mod descendent, ierarhia astfel stabilită.

6.2. Organizarea logică a fişierelor

6.2.1. Introducere

Vom examina acuma modurile principale de organizare logică a unui fişier şi realizarea lor. La această fază a studiului este suficient să traducem în adrese logice sau deplasare localizarea înregistrărilor, manipulate de funcţiile de acces; determinarea adreselor lor fizice este examinată în 6.4.

Pentru specificarea organizării logice a unui fişier definim mai întâi o structură abstractă a fişierului. Pentru aceasta vom utiliza un model simplu de descriere a datelor, care ne va permite să caracterizăm înregistrările, să definim organizarea globală a fişierului şi să exprimăm funcţiile de acces. Vom preciza mai apoi reprezentarea concretă a informaţiilor, care se conţin în fişier.

Un fişier este o mulţime de înregistrări (articole). Fiecărui articol îi este asociat un număr constant de *n* atribute, aceleaşi pentru toate articolele. Un atribut este definit printr-un nume şi un domeniu de valori. Numele este pentru

desemnarea atributului; două atribute distincte au nume diferite. Domeniul de valori specifică valorile pe care le poate lua atributul. Specificarea organizării fişierului este completată de restricţiile, care trebuie să fie satisfăcute de către articole. Aceste restricţii pot lua forme diverse (restricţii de ordine a articolelor, restricţii legate de valorile atributelor, relaţii între atributele diferitelor articole, etc.). Desemnarea articolelor, în expresia funcţiilor de acces, utilizează atributele articolelor, ţinând cont de restricţiile specificate.

Ilustrăm aceste definiţii cu două exemple, care vor fi necesare mai apoi.

Exemplul 6.1. **Fişier secvenţial de caractere.** Fiecare înregistrare a fişierului posedă următoarele atribute:

(*număr, <întreg>*)
(*conţinut, <caracter ASCII>*)

Restricţia este, că articolele fişierului sunt ordonate, articolele succesive având drept *număr* valori întregi consecutive.
În reprezentarea fişierului atributul *număr* nu este reprezentat explicit, dar este definit de ordinea articolelor. Mai mult, atributul *valoare* este reprezentat doar prin valoarea sa, iar fişierul este simplu reprezentat sub forma unei secvenţe de caractere.
Utilizarea atributului *număr* permite specificarea diferitor funcţii de acces. De exemplu, dacă ne vom limita la consultare:

citire(urm) citeşte caracterul, care urmează după o poziţie curentă specificată (poziţia ultimei lecturi); pune valoarea caracterului în *urm.*
citire(i, c) citeşte caracterul cu numărul *i*; valoarea acestuia este pusă în *c.*

În ambele cazuri se va specifica efectul funcţiei, dacă caracterul nu există.
Prima funcţie este de tip *acces secvenţial*, cea de-a doua de tip *acces direct*. ◄

Exemplul 6.2. **Fişier de tip *document*.** Fiecare înregistrare a acestui fişier, folosit pentru gestiunea unei biblioteci, are următoarele atribute:

(*nume_lucrare, <identificatorul lucrării>*)
(*autor, <lanţ de caractere>*)
(*editor, <lanţ de caractere>*)
(*an, <întreg>*)
(*subiect, <lanţ de caractere>*)

O restricţie poate fi că două articole diferite au ca nume identificatori de lucrări distincte (cu alte cuvinte, o lucrare este determinată în mod unic de numele său).

Să examinăm reprezentarea acestui fişier. Dacă stabilim o anumită ordine a celor cinci atribute ale unei înregistrări, numele acestor atribute (care este acelaşi pentru orice articol), nu este necesar să fie păstrat în mod explicit. Fiecare articol este reprezentat de un şir de cinci câmpuri, fiecare având un format prestabilit.

Pot fi specificate funcţii de acces direct, desemnând un articol prin atributul *nume_lucrare* (dacă această desemnare este unică). Pentru a specifica funcţii de acces secvenţial va fi necesar să ordonăm în prealabil articolele cu ajutorul unei restricţii suplimentare (relaţie de ordine, de exemplu, definind o ordine a numelor, autorilor, anului de editare, etc.). ◄

Metodele de organizare logică a fişierelor (reprezentarea datelor, realizarea funcţiilor de acces) sunt aplicaţii directe ale structurilor de date: tabele, fire de aşteptare, liste, etc., care sunt tratate în alte discipline. Ne vom opri la o succintă descriere a organizării uzuale, făcând trimitere la literatura de specialitate pentru studii mai aprofundate. Interesul principal al acestei descrieri este de a elucida restricţiile introduse de organizarea fizică a fişierelor.

6.2.2. Acces secvenţial

În cadrul unei organizări secvenţiale înregistrările sunt ordonate şi pot fi desemnate de valori întregi consecutive. Totuşi, aceste numere de ordine nu pot fi folosite în cadrul funcţiilor de acces; este permisă doar utilizarea funcţiei "succesor". Accesul secvenţial este modul obişnuit de utilizare a unui fişier, implantat fizic pe un suport în care accesarea amplasamentelor este ea însăşi secvenţială, cum ar fi banda magnetică.

Un fişier *f* poate fi deschis pentru citire sau scriere. Unele organizări autorizează scrierea începând de la o înregistrare oarecare (v. exemplul din capitolul 9). Vom considera că scrierea se face la sfârşitul fişierului şi deschiderea pentru scriere iniţializează fişierul în "vid".

Deschiderea pentru scriere este realizată prin operaţia:

```
deschide(mod):
    if mod=citire then
        f.rest:=<şirul înregistrărilor fişierului>
    else
        f:=<vid>
    endif;
    f.mod:=mod;
    avansare
```

unde funcţia *avansare* este definită după cum urmează:

```
if f.rest=vid then
    f.sfârşit:=true
else
```

```
        f.primul:=<pointer pe primul(f.rest)>;
        f.rest:=f.rest-primul(f.rest)
    endif
```

Operaţia de citire a unei înregistrări are forma:

```
citire(f, a):
    if f.sfârşit=false then
        a:=<înregistrarea desemnată de f.primul>;
        avansare
    endif
```

Scrierea adaugă o înregistrare la sfârşitul fişierului:

```
scriere(f,a):
    if mod=citire then
        <eroare>
    else
        alocare spaţiu pentru articol nou;
        if eşec then            -- lungimea maximă a fost atinsă sau este necesar un spaţiu mai mare
            <eroare>
        else
            f.pa:=<pointer pe amplasamentul înregistrării noi>;
            <copiere a în amplasamentul desemnat de f.pa>
        endif
    endif
```

Aceste funcţii de acces se traduc utilizând adresele logice. Dacă articolul curent nu este ultimul în fişier, adresa logică a succesorului său este dată de *adresa(curentă)+lungimea(curentă)*.

Dacă articolele sunt de lungime variabilă, lungimea înregistrării curente poate fi obţinută plecând de la conţinutul acestui articol (de obicei, lungimea unei înregistrări este prezentă explicit).

6.2.3. Acces direct

În cadrul organizărilor cu acces direct funcţiile de acces sunt exprimate ca funcţii ale atributelor înregistrărilor; aceste atribute sunt valori ale diferitor câmpuri. Se numeşte **cheie** orice câmp al unei înregistrări valoarea căruia poate servi la identificarea înregistrării. Conform organizării adoptate, una sau mai multe câmpuri pot servi drept cheie.

6.2.3.1. Cheie unică

Într-un fişier cu cheie unică fiecare înregistrare conţine o singură cheie, care identifică înregistrarea fără ambiguitate; restul înregistrării este informaţia propriu-zisă. Două înregistrări distincte vor avea totdeauna două valori diferite ale cheii. Definim o procedură *căutare(cheie, al)*, care pentru orice valoare a cheii:

- sau pune la dispoziţie adresa logică *al* (unică) a înregistrării pentru care cheia posedă valoarea dată (caz de succes)
- sau semnalizează cu ajutorul unui mecanism special, că o atare înregistrare nu există (caz de eşec): excepţie, cod de condiţie; valoarea *al* este adresa la care înregistrarea ar putea fi inserată (valoare specială, dacă nu mai este spaţiu liber).

Procedura *căutare* serveşte la realizarea funcţiilor elementare de acces direct *citire(cheie, info), adăugare(cheie, info), suprimare(cheie, info), modificare(cheie, info)*. Funcţiile *citire, suprimare, modificare* pot conduce la erori, dacă procedura *căutare* eşuează; procedura *adăugare* generează o eroare, dacă procedura *căutare* se termină cu succes.

Metodele realizării procedurii *căutare* formează obiectul multor cercetări speciale. Aducem aici doar rezultatele principale. Procedura *căutare* poate fi realizată utilizând două metode: adresarea dispersată şi constituirea unui indice.

a) *Adresare dispersată (funcţie "hash")*

Procedura *căutare* este realizată direct, construind o funcţie *al = f(cheie)*. Funcţia *f* se numeşte funcţie de dispersare (funcţie "hash"). Pare a fi o soluţie funcţia identitate: de exemplu, în cazul unei chei cu valoare întreagă, trebuie să luăm în calitate de adresă *al* valoarea cheii (sau de *k* ori această valoare, dacă o înregistrare ocupă *k* amplasamente). Totuşi, această metodă este impracticabilă. Fie, de exemplu, un fişier pentru care cheia are 9 cifre; un atare fişier poate să ocupe maximum 10^9 amplasamente logice. Dacă lungimea medie a fişierului este de 10^5 înregistrări rata de umplere a fişierului cu înregistrări utile este doar de ordinul 10^{-4}. Dacă dorim să evităm utilizarea ineficientă a spaţiului memoriei va trebui să

îndreptăm atenţia spre alegerea funcţiei de corespondenţă a adreselor logice şi celor fizice. Folosirea funcţiei identitate este limitată la cazurile pentru care avem o rată ridicată de ocupare a memoriei.

O funcţie de dispersare ideală realizează o permutare între mulţimea cheilor şi cea a adreselor logice, limitată la numărul de înregistrări din fişier. Pentru simplitate presupunem, că adresele logice pentru un fişier cu *n* înregistrări sunt *0,..., n-1*. Funcţia hash trebuie să posede următoarele proprietăţi:

- pentru orice înregistrare din fişier cu cheia *c:* $0 \le f(c) < n$, (6.1)
- pentru orice cuplu de înregistrări $f(c_1) \neq f(c_2)$, dacă $c_1 \neq c_2$. (6.2)

În practică este foarte dificil să se satisfacă proprietatea (6.2). Din această cauză trebuie admisă posibilitatea coliziunilor, adică existenţa cheilor, care nu satisfac ultima condiţie. Numărul de valori distincte calculate de funcţia de dispersare este inferior valorii lui *n*. În caz de coliziune este necesară o fază suplimentară pentru determinarea înregistrării căutate sau, în cazul unei inserări, să i se caute un loc. Alegerea funcţiei de dispersare în scopul reducerii probabilităţii coliziunilor şi a metodelor de tratare a coliziunilor trebuie să ţină cont de caracteristicile utilizării fişierului:

- probabilitatea unor valori diferite ale cheii,
- frecvenţa relativă a operaţiilor de căutare, inserare şi suprimare a articolelor.

Figura 6.3 prezintă schematic organizarea unui fişier cu acces direct prin adresare dispersată.

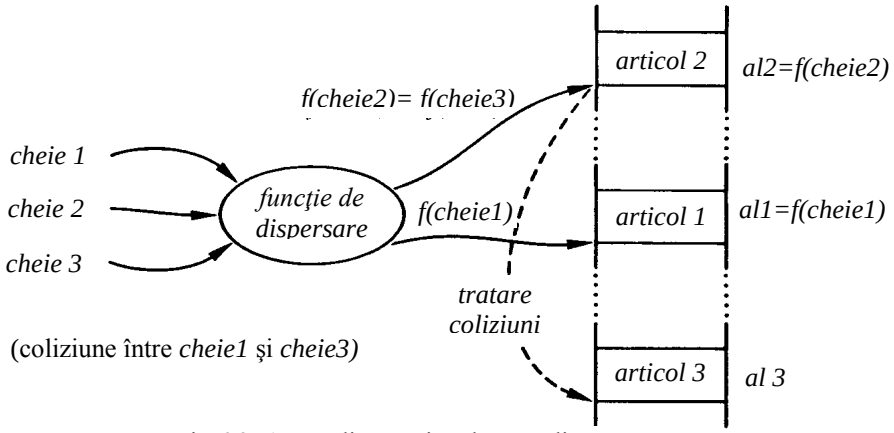


Fig.6.3. Acces direct prin adresare dispersată

Principalul avantaj al metodei adresării dispersate (dacă se reuşeşte să se asigure un nivel acceptabil al coliziunilor) este rapiditatea: în lipsa coliziunilor găsirea unei înregistrări necesită o singură accesare a discului. Totuşi, în cazul cel mai frecvent, când mulţimea cheilor este ordonată, funcţia de dispersare nu întotdeauna asigură o relaţie simplă între ordinea cheilor şi ordinea adreselor logice a înregistrărilor respective. Drept consecinţă, un acces secvenţial, care ar respecta ordinea cheilor, trebuie să fie realizat ca o suită de accese directe, fără simplificări. Metodele accesului indexat permit remedierea acestui inconvenient.

b) *Fişiere indexate*

Metodele accesului indexat sunt utilizate în cazul când mulţimea cheilor este ordonată. Relaţia dintre cheie şi adresa logică este materializată printr-un tabel, numit **tabel al indicilor**, în care ordinea cheilor este semnificativă. Schema principiului organizării indexate este dată de fig.6.4

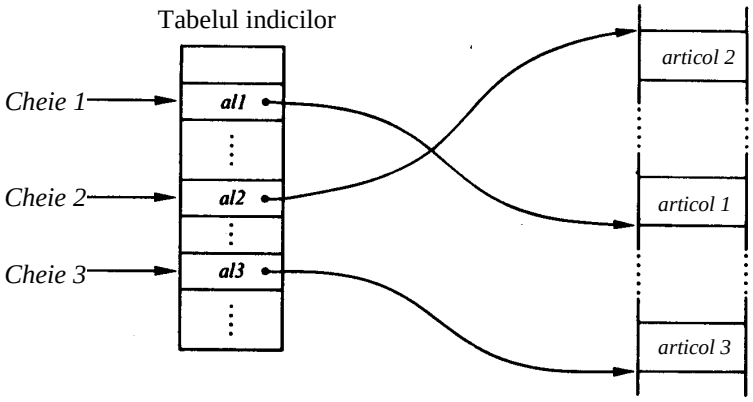


Fig.6.4. Acces direct prin indexare

Schemele utilizate efectiv în practică sunt mai complexe pentru a permite:

- accelerarea căutării în cadrului indicelui,

- facilitarea inserării şi a suprimării înregistrărilor.

Fie *n* numărul de înregistrări într-un fişier. Dacă tabelul indicilor este organizat secvenţial în ordinea cheilor şi căutarea unei chei se face prin parcurgere secvenţială, numărul mediu de comparări (şi de accesare a tabelului) este de ordinul *n/2*. Această valoarea poate fi redusă până la $log_2 n$ adoptând o organizare arborescentă a tabelului ($log_2 n$ reprezintă adâncimea medie a unui arbore binar echilibrat cu *n* vârfuri). Arborele riscă să devină dezechilibrat, dacă există foarte multe inserări şi suprimări; pentru cazul cel mai puţin favorabil adâncimea arborelui devine de ordinul lui *n*. De asemenea sunt utilizate forme particulare de arbore, cum ar fi ***B-arborii***, pentru care este garantat, că timpul de căutare rămâne de ordinul lui $log_2 n$.

Atunci când un fişier cu acces direct trebuie să poată fi utilizat în egală măsură şi în acces secvenţial, fişierul poate fi organizat conform unei scheme mixte, numită **secvenţial indexată**, care conţine un tabel al indicilor la diferite nivele la care există adrese fizice directe (v.6.3.2).

6.2.3.2. Chei multiple

Este cazul când pot fi utilizate mai multe chei pentru a desemna o înregistrare. În caz general, pot exista mai multe înregistrări pentru care o cheie particulară are o valoare dată. O cheie, valoarea căreia determină înregistrarea în mod univoc, se numeşte cheie primară; acest termen este aplicat, prin extensie, şi pentru o combinaţie de chei.

Tehnica de bază folosită pentru manipularea unui fişier cu chei multiple este organizarea multilistă. Este utilizat câte un indice distinct pentru fiecare cheie. Fiecare intrare a tabelului indicilor, asociată unei chei concrete, care corespunde unei valori distincte a acestei chei, punctează topul unei liste în care sunt toate înregistrările pentru care cheia considerată posedă valoarea dată. Pentru realizarea acestor liste fiecare înregistrare trebuie să conţină tot atâţia pointeri câte chei ale referinţelor distincte există. Figura 6.5 ilustrează această organizare pentru exemplul unui fişier document.

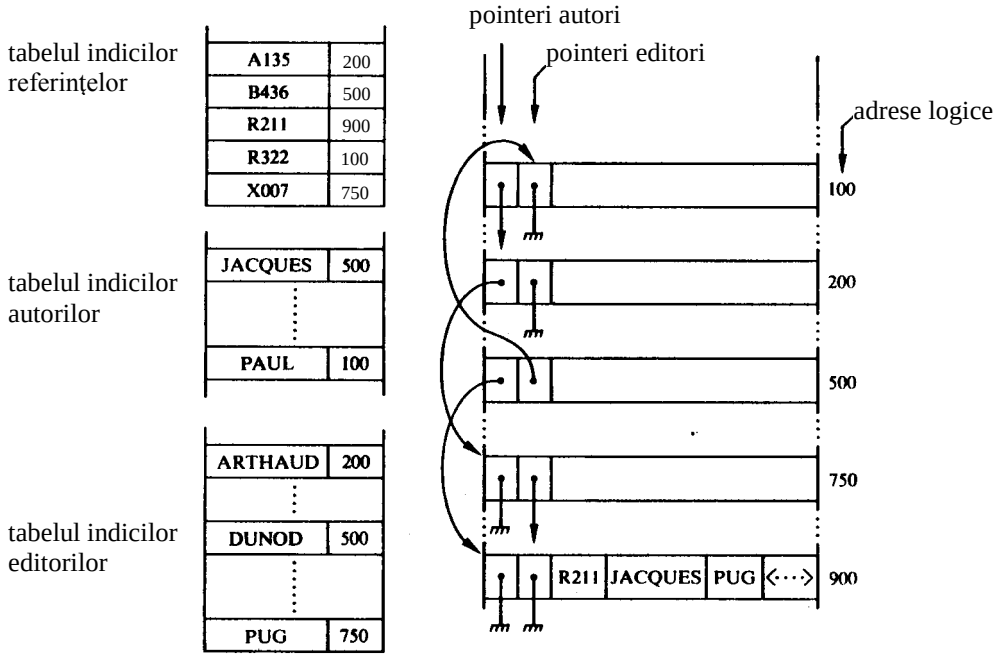


Fig.6.5. Organizare multilistă pentru un fişier cu chei multiple

În afara tabelului indicilor referinţelor au mai fost introduse un tabel al autorilor şi unul al editorilor. Fiecare intrare a tabelului indicilor autorilor este topul unei liste, care grupează toate înregistrările pentru care valoarea câmpului *autor* este aceeaşi. Pointerii acestor liste sunt adrese logice. Este simplu de găsit toate lucrările unui autor publicate la una şi aceeaşi editură. Pot fi obţinute combinaţii pentru aceste cereri cu ajutorul operaţiei *intersecţie*.

Este posibilă comprimarea reprezentării fişierului, utilizând liste circulare, care includ intrările corespunzătoare ale tabelelor indicilor (lista lucrărilor scrise de Paul, de exemplu, se închide la intrarea *Paul* din tabelul indicilor autorilor). Câmpurile *autor* şi *editor* ale părţii *info* a reprezentării înregistrărilor pot fi suprimate pentru că ele pot fi determinate din listele respective. Dezvoltând această idee, se poate crea câte un indice distinct pentru fiecare câmp; reprezentarea înregistrărilor va conţine doar pointeri şi toată informaţia se conţine doar în tabelele indicilor. Un fişier reprezentat în acest mod se numeşte **inversat**. Această reprezentare permite un răspuns imediat la cereri legate de combinaţii ale câmpurilor şi poate fi combinată cu organizarea directă, utilizând o cheie primară. Tabelele indicilor secundari conţin în acest caz listele valorilor cheii primare. Pentru exemplul unui fişier documentar organizarea descrisă mai sus este schematic prezentată în figura 6.6.

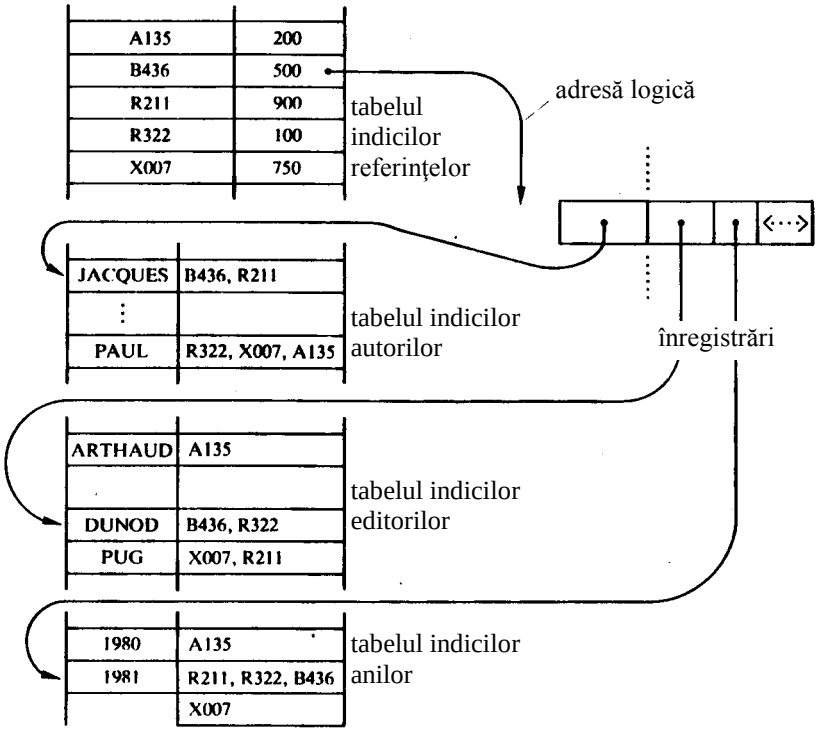


Fig.6.6. Organizarea unui fişier inversat

Terminăm studierea funcţiilor de acces logic subliniind, că funcţiile de acces complex sunt construite pornind de la funcţii elementare. De exemplu, fişierele sistemului Unix sunt fişiere de caractere pentru care sunt puse la dispoziţie un set de primitive de acces direct. Orice funcţie de acces mai complicat trebuie programată, utilizând aceste primitive.

6.3. Organizarea fizică a fişierelor

În modelul ierarhic, prezentat mai sus, problema organizării fizice a fişierelor poate fi formulată astfel: să se elaboreze o implantare în memoria secundară a unei mulţimi de fişiere respectând următoarele ipoteze:

- orice fişier este definit logic ca un segment, adică ca o mulţime de informaţii, care ocupă amplasamente adiacente, reperate prin adrese logice sau deplasări,
- dacă nu există concretizări particulare (bandă magnetică, de exemplu), memoria secundară este de tip disc, organizată ca o mulţime de blocuri de lungime fixă. Modul de adresare a blocurilor este precizat în 6.3.3; este suficient să se cunoască că blocurile sunt desemnate de adrese fizice ordonate, adrese consecutive desemnând blocuri adiacente.

Notăm aici, că dacă sistemul de operare realizează o memorie virtuală segmentată, implantarea fizică a segmentelor şi realizarea funcţiilor de acces elementar (citire, scriere, execuţie) sunt puse în şarja sistemului. Este posibilă în acest caz confundarea noţiunilor de segment şi fişier. Funcţiile SGF sunt reduse la gestiunea numelor şi organizarea logică, ceea ce este realizat, de exemplu, pentru sistemul Multics.

Vom trata cazul unei gestionări directe de către SGF a implantării fizice a fişierelor. Pot fi evidenţiate două clase de metode, dacă luăm în consideraţie respectarea sau nerespectarea contiguităţii adreselor logice.

6.3.1. Implantare secvenţială

În acest caz fiecare fişier ocupă o mulţime de blocuri consecutive în memoria secundară. Este unicul mod de implantare în cazul unei benzi magnetice: fişierele sunt aranjate consecutiv pe bandă, fiecare fiind separat de următorul cu ajutorul unui simbol, numit sfârşit de fişier (end of file, EOF). Un EOF dublu marchează sfârşitul părţii utilizate a benzii. Descriptorul unui fişier este plasat la începutul fişierului şi, adesea, repetat la sfârşitul fişierului. În interiorul fişierului înregistrările sunt aranjate consecutiv; dacă lungimea lor este variabilă, ea va fi prezentă la începutul fiecărei înregistrări. Caracterul EOF este detectat în mod automat de controlerul mecanismului de derulare a benzii. O operaţie "căutarea EOF-ului fişierului" permite saltul de la un fişier la altul pentru căutarea unui fişier cu numele dat.

Implantarea secvenţială poate fi în egală măsură folosită şi pentru discuri. Avantajul principal constă în garantarea unui acces secvenţial eficient (informaţiile cu adrese logice succesive sunt implantate în blocuri adiacente) permiţând în acelaşi timp un acces direct eficient (calcularea adresei fizice pornind de la adresa logică este foarte simplă şi nu cere

accesarea discului). Totuşi, această metodă prezintă inconveniente grave în cazul în care crearea, distrugerea sau modificarea lungimii unui fişier sunt operaţii frecvente:

- memoria secundară devine fragmentată, defragmentarea periodică este costisitoare;
- este complicată organizarea modificării lungimii unui fişier: un fişier poate fi de o lungime mai mare doar recopiindu-l integral într-o zonă de memorie mai mare.

Drept consecinţă, implantarea secvenţială pe disc este utilizată doar în cazurile în care dezavantajele susmenţionate pot fi diminuate:

- pentru fişierele numărul şi lungimea cărora nu variază (de exemplu, fişiere create odată pentru totdeauna şi utilizate mai apoi doar pentru consultare);
- sisteme primitive pentru microcalculatoare, când simplitatea realizării este un factor preponderent.

6.3.2. Implantare non contiguă

Dacă abandonăm restricţia contiguităţii implantării, memoria secundară se transformă într-o resursă banalizată, blocurile memoriei secundare fiind echivalente din punctul de vedere al alocării lor. Pentru început introducem informaţiile necesare pentru realizarea funcţiilor de acces; informaţiile pentru asigurarea securităţii sunt descrise în 6.5.

6.3.2.1. Blocuri înlănţuite

Blocurile fizice, care conţin amplasamente logice consecutive sunt înlănţuite între ele; pentru această înlănţuire trebuie să fie rezervat în fiecare bloc un pointer. Descriptorul conţine un pointer la primul şi ultimul bloc şi numărul blocurilor ocupate (fig.6.7).

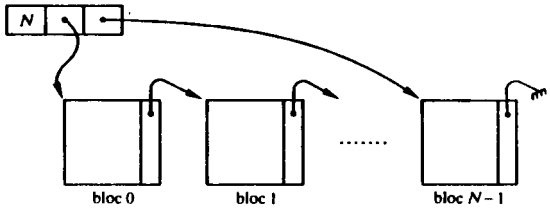


Fig.6.7. Alocare prin blocuri înlănţuite

Ultimul bloc, care poate fi utilizat parţial, trebuie să conţină indicaţii despre numărul de amplasamente ocupate. Deci, este necesar un amplasament în fiecare bloc pentru această informaţie sau cel puţin un bit (indicator) pentru marcarea ultimul bloc. Deoarece există un pointer la ultimul bloc este simplu să extindem un fişier, adăugând informaţii la sfârşit.

Acest mod de alocare este bine adaptat accesului secvenţial. Ca rezultat, putem accesa un bloc doar respectând înlănţuirea; accesul direct este costisitor, deoarece fiecare citire a unui pointer necesită o accesare a discului. La fel şi extinderea altfel, decât la sfârşit, este foarte dificilă: trebuie să permitem existenţa blocurilor parţial pline, să prevedem o dublă înlănţuire, etc. Utilizarea alocării înlănţuite este limitată de cazul sistemelor mici, în special pentru organizarea fişierelor pe dischete.

6.3.2.2. Tabele de implantare

În cazul accesului direct timpul de acces la un bloc trebuie să nu depindă de adresa sa, ceea ce poate fi obţinut punând toţi pointerii într-un tabel unic de implantare. Descriem mai multe variante ale acestei metode, care diferă prin modul de organizare a tabelului. Problema principală este garantarea uniformităţii timpilor de acces pentru tabelele de lungime mare şi permiterea inserării şi distrugerii blocurilor în orice punct al fişierului.

1) Tabel unic

Figura 6.8 (a) descrie o organizare cu tabel unic. Lungimea fişierului este limitată de numărul blocurilor pe care descriptorul le defineşte în tabel (el însuşi conţinându-se într-un număr întreg de blocuri).

2) Tabel înlănţuit

Conform organizării din fig.6.8 (b) tabela de implantare constă dintr-o suită de blocuri înlănţuite. Putem depăşi în acest fel limitarea dimensiunii unui fişier, devine posibilă inserarea blocurilor în mijlocul fişierului, cu condiţia rezervării unor amplasamente libere în tabel. Reorganizarea generată de inserare este legată de un bloc al tabelului, iar dimensiunea tabelului de implantare este limitată de costul căutării în acest tabel. Pentru tabele mari organizarea descrisă este mai eficace.

3) Tabele cu mai multe nivele

Figura 6.9 descrie o organizare în care tabelul de implantare a fişierului este organizat în mod arborescent, pe niveluri (de obicei, două sau trei niveluri).

Organizarea dată permite un acces direct rapid la un bloc cu adresa logică dată; ea permite inserările cu unele măsuri de precauţie.

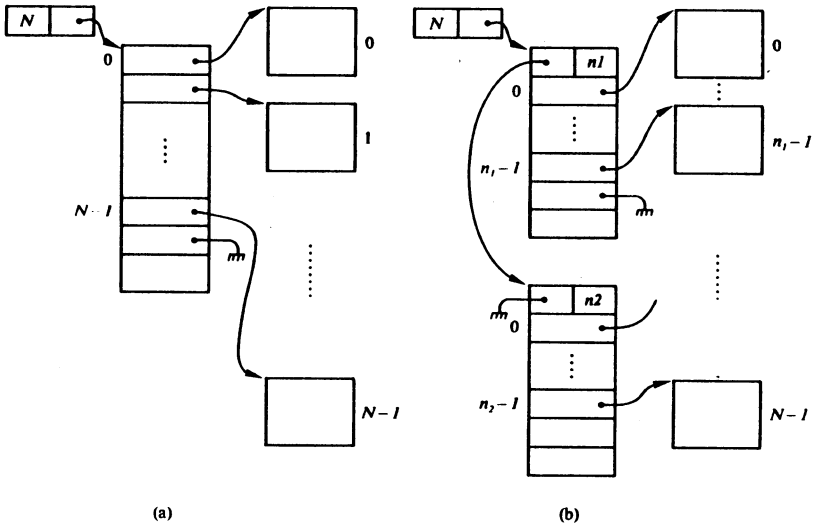


Fig.6.8. Tabel de implantare cu un nivel

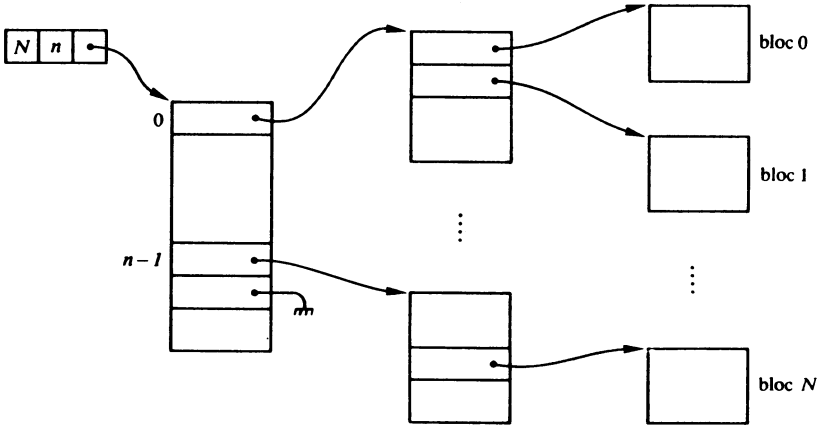


Fig.6.9. Tabel de implantare cu două nivele

4) Aplicaţii: organizare secvenţială indexată

Cu titlu de exemplu vom descrie o organizare frecvent utilizată, cea a fişierelor secvenţiale indexate. Este vorba de o schemă în care organizarea logică şi cea fizică nu sunt separate net: tabelul indicilor şi tabelul implantării sunt grupate într-o structură de date unică.

Această organizare se aplică la fişiere pentru care se cere utilizarea simultană a accesului direct (cu ajutorul cheilor) şi a accesului secvenţial (specificat de o ordonare a cheilor). Pentru simplitate, vom descrie o organizare în care cheia este unică.

Principiul constă în organizarea tabelului indicilor (ordonaţi conform valorii cheii) ca un tabel de implantare cu mai multe niveluri în care figurează nu adrese logice, ci adrese fizice. Printre altele, cel puţin pentru implantarea iniţială se încearcă să se plaseze informaţii logic contigue în blocuri vecine pentru a permite un acces secvenţial eficient.

Figura 6.10 ilustrează principiul organizării secvenţiale indexate pentru cazul, descris în exemplul 2 din 6.2.1. Cheia utilizată este numărul referinţei. Este folosită o organizare pe două niveluri. Intrarea *i* a tabelului primar al indicilor (cheia *cp[i]*) punctează la un tabel secundar al indicilor, care reperează înregistrările pentru care cheia *c* verifică relaţia

$$cp[i\text{-}1] < c \leq cp[i]$$

Mai mult, fiecare intrare a tabelului secundar punctează o zonă, unde sunt plasate secvenţial înregistrările cu aceeaşi organizare a cheilor. Această ultimă zonă este de lungime suficient de mică pentru a putea fi citită întreagă în memorie atunci când este accesat un articol pe care ea îl conţine. Tabelul primar fiind păstrat în memoria centrală atunci când fişierul este deschis, consultarea unei înregistrări cere două accesări a discului pentru prima citire a unei zone şi nici una pentru consultarea înregistrărilor succesive din zonă. Accesul secvenţial este astfel eficient, însă accesul direct rămâne posibil graţie tabelului indicilor.

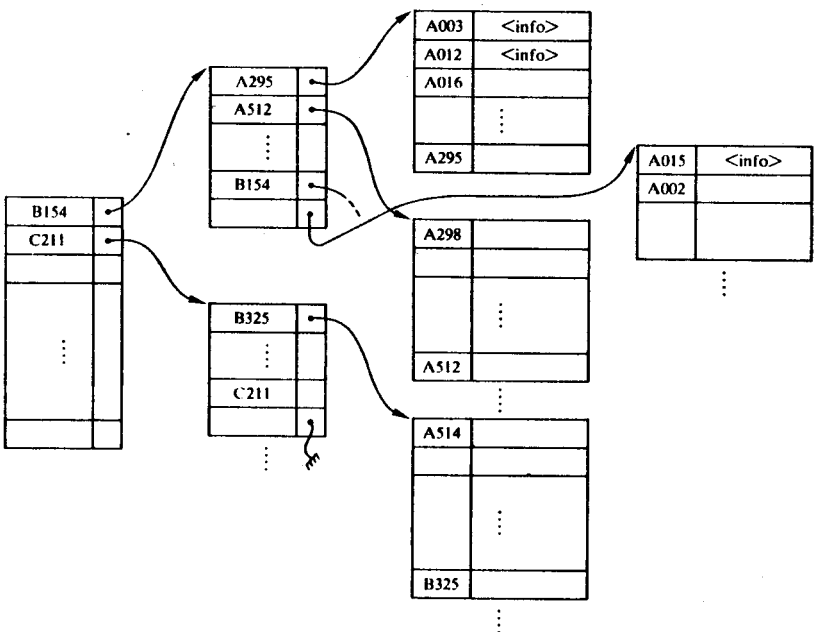


Fig.6.10. Fişier secvenţial indexat

Problema principală a organizării secvenţiale indexate vine de la inserarea şi suprimarea înregistrărilor. Chiar de la crearea iniţială a fişierului, pot fi lăsate amplasamente libere în fiecare zonă, care este apoi modificat prin compresie, atunci când au loc modificări. Dacă toate amplasamentele rezervate sunt ocupate, sunt utilizate zone de depăşire: fiecărui tabel secundar i se asociază o zonă de depăşire, înregistrările fiind ordonate în ordinea inserării. Căutarea unui articol are o durată mai mare, iar atunci când zonele de depăşire sunt foarte pline este preferabil să fie reorganizat fişierul în întregime, alocând zone noi şi creând, la necesitate, tabele secundare noi.

6.3.3. Alocarea memoriei secundare

Problema alocării memoriei secundare poate fi pusă în termeni similari cu cea a memoriei principale; restricţiile principale sunt:

- alocarea prin blocuri de lungime fixă,
- costul ridicat al accesului,
- caracteristicile proprii fiecărui suport: organizarea pe piste, dispozitive de citire-scriere mobile sau fixe.

Interfaţa alocatorului memoriei secundare conţine două primitive:

cerere_zonă(n,a)	date	:	*n* (întreg), numărul de blocuri contigue cerute
	rezultat	:	*a* (adresă), adresa primului bloc
	excepţie	:	număr de blocuri insuficiente
eliberare_zonă(n,a)	date	:	*n* (întreg), numărul de blocuri contigue eliberate
			a (adresă), adresa primului bloc
	excepţie	:	număr de blocuri eliberate

Cazul *n=1* este întâlnit frecvent şi poate justifica folosirea primitivelor particulare (*cerere_zonă, eliberare_zonă*).

Structura de date cel mai des utilizată pentru a descrie starea de ocupare a memoriei este un lanţ de biţi, bitul cu numărul *i* indicând starea (ocupat sau liber) a blocului cu acelaşi număr.

Acest tabel de ocupare poate atinge dimensiuni importante; de exemplu, sunt necesari 12,5 Ko pentru a descrie ocuparea unui disc de 200 Mo alocaţi prin blocuri de 2 Ko. Pentru a spori eficienţa algoritmilor de alocare tabelul poate fi organizat pe mai multe niveluri.

Exemplul 6.3. Discurile cu braţe mobile sunt organizate pe **cilindre** (mulţime de blocuri, accesibile pentru o poziţie dată a braţului mobil). Fiecare cilindru conţine tabelul de ocupare a blocurilor pe care intră în componenţa sa; mai mult, un tabel care duce evidenţa umplerii cilindrelor, încărcat în memoria centrală, indică numărul de blocuri libere în fiecare cilindru. Căutarea unei zone libere poate să se facă fără deplasarea braţului, dacă cilindrul curent conţine un număr suficient de blocuri libere; în caz contrar, dintre cilindrele posibile poate fi ales cilindrul, care este cel mai aproape de poziţia curentă a braţului. ◄

Din raţionamente de securitate (v. 6.5), starea *liber* sau *ocupat* a unui bloc este păstrată chiar în blocul propriu-zis.

6.4. Realizarea funcţiilor de acces elementar

6.4.1. Organizarea descriptorilor

Un descriptor de fişier trebuie să conţină informaţii de tipuri diverse:

- informaţii pentru localizarea fizică,
- informaţii referitor la utilizare,
- informaţii asociate protecţiei şi securităţii.

6.4.1.1. Localizarea fizică

Informaţiile pentru localizarea fizică a unui fişier au fost descrise în 6.3. Dacă aceste informaţii sunt de dimensiune fixă şi redusă (implantare contiguă, blocuri înlănţuite) ele se conţin direct în descriptor; în caz contrar (tabele de implantare înlănţuite sau pe mai multe niveluri), descriptorul conţine doar un pointer la tabelul de implantare şi nişte informaţii sintetice, cum ar fi dimensiunea fişierului, dimensiunea tabelului de implantare, etc.

6.4.1.2. Informaţii de utilizare

Informaţiile, legate de utilizarea unui fişier pot fi clasificate după cum urmează:

1) *Informaţii de stare.* Aceste informaţii definesc starea curentă a fişierului: deschis sau închis, gradul de partajare (numărul de utilizatori, care au deschis simultan fişierul), disponibilitatea pentru modificare, etc.

2) *Informaţii despre conţinutul fişierului*. Aceste informaţii permit interpretarea conţinutului fişierului. Este posibil să se asocieze unui fişier un tip, care specifică operaţiile permise. Sunt utilizate, de exemplu, tipurile "text ASCII", "binar translatabil", "binar absolut", "catalog", etc. Chiar la deschiderea unui fişier putem verifica, dacă utilizarea pretinsă este compatibilă cu tipul fişierului.

3) *Informaţii despre structura logică.* Aici pot fi plasate informaţiile care permit trecere de la structura logică la structura fizică a fişierului. De exemplu, în cazul înregistrărilor de lungime fixă este convenabil să se aleagă lungimea înregistrării multiplu sau submultiplu al lungimii blocului fizic. Raportul lungimea blocului/lungimea înregistrării, adesea numit *factor de bloc*, figurează în cadrul descriptorului.

4) *Informaţii despre utilizările precedente.* Aceste informaţii pot fi de natură statistică (numărul deschiderilor, accesărilor, intervalul mediu timp de între accesări, gradul mediu de partajare, etc.) şi au scopul de a înţelege mai bine funcţionarea sistemului pentru ameliorarea lui. De exemplu, informaţiile despre utilizare pot ajuta la o gestiune automată a ierarhiei memoriei secundare: fişierele neutilizate o perioadă lungă pot fi transferate într-o memorie de arhivare pentru a elibera loc în memoriile cu acces rapid. Aceste informaţii mai pot fi utilizate pentru diferite funcţii de securitate: înregistrarea datei ultimului acces sau a ultimei modificări permite un control al utilizării fişierului.

Precizăm realizarea funcţiilor elementare de acces (primitive), definite în 6.1.

Ca şi majoritatea funcţiilor unui sistem de operare, aceste primitive pot fi apelate în două moduri: cu ajutorul unor instrucţiuni sau prin apelarea regimului supervizor. La nivelul limbajului de comandă fişierele sunt desemnate prin numele lor simbolic, care este interpretat în contextul curent. Dacă o primitivă este apelată printr-un apel al supervizorului, în timpul execuţiei unui program, fişierele sunt, de obicei, desemnate de numele local, care este cel al unui descriptor valid în interiorul programului curent. Corespondenţa între numele simbolice şi numele locale este realizată prin instrucţiuni de asociere. Interpretarea numelor fişierelor este comună tuturor primitivelor.

Pentru fiecare primitivă vom prezenta:

- specificările interfeţei,
- algoritmul care o realizează,
- situaţiile anormale posibile.

6.4.2. Crearea şi distrugerea

6.4.2.1. Crearea

1) Specificarea interfeţei

Parametrii furnizaţi sunt:

numele fişierului	
tipul	(opţional)
lungimea	(opţional)
starea iniţială	(opţional)

Cel mai frecvent, parametrii *starea iniţială* şi *lungimea* nu sunt furnizaţi, având valori implicite: lungimea = *0,* fişierul este închis, lista de acces predefinită.

2) *Operaţii*

- să se creeze un descriptor pentru fişier, obţinem astfel un nume intern,
- dacă numele furnizat este un nume extern, să se creeze o intrare a acestui nume în catalogul curent,
- să se aloce memorie fişierului; chiar dacă lungimea sa este nulă, un fişier conţine un antet, care ocupă primul său bloc,
- să se iniţializeze descriptorul cu informaţiile de localizare, lungime, protecţie (lista drepturilor iniţiale de acces),
- dacă descriptorul nu se conţine direct în catalog, să se introducă o intrare în catalog cu numele intern al fişierului (adresa descriptorului).

3) *Cazuri anormale*

- nume furnizat incorect (incorect construit sau care desemnează un fişier deja existent);
- memorie insuficientă în memoria secundară.

Fişierul nu poate fi creat şi situaţia anormală este semnalată.

6.4.2.2. Distrugerea

1) *Specificarea interfeţei*

Unicul parametru furnizat este numele fişierului. Efectul operaţiei este de a suprima orice acces ulterior la fişier, invalidând toate numele acestuia şi eliberând toate resursele pe care fişierul le utilizează (intrări în tabele, memorie).

2) *Operaţii*

- să se elibereze toată memoria, care a fost alocată fişierului;
- să se elibereze memoria alocată descriptorului şi numele intern să devină reutilizabil (sau invalidat definitiv);
- suprimarea numelui extern al fişierului în catalogul, care-l include, de asemenea, dacă este cazul, în tabelul numelor locale.

Această ultimă operaţie poate fi destul de delicată, dacă există mai multe căi de acces la fişier. Poate fi utilizată una din următoarele soluţii:

- dacă căile multiple de acces sunt legături, nu se va lua nici o măsură; un acces ulterior prin una din aceste legături va conduce la numele extern. Inexistenţa fişierului va fi detectată (cu excepţia cazului în care un fişier cu acelaşi nume a fost creat în intervalul de timp dat);
- dacă căile multiple de acces sunt nume (ceea ce se poate produce, dacă structura catalogului nu este o arborescenţă) vor fi utilizate sau legături inverse (metodă costisitoare), sau un contor al referinţelor păstrat în descriptor, care duce evidenţa numărului căilor de acces. Un fişier poate fi distrus doar dacă există o singură cale de acces. Această soluţie este utilizată, de exemplu, în sistemul Unix.

6.4.3. Deschiderea şi închiderea

6.4.3.1. Deschiderea

1) *Specificarea interfeţei*

Parametrii operaţiei de deschidere sunt:

- numele fişierului
- condiţiile de utilizare:
  - modul de acces (citire, scriere, execuţie, etc.)
  - procedurile de acces (secvenţial, direct, sincron, etc.)
  - parametrii de transfer (tampoane, etc.)

Operaţia de deschidere are scopul de a pune un fişier în starea în care accesul este posibil conform modului specificat. Această operaţiei este de dublă utilitate pentru SGF:

- Protecţie. SGF poate să controleze:
  - la deschidere, dacă utilizatorul este autorizat să acceseze fişierul în condiţiile specificate,
  - la fiecare accesare a fişierului, dacă condiţiile de acces sunt compatibile cu cele specificate la deschidere,
  - în caz de partajare, dacă condiţiile de accesare ale utilizatorilor sunt reciproc compatibile.
- Eficienţă. SGF poate să accelereze accesul la fişierele deschise, aducând în memorie descriptorii lor, tabelele de implantare şi textele procedurilor de acces. Fişierul însuşi poate fi transferat, eventual, pe un suport cu acces mai rapid; dacă el se află pe un suport amovibil nemontat, montarea este necesară.

Numele furnizat ca parametru poate fi un nume extern sau local. Pentru durata deschiderii unui fişier SGF trebuie să poată accesa rapid toate informaţiile necesare utilizării: descriptorul şi condiţiile de utilizare valabile pentru perioada deschiderii. Aceste informaţii sunt apoi plasate într-un descriptor local, durata de existenţă a căruia este durata aflării fişierului în starea *deschis*; numele local este numele acestui descriptor local.

2) Operaţii

La deschidere vor fi realizate următoarele operaţii:

- localizarea fişierului şi, eventual, transferarea lui pe un suport mai rapid; cerere eventuală de montare a acestuia,
- controlarea dreptului utilizatorului de a deschide fişierul în modul specificat (consultând lista de acces),
- crearea unui descriptor local (dacă fişierul nu posedă deja unul în mediul curent) şi atribuirea unui nume local,
- alocarea eventuală a memoriei pentru zonele tampon de intrare-ieşire; în unele cazuri programul de intrare-ieşire al canalului poate fi construit în dependenţă de condiţiile specificate pentru transfer.

3) Cazuri anormale

- condiţii de deschidere incompatibile cu drepturile utilizatorului,
- imposibilitatea deschiderii, datorată restricţiilor specificate pentru partajare (de exemplu, restricţii de tipul cititor-redactor),
- lipsa spaţiului în memorie pentru descriptorul local sau zonele tampon.

6.4.3.2. Închiderea

1) Specificarea interfeţei

Unicul parametru necesar este numele fişierului. Efectul operaţiei este de a interzice orice acces ulterior la fişier şi de a-l transfera într-o stare coerentă şi stabilă, redându-i definitiv toate modificările operate în perioada cât a fost deschis.

2) Operaţii

Descriptorul fişierului este actualizat (dacă aceasta nu a fost făcut anterior) pentru a înregistra definitiv modificările operate în perioada cât a fost deschis; descriptorul local este suprimat, numele local putând fi reutilizat. Memoria ocupată de zonele tampon şi procedurile de intrare-ieşire este eliberată, dacă ea a fost alocată doar pentru perioada de deschidere a fişierului. Fişierul este, eventual, transferat pe suportul său de origine; o cerere de demontare este trimisă dacă suportul de origine este amovibil.

Unele sisteme operează cu fişiere temporare, adică fişiere desemnate doar de numele local. La închiderea unui astfel de fişier, utilizatorul are de ales între două posibilităţi:

- să catalogheze fişierul, atribuindu-i un nume extern; fişierul va fi în acest caz salvat.
- (implicit) să nu facă nimic, fişierul fiind distrus în acest caz.

3) Cazuri anormale

Pentru garantarea coerenţei fişierului, în timpul închiderii nu trebuie să aibă loc operaţii de transfer (a descriptorului sau a fişierului).

6.4.4. Acces elementar la informaţii

Parametrii primitivelor de transfer permit specificarea:

- "metodei de acces" utilizate, care determină modul de desemnare a înregistrărilor - obiectul transferului (acces secvenţial, direct, cu chei, etc.),
- sincronizarea transferului (transfer sincron sau asincron); în cazul unui transfer asincron, modul de deblocare a unui proces apelant,
- zonele tampon utilizate (specificate de către utilizator sau furnizate de SGF).

Accesul elementar (citire sau scriere) la o înregistrare a fişierului constă din două etape:

- determinarea adresei fizice a înregistrării plecând de la desemnarea sa logică; pentru aceasta sunt utilizate metodele descrise în 6.2,
- executarea transferului fizic; va fi folosită adresa obţinută mai sus şi primitivele de acces fizic la disc (v.cap.4).

6.5. Securitatea şi protecţia fişierelor

6.5.1. Despre securitate şi protecţie

Fişierele conţin toată informaţia administrată de sistemul de operare, inclusiv informaţia necesară gestiunii sistemului. Este important să se garanteze integritatea acestor informaţii în eventualitatea unor pene fizice, logice sau în caz de rea voinţă. Obiectivele şi metodele pot fi considerate aparţinând la două direcţii: **securitate** şi **protecţie**.

Cu titlul **securitate** vom îngloba mulţimea metodelor, care sunt chemate să garanteze execuţia tuturor operaţiilor asupra unui fişier în conformitate cu specificaţiile lor, chiar în cazul unor defecte în cursul execuţiei, şi că informaţiile conţinute într-un fişier nu sunt alterate, dacă nici o operaţie nu a fost executată. Metodele utilizate sunt, în caz general, cele care asigură toleranţa sistemului la defecte; baza lor este redundanţa informaţiilor.

Prin **protecţie** vom înţelege mulţimea metodelor chemate să specifice regulile de utilizare şi să garanteze respectarea lor. O descriere succintă a metodelor principale de protecţie a fost adusă în 5.1.4. Ele se bazează pe noţiunea de drept de acces şi existenţa unui mecanism, care permite să se garanteze conformitatea operaţiilor drepturilor de acces specificate.

6.5.2. Securitatea fişierelor

Securitatea fişierelor este asigurată, după cum a fost menţionat, prin redundanţa informaţiilor. Redundanţa poate fi introdusă în două moduri: salvarea periodică a unor informaţii, pentru a putea restabili o stare anterioară în caz de distrugere, şi redundanţa internă, care permite reconstituirea unor informaţii devenite incoerente din cauza unui accident de origine fizică sau logică, care a provocat o alterare parţială.

6.5.2.1. Redundanţa internă şi restabilirea informaţiilor

Principiul redundanţei interne constă în organizarea structurilor de date astfel, încât orice informaţie să poată fi obţinută prin cel puţin două căi distincte. Se reduce astfel probabilitatea că distrugerea accidentală a unei părţi a informaţiei va conduce la pierderea iremediabilă a conţinutului fişierului.

Tehnicile frecvent utilizate în acest scop sunt următoarele:

- pentru un fişier păstrat pe disc în blocuri înlănţuite se va folosi înlănţuirea dublă (succesor-predecesor),
- se va include în fiecare bloc un pointer la blocul, care conţine descriptorul, sau indicaţia că blocul este liber,
- se va include în descriptor numele simbolic al fişierului şi un pointer la catalogul în care se conţine fişierul,
- o parte a descriptorului va fi dublată (de exemplu, în primul şi ultimul bloc al fişierului).

Exemplul 6.4. Cu titlul de ilustrare a tehnicilor de redundanţă internă descriem principiul organizării fişierelor pe un calculator personal (Alto Xerox, organizare, care a fost preluată în sistemul Pilot al firmei Xerox).

Un fişier este organizat logic sub forma unei suite de înregistrări de lungime fixă (256 octeţi), numite pagini. Pagina fişierului este informaţia elementară accesibilă. Concepţia SGF se bazează pe următoarele principii:

1. Informaţiile se împart în *absolute* şi *auxiliare*; informaţiile absolute sunt suficiente, în principiu, pentru definirea totală a SGF; informaţiile auxiliare servesc pentru accelerarea accesului,
2. Dacă o informaţie auxiliară este utilizată pentru un acces, validitatea accesului este întotdeauna verificată utilizând o informaţie absolută,
3. Informaţiile auxiliare pot fi reconstituite pornind de la cele absolute.

Sunt disponibile următoarele funcţii:

- crearea unui fişier (iniţial vid),
- adăugarea unei pagini la sfârşitul unui fişier,
- distrugerea ultimei pagini a unui fişier,
- citirea sau scrierea unei pagini oarecare a unui fişier,
- distrugerea fişierului.

Un fişier este desemnat cu ajutorul unui nume intern *FV*, care este concatenarea unui nume unic *F* şi unui număr de versiune *V*. O pagină cu numărul *N* a unui fişier *FV* este desemnată printr-un nume logic *(FV, N)*. Orice acces la o pagină a fişierului utilizează numele său logic.

Fiecare pagină ocupă un bloc pe disc. Un bloc cu adresa fizică *AD* pe disc, care conţine o pagină (*FV, N*) conţine următoarele informaţii:

- numărul discului şi adresa *AD* paginii pe acest disc,
- un antet compus după cum urmează:

F	nume unic	absolut
V	numărul versiunii	absolut
N	numărul paginii în fişier	absolut
AP	adresa pe disc a paginii *N-1*	auxiliar
AS	adresa pe disc a paginii *N+1*	auxiliar

- 256 octeţi care conţin "valoarea" paginii fişierului.

Desemnarea externă a fişierelor este realizată cu ajutorul cataloagelor, care ele singure sunt fişiere. Orice intrare a unui catalog, care corespunde unui fişier, este un triplet (*id, FV, AD*) unde:

id	este identificatorul fişierului
FV	este numele intern al fişierului
AD	este adresa pe disc a paginii *0* a fişierului.

Un fişier poate aparţine mai multor cataloage; unul dintre acestea fiind identificat catalog proprietar.

Fiecare fişier conţine o pagină suplimentară, numită *pagină de gardă*, identificată de numărul *–1*. Această pagină conţine:

- identificatorul fişierului,
- numele catalogului proprietar şi adresa primei sale pagini.

Aceste informaţii sunt absolute, conţinutul catalogului, din contra, este considerat informaţii auxiliare.
Principiul redundanţei active este realizat după cum urmează.

a) Verificare la alocare şi la eliberare

Un bloc liber are antetul plin de zerouri. Această informaţie este absolută; tabelul de ocupare a discului (un bit per bloc, starea liber sau ocupat) are doar valoare auxiliară.
Antetul unei pagini este modificat în două situaţii:

- la prima scriere după alocare; se verifică dacă blocul este liber,
- la eliberarea blocului; se verifică dacă *FV* este al paginii eliberate, umplând antetul cu zerouri.

b) Verificarea accesului

Numele absolut al unei pagini de fişier este (*FV, N*); este numele utilizat pentru specificarea unui acces. De fapt, aceasta este o adresă, informaţie auxiliară, care este utilizată pentru operaţia fizică de acces. La fiece acces fizic la o pagină antetul acesteia este consultat pentru a verifica dacă numele absolut (*FV, N*), care-i prezent aici este chiar cel al paginii specificate. Modul de funcţionare a controlerului discului permite executarea acestei operaţii "din zbor" fără a pierde o turaţie a discului.

c) Restabilirea

Operaţia de restabilirea permite reconstituirea informaţiilor auxiliare pornind de la informaţiile absolute, în mod special:

- se verifică, dacă toate legăturile sunt valide şi sunt restabilite care au putut fi distruse,
- sunt restabilite cataloagele,
- este restabilit tabelul de ocupare a discului,
- sunt semnalate toate incoerenţele, care nu au putut fi reparate.

Restabilirea este realizată la lansarea sistemului, dar mai poate fi cerută, dacă au apărut unele incoerenţe în timpul lucrului. Algoritmul restabilirii poate fi de forma:

1) Va fi parcurs tot discul, bloc cu bloc; pentru fiecare bloc de număr *i* procedându-se după cum urmează:
   - dacă antetul blocului conţine doar zerouri, se va marca blocul *i* liber în tabelul de ocupare,
   - în caz contrar, să se introducă (*FV, N, i*) într-un tabel de blocuri, triat conform *FV, N;* să se verifice legăturile şi să fie actualizate, dacă este cazul.
2) Vor fi parcurse toate cataloagele (cataloagele au numerele *F* rezervate); se va verifica dacă fiecare catalog conţine un pointer la pagina *0* a unui fişier; dacă rămân fişiere necatalogate, se va crea o intrare pentru fiecare în catalogul proprietarului său; dacă acest catalog nu există, fişierul va fi clasat într-un catalog al fişierelor eronate.
3) Vor fi inventariate fişierele, care nu au putut fi restabilite. ◄

6.5.2.2. Salvare periodică

Se poate apriori conveni să fie salvate, la intervale regulate de timp, toate informaţiile conţinute într-un SGF. Admiţând o salvare pe discul dur la un debit de 20 Mo/s, durata acestei operaţii este de aproximativ 50 s pentru 1 Go. Poate fi realizată salvarea doar a modificărilor, care au avut loc pentru ultima perioadă, concomitent cu lucrul normal al sistemului.

Informaţiile obţinute în acest mod pot fi utilizate în două moduri:

- la cererea explicită a unui utilizator, pentru a restabili o stare anterioară a unui fişier specificat,
- după o pană, conform procedurii de mai jos.

În caz de pană, se va încerca mai întâi să se restabilească informaţiile SGF utilizând redundanţa internă, cum a fost descris mai sus. La terminarea acestei operaţii vom avea la dispoziţie o listă a fişierelor bănuite a fi alterate. După aceasta vor fi restabilite fişierele presupuse a fi alterate, folosind datele cele mai recent salvate.

6.5.3. Protecţia fişierelor

Protecţia fişierelor foloseşte principiul listelor de acces. O listă de acces, asociată unui fişier, conţine drepturile fiecărui utilizator asupra fişierului, adică mulţimea operaţiilor care îi sunt permise. În practică, diferite convenţii permit reducerea listei. Este definită mai întâi mulţimea drepturilor atribuite implicit fiecărui utilizator (mulţime, care poate fi vidă); lista de acces va conţine doar utilizatorii, drepturile cărora diferă de cele implicite. Convenţii de grupare permit asocierea unei mulţimi de drepturi unui grup de utilizatori (membrii unui proiect, etc.), ceea ce reduce şi mai mult lista. Lista de acces serveşte la verificarea validităţii oricărei operaţii asupra fişierului.

6.5.4. Autentificarea în Windows NT

Sistemul de operare Windows NT posedă mai multe instrumente de control a drepturilor de acces al utilizatorilor la resursele sistemului. Aceste instrumente ar fi absolut inutile în lipsa posibilităţii de a controla dacă la calculator se află anume acel utilizator, care corespunde datelor de evidenţă. Mecanismul, cu ajutorul căruia un utilizator îşi poate confirma autenticitatea se numeşte *de autentificare*. Autentificarea se produce în baza unui element secret (autentificator), aflat doar în posesia utilizatorului definit de informaţia de evidenţă. De obicei, drept autentificator serveşte parola utilizatorului, însă acesta poate fi şi o cartelă specială (smart, badge) sau informaţii, legate de datele biometrice ale utilizatorului. Windows NT permite dezvoltatorilor externi extinderea mecanismelor de autentificare. SO Windows NT execută autentificarea la intrarea în sistem. Pentru majoritatea cazurilor aceasta este suficient, deoarece sistemul în mod automat personalizează accesul în baza numelui utilizatorului intrat în sistem. Mai mult, Microsoft nu recomandă introducerea unor mecanisme de autentificare proprii în aplicaţii, respectând principiul intrării unice (Unified Logon) – utilizatorul trebuie să introducă parola o singură dată la intrarea în sistem. Există mai multe motivaţii pentru aceasta:

1. Cereri suplimentare de introducere a numelui şi parolei pot fi enervante pentru unii utilizatori,

2. Programarea incorectă a mecanismului de autentificare poate diminua securitatea sistemului,
3. Windows NT permite extinderea mecanismului de autentificare. Programele, construite cu un mecanism propriu de autentificare pot fi inutile, dacă sistemul utilizează un alt mecanism, decât cel presupus la elaborarea aplicaţiei.

Dacă se presupune obligator controlul numelui utilizatorului şi a parolei, pot fi propuse două metode de autentificare:

1. cu ajutorul funcţiei LogonUser - este cea mai simplă metodă, dar cere existenţa unor priorităţi pentru cel care apelează funcţia dată (nu este cazul şi pentru Windows XP).
2. utilizând Security Support Provider Interface (SSPI) - o metodă mai complicată, dar nu cere priorităţi suplimentare. Mai mult, metoda este funcţională şi pentru autentificare la distanţă.

Vom mai cerceta şi metoda bazată pe funcţia NetUserChangePassword, recomandată pentru controlul parolelor. Unii autori nu agreează această metodă la prima vedere foarte simplă.

Fiecare metodă va fi ilustrată cu ajutorul unei funcţii de forma:

```
BOOL CheckPassword_Method(
   IN PCTSTR pszDomainName,  // numele domenului
   IN PCTSTR pszUserName,    // numele utilizatorului
   IN PCTSTR pszPassword,    // parola
   OUT PHANDLE phToken       // token-ul utilizatorului
   );
```

La intrarea funcţiei se va da numele informaţiei de evidenţă, parola şi numele domenului care conţine înregistrarea de evidenţă. Numele domenului poate fi NULL, în acest caz funcţia alege singură domenul. Dacă numele utilizatorului şi parola au fost indicate corect, funcţia returnează TRUE, altfel – FALSE. Codul erorii poate fi obţinut prin funcţia **GetLastError**. Funcţia mai întoarce aşa numitul *token* al utilizatorului (obiectul marker). Acesta poate fi util, dacă va fi necesar să se îndeplinească anumite operaţiuni în numele utilizatorului autentificat (cerere de ajutor în cazul în care parola a fost uitată poate fi un exemplu). În caz contrar parametrul **phToken** poate fi pus NULL.

6.5.4.1. Funcţia LogonUser

Funcţia LogonUser din Win32 API este destinată exclusiv pentru soluţionarea problemei autentificării şi este de forma:

```
BOOL LogonUser(
   PTSTR pszUsername,      // numele utilizatorului
   PTSTR pszDomain,        // numele domenului
   PTSTR pszPassword,      // parola
   DWORD dwLogonType,      // tipul accesării
   DWORD dwLogonProvider,  // provider-ul accesării
   PHANDLE phToken         // token-ul utilizatorului
   );
```

Destinaţia primilor trei parametri este evidentă. Unele versiuni timpurii ale SO Windows NT nu permiteau să fie indicată valoarea NULL pentru numele domenului. Acest caz trebuie tratat în mod separat, dacă se doreşte ca procedura autentificării să funcţioneze în toate versiunile sistemului. Parametrul dwLogonType indică modul de intrare în sistem. Principiul de bază, care permite diferenţierea intrărilor în sistem, este legat de priorităţile prezente în informaţiile de evidenţă şi setul de grupe introdus în token-ul final. De exemplu, dacă se încearcă intrarea în sistem în regim de tratare pe loturi, informaţia de evidenţă, desemnată de parametrul pszUsername, trebuie să posede prioritatea SE_BATCH_LOGON_NAME, iar în token-ul utilizatorului va fi adăugat grupul Batch (S-1-5-3). Suplimentar există o serie de diferenţe subtile, care pot fi găsite în documentaţia LogonUser. Pentru exemplul nostru vom folosi metoda de intrare LOGON32_LOGON_NETWORK, care este cea mai rapidă. În acest caz utilizatorii, controlaţi cu ajutorul funcţiei date, trebuie să posede dreptul de intrare în reţea (în mod implicit toţi utilizatorii au acest drept). Parametrul dwLogonProvider indică care provider va fi utilizat. Utilizând valoarea LOGON32_PROVIDER_DEFAULT, va fi folosit provider-ul implicit. Parametrul phToken punctează variabila în care, în caz de succes, va fi introdus token-ul utilizatorului.

Mai jos este prezentat un fragment de cod în care funcţia CheckPassword_LogonUser foloseşte LogonUser pentru controlul numelui şi parolei utilizatorului:

```
BOOL CheckPassword_LogonUser(
   IN PCTSTR pszDomainName,
   IN PCTSTR pszUserName,
   IN PCTSTR pszPassword,
   OUT PHANDLE phToken
   )
{
```

```
_ASSERTE(pszUserName != NULL);
_ASSERTE(pszPassword != NULL);

HANDLE hToken;

TCHAR szDomainName[DNLEN + 1];
TCHAR szUserName[UNLEN + 1];
TCHAR szPassword[PWLEN + 1];

if (pszDomainName == NULL)
{
   BYTE bSid[8 + 4 * SID_MAX_SUB_AUTHORITIES];
   ULONG cbSid = sizeof(bSid);
   ULONG cchDomainName = countof(szDomainName);
   SID_NAME_USE Use;

   if (!LookupAccountName(NULL, pszUserName, (PSID)bSid, &cbSid,
                  szDomainName, &cchDomainName, &Use))
      return FALSE;
}
else
   lstrcpyn(szDomainName, pszDomainName, countof(szDomainName));

lstrcpyn(szUserName, pszUserName, countof(szUserName));
lstrcpyn(szPassword, pszPassword, countof(szPassword));

if (!LogonUser(szUserName, szDomainName, szPassword,
          LOGON32_LOGON_NETWORK, LOGON32_PROVIDER_DEFAULT,
          &hToken))
   return FALSE;

if (phToken == NULL)
   _VERIFY(CloseHandle(hToken));
else
   *phToken = hToken;
return TRUE;
 }
```

Câteva comentarii sunt necesare. În primul rând, toate string-urile de intrare sunt copiate în masive speciale din stiva funcţiei. Aceasta este necesar din cauza că toţi parametrii funcţiei sunt declaraţi constante string, iar LogonUser primeşte indicatori la string-uri, care pot fi modificate. În unele condiţii LogonUser poate scrie în parametrii pe care îi primeşte ceea ce poate conduce la prohibitarea protecţiei memoriei. Doi: conform celor menţionate mai sus, unele versiuni timpurii ale sistemului de operare Windows NT nu permit valoarea NULL pentru numele domenului. Această situaţie va fi tratată în mod individual şi numele corespunzător pentru domen este depistat cu ajutorul funcţiei LookupAccountName.

Observăm, că utilizarea funcţiei LogonUser nu este complicată, dar în Windows NT şi Windows 2000 are o restricţie substanţială: utilizatorul, care apelează această funcţie trebuie să posede prioritatea SE_TCB_NAME. Este o prioritate foarte înaltă, chiar nici administratorii nu o posedă, semnificând controlul total asupra sistemului. TCB - Trusted Computing Base (Baza Calculelor Sigure, în care poţi avea încredere) este o componentă a sistemului de calcul, care asigură îndeplinirea politicii securităţii. Este un cod, executat în regim de nucleu şi un cod de regim user, executat în contextul informaţiilor de evidenţă, care are prioritatea TCB. Deşi nu este periculos să fie acordată această prioritate înregistrării de evidenţă, destinate execuţiei unui serviciu de sistem, Microsoft nu recomandă acest lucru, ci recomandă să fie executat un cod, care cere prezenţa priorităţii TCB în contextul înregistrării de evidenţă de sistem. Folosirea înregistrării de evidenţă de sistem are un avantaj: aceasta nu posedă parolă, în rezultat ea nu poate fi furată sau calculată. Dacă nu este respectată recomandarea Microsoft, securitatea sistemului este redusă la zero, controlul parolei devenind inutil.

Necesitatea prezenţei priorităţii TCB restricţionează substanţial domeniul de utilizare a metodei date. În Windows XP LogonUser nu mai cere prezenţa priorităţii TCB la apelant, dar numărul de copii Windows NT şi Windows 2000 este încă foarte mare.

6.5.4.2. Autentificare cu ajutorul Security Support Provider Interface

O altă posibilitate este cea bazată pe *Security Support Provider Interface* (SSPI), o variaţie a standardului Generic Security Service API (GSS-API), realizat de firma Microsoft. Ambele interfeţe sunt destinate autentificării şi protecţiei informaţiilor, transmise prin reţea. Ideea se bazează pe faptul, că toate protocoalele autentificării de reţea (fie NTLM, Kerberos sau SSL) pot fi reprezentate sub forma unui schimb ordonat de mesaje, în timpul căruia un participant identifică pe celălalt (sau părţile se autentifică reciproc). Evident, SSPI poate fi utilizat şi local, în acest caz transmiterea pachetelor prin reţea este înlocuită de transmiterea zonei de memorie tampon dintr-un loc al programului în altul.

SSPI este doar o interfaţă care permite acces standardizat la diferite pachete de securitate (security packages). În componenţa SO Windows NT 4 intră un singur pachet (NTLM), care poate fi utilizat pentru autentificarea utilizatorilor. Începând cu Windows 2000 la acest pachet au mai fost adăugate pachetele Kerberos şi Negotiate. Fragmentul de cod, care urmează controlează corectitudinea numelui şi parolei unui utilizator folosind interfaţa SSPI şi pachetul de securitate NTLM.

```
BOOL CheckPassword_SSPI(
   IN PCTSTR pszDomainName,
   IN PCTSTR pszUserName,
   IN PCTSTR pszPassword,
   OUT PHANDLE phToken
   )
{
   _ASSERTE(pszUserName != NULL);
   _ASSERTE(pszPassword != NULL);

   ULONG lRes;
   CSspiClient Client;
   CSspiServer Server;

   // iniţializarea obiectului client
   lRes = Client.Initialize(pszDomainName, pszUserName, pszPassword);
   if (lRes != ERROR_SUCCESS)
      return SetLastError(lRes), FALSE;

   // iniţializarea obiectului server
   lRes = Server.Initialize();
   if (lRes != ERROR_SUCCESS)
      return SetLastError(lRes), FALSE;

   PVOID pRequest = NULL, pResponse = NULL;
   ULONG cbRequest, cbResponse;

   // generarea cererii iniţiale de autentificare
   lRes = Client.Start(&pRequest, &cbRequest);
   if (lRes != ERROR_MORE_DATA)
      return SetLastError(lRes), FALSE;

   // ciclul principal de schimburi de mesaje
   for (;;)
   {
      // prelucrarea cererii client şi generarea răspunsului
      lRes = Server.Continue(pRequest, cbRequest,
                        &pResponse, &cbResponse);
      if (lRes != ERROR_MORE_DATA)
         break;

      Client.FreeBuffer(pRequest);
      pRequest = NULL;

      // prelucrarea cererii serverului şi crearea unei cereri noi
```

```
            lRes = Client.Continue(pResponse, cbResponse,
                        &pRequest, &cbRequest);
            if (lRes != ERROR_SUCCESS &&
               lRes != ERROR_MORE_DATA)
               break;

            Server.FreeBuffer(pResponse);
            pResponse = NULL;
         }

         if (pRequest != NULL)
            Client.FreeBuffer(pRequest);
         if (pResponse != NULL)
            Server.FreeBuffer(pResponse);

         if (lRes != ERROR_SUCCESS)
            return SetLastError(lRes), FALSE;

         // control înregistrare de evidenţă a oaspeţilor
         lRes = Server.CheckGuest();
         if (lRes != ERROR_SUCCESS)
            return SetLastError(lRes), FALSE;

         // creare token la necesitate
         if (phToken != NULL)
         {
            lRes = Server.GetToken(phToken, TOKEN_ALL_ACCESS);
            if (lRes != ERROR_SUCCESS)
               return SetLastError(lRes), FALSE;
         }

         return TRUE;
      }
```

În codul iniţial al funcţiei CheckPassword_SSPI nu există apelări directe a funcţiei SSPI – toate apelurile sunt incluse în superclasele CSspiClient şi CsspiServer. Aceasta permite pe de o parte evidenţierea schimbului de mesaje între client şi server, iar pe de altă parte – folosirea acestor clase în calitate de bază, dacă se doreşte realizarea autentificării de reţea în aplicaţie.

Lucrul începe cu crearea şi iniţializarea obiectelor clientului şi ale serverului, în procesul de iniţializare obiectului clientului fiindu-i transmise informaţii despre utilizator. Apoi este apelată metoda CSspiClient::Start, pentru a genera cererea iniţială de autentificare şi are loc trecerea în ciclul de mesaje. Aici funcţia apelează metoda CSspiServer::Continue şi CSspiClient::Continue. Aceste metode primesc la intrare mesajul celeilalte părţi şi returnează mesajul de răspuns. Ciclul de mesaje durează până în momentul în care serverul stabileşte, că autentificarea s-a petrecut cu succes sau invers. Pentru autentificarea NTLM în Windows NT 4.0 şi Windows 2000 este caracteristic următorul moment: dacă un client încearcă să intre în sistem cu un nume necunoscut nici pentru sistemul local, nici pentru domene, şi în sistemul local nu este blocată înregistrarea de evidenţă a oaspeţilor, atunci autentificarea se termină cu succes, iar clientul intră în sistem sub numele înregistrării locale de evidenţă a oaspeţilor. Aceasta s-a realizat din dorinţa compatibilităţii cu produsul mai vechi al firmei Microsoft NT Lan Manager (de unde şi vine numirea NTLM). De aceea, în caz de succes, este apelată metoda CSspiServer::CheckGuest, care controlează, dacă nu cumva succesul autentificării se datorează acestui moment. În sfârşit, dacă este necesar token-ul utilizatorului, funcţia apelează metoda CSspiServer::GetToken, responsabilă de obţinerea token-ului. Cercetarea claselor CSspiClient şi CsspiServer, care necesită o studierea mai amănunţită a interfeţei SSPI, este lăsată ca exerciţiu.

6.5.4.3. Funcţia NetUserChangePassword

Utilizarea acestei funcţii pentru autentificare are loc conform codului care urmează:

```
BOOL CheckPassword_ChangePwd(
   IN PCTSTR pszDomainName,
   IN PCTSTR pszUserName,
   IN PCTSTR pszPassword,
```

```
    OUT PHANDLE phToken
    )
{
    _ASSERTE(pszUserName != NULL);
    _ASSERTE(pszPassword != NULL);
    _ASSERTE(phToken == NULL);

    USES_CONVERSION;

    PCWSTR pszPasswordW = T2CW(pszPassword);

    ULONG lRes = NetUserChangePassword(T2CW(pszDomainName),
                T2CW(pszUserName), pszPasswordW, pszPasswordW);

    if (lRes == ERROR_INVALID_PASSWORD ||
        lRes == NERR_UserNotFound)
        return SetLastError(ERROR_LOGON_FAILURE), FALSE;

    return TRUE;
}
```

Există mai multe considerente, care nu recomand utilizarea acestei funcţii pentru autentificare. Principalul considerent este faptul, că funcţia NetUserChangePassword nu este destinată autentificării utilizatorilor.

6.6. SGF din sistemul de operare Unix

Descriem cu titlu de exemplu organizarea fişierelor în sistemul de operare Unix.

6.6.1. Caracteristici generale

Toate fişierelor manipulate de utilizator formează o bază de date de tip ierarhic, modelul matematic fiind un arbore în care vârfurile intermediare corespund cataloagelor, iar frunzele – cataloagelor vide sau fişierelor. Structura sistemului de gestiune a fişierelor este prezentată în fig. 6.11. În realitate fiecare disc logic conţine o structură ierarhică separată de cataloage şi fişiere. Pentru a obţine în mod dinamic arborele general este utilizată „conectarea” ierarhiilor separate la o rădăcină fixă a sistemului de fişiere. În mediul utilizatorilor sistemului de operare UNIX istoric noţiunea de „sistem de fişiere” semnifică în acelaşi timp ierarhia cataloagelor şi fişierelor şi partea nucleului sistemului de operare, care administrează cataloagele şi fişierele. Ar fi mai corect că numim ierarhia cataloagelor şi fişierelor arhivă a fişierelor, iar noţiunea de „sistem de fişiere” să fie utilizată doar în cel de-al doilea sens.

Utilizatorii sistemului de operare Unix cunosc trei tipuri de fişiere: fişiere ordinare, cataloage şi fişiere speciale. Fişierele speciale sunt de fapt unica cale de acces direct la periferice; utilizarea lor este prezentată mai târziu. Diferenţa dintre fişierele ordinare şi cataloage este că modificarea conţinutului cataloagelor este permisă doar prin intermediul unor comenzi speciale ale sistemului. Un fişier în Unix este o suită de caractere (8 biţi); alte structurări nu sunt furnizate de către SGF. Sistemul de gestiune a fişierelor nu face nici o ipoteză despre conţinutul fişierelor ordinare, interpretarea fiind pusă în şarja utilizatorului.

Desemnarea fişierelor se bazează pe schema arborescentă, descrisă în 5.2.2. Identificatorii sunt interpretaţi în mediul catalogului curent; simbolurile speciale . (un punct) şi .. (două puncte) desemnează respectiv, în orice mediu, altul decât catalogul rădăcină, catalogul curent şi tatăl său. Simbolul / desemnează catalogul rădăcină; orice identificator, care începe cu / este interpretat în mediul rădăcinii. Acelaşi simbol / serveşte ca separator pentru numele calificate.

Este posibilă crearea unor legături la un fişier. O legătură, definită pentru fişier, cablează un nume nou, care are acelaşi statut ca şi numele iniţial. Pentru a păstra organizarea arborescentă a cataloagelor nu se permite crearea unei legături la un catalog. Suprimarea ultimei legături la un fişier antrenează în mod automat distrugerea fişierului.

6.6.2. Organizarea datelor

6.6.2.1. Descriptorii

Într-un catalog, intrarea care corespunde unui fişier conţine identificatorul fişierului şi un nume intern, care este indicele descriptorului său într-un tabel general, asociat suportului. Descriptorul (“i-nod” în terminologia Unix) conţine:

- tipul fişierului (ordinar, catalog, special),
- numele şi grupul proprietarului fişierului (v.6.5.3),
- informaţiile de protecţie,
- informaţiile localizării fizice (v.6.3.2.2),

- lungimea fişierului, în caractere,
- datele despre creare, ultima modificare, ultimul acces.

Fiecare suport fizic are un tabel propriu al descriptorilor (6.6.2.4).

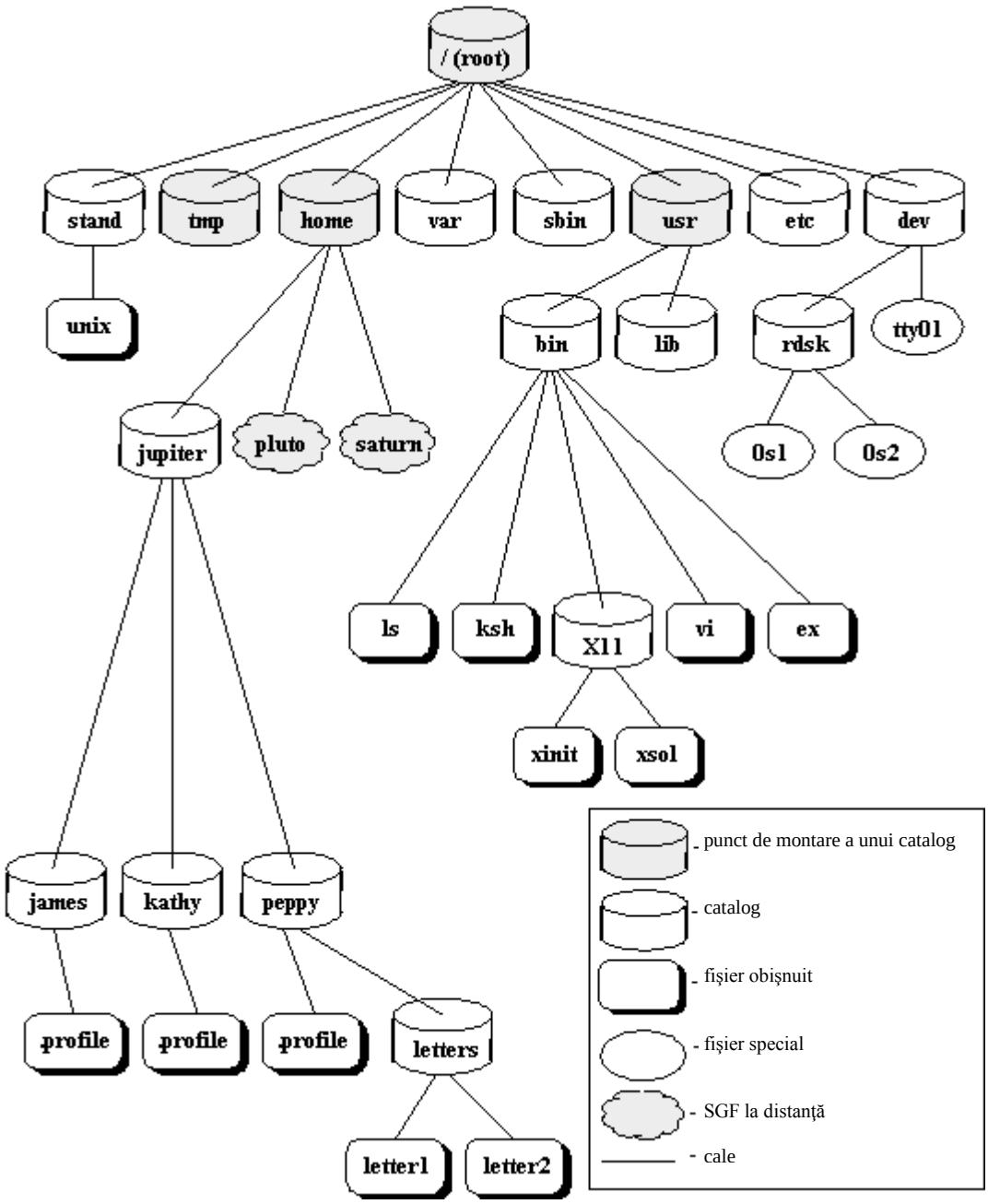


Fig.6.11. Structura SGF UNIX

6.6.2.2. Implantarea fizică

Memoria secundară este organizată în blocuri de lungime fixă (512 octeţi). În descriptorul unui fişier este rezervat un tabel *T* de 13 cuvinte pentru informaţiile despre localizarea fizică. În dependenţă de numărul *t* al blocurilor ocupate de fişier, sunt utilizate *1, 2* sau *3* niveluri de direcţionare (v.6.3.2.2):

- cuvintele de la *T[0]* la *T[9]* conţin adresele blocurilor *0 – 9* ale fişierului (ultimele fiind puse în *0*, dacă *t<10*),
- dacă *t>10*, cuvântul *T[10]* conţine adresa unui tabel de implantare de 128 de cuvinte, în care sunt adresele blocurilor de la *10* la *137*,
- dacă *t>138*, în cuvântul *T[11]* se află adresa unui tabel, care stabileşte adresele a *128* noi tabele de implantare; se poate ajunge astfel până la $10+128+128^2$ = 16522 blocuri,

- în sfârşit, dacă *t>16522*, pentru blocurile următoare este utilizată o direcţionare pe trei niveluri; adresa tabelului primar este în *T[12]*; se poate astfel atinge o lungime maximă de $16522+128^3 = 2\ 113\ 674$ blocuri.

Notăm, că timpul de accesare a informaţiilor unui fişier nu este uniform, începutul fişierului fiind privilegiat. Astfel, timpul de acces la primele 5120 caractere (10 blocuri) nu depinde de lungimea fişierului. Această proprietate poate fi exploatată, plasând în fişier informaţiile cel mai frecvent utilizate.

Dacă există mai multe suporturi de memorie secundară, sau volume, pentru fişiere, fiecărui suport i se va asocia un tabel distinct de descriptori. Tabelul se află la o adresă fixă pe volum. Numele intern al unui fişier este cuplul (adresă suport, indicile descriptorului în tabel).

6.6.2.3. Administrarea perifericelor

Perifericele sunt reprezentate de fişiere speciale, grupate într-un catalog unic (/dev). În locul informaţiilor de implantare, descriptorul unui fişier special conţine un pointer la descriptorul perifericului, care, la rândul său conţine doi indici, numiţi major şi minor. Indicele major identifică tipul perifericului şi punctează la programele primitivelor sale de acces, partajate de periferice de acelaşi tip. Indicele minor identifică individual fiecare periferic şi permite accesul la datele sale proprii, adică la tampoanele de intrare-ieşire utilizate.

6.6.2.4. Volume amovibile

Administrarea volumelor amovibile este realizată conform principiilor, descrise în 5.2.3.2. Catalogul unui volum amovibil este organizat exact ca şi catalogul general. Comanda unei montări logice (*montare*) permite integrarea catalogului unui volum în catalogul general, la un nivel oarecare şi cu un nume ales. De la acest moment, toate fişierele volumului devin accesibile. Unica restricţie este că nu-i posibilă crearea unor legături între suporturi diferite: un fişier dintr-un volum este accesibil doar traversând rădăcina catalogului acestui volum, iar această rădăcină nu punctează la tatăl său. Sunt evitate astfel orice căutări de legături la demontare.

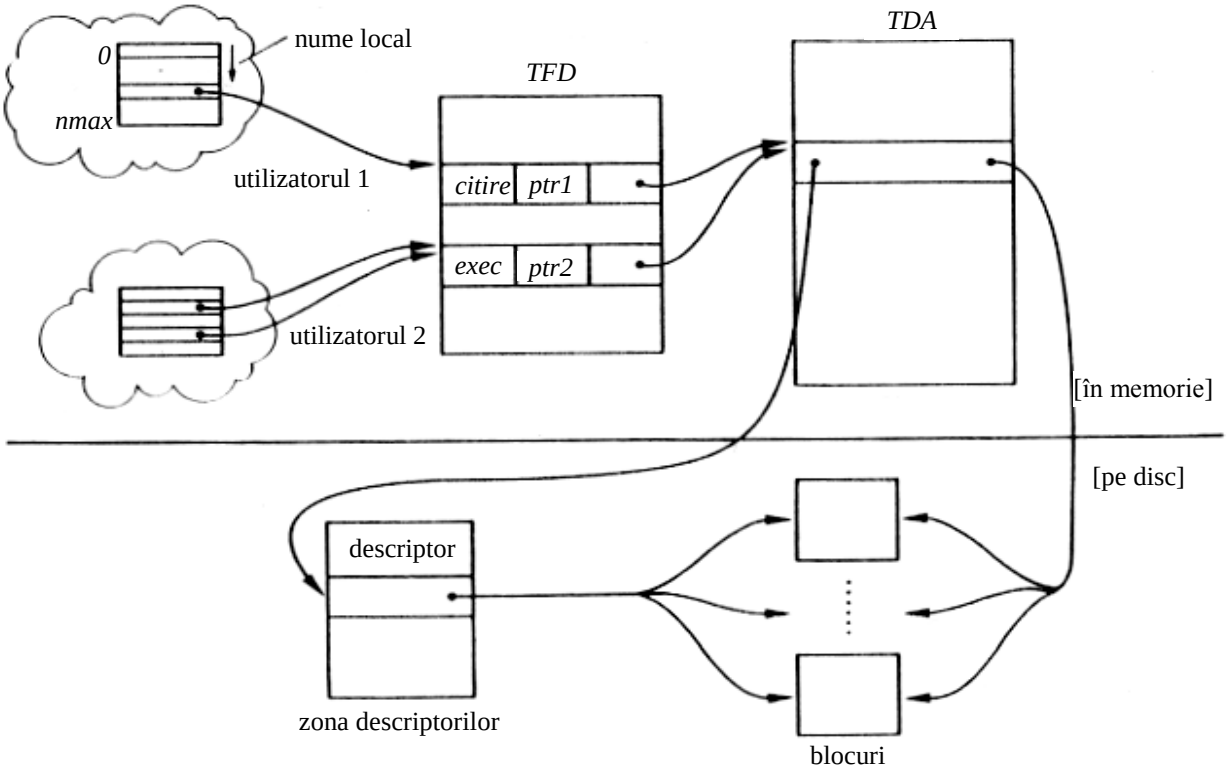


Fig.6.12. Descriptorii fişierelor în Unix

6.6.3. Funcţionarea şi utilizarea

6.6.3.1. Gestiunea descriptorilor. Nume locale

Din considerente de economie, doar descriptorii fişierelor deschise au o copie în memoria centrală. Aceşti descriptori, zişi activi, sunt grupaţi într-un tabel unic (*TDA*). Un fişier poate fi deschis simultan de mai mulţi utilizatori, fiecare utilizându-l independent cu ajutorul unui pointer de citire-scriere propriu. Acest pointer nu poate fi conţinut în descriptorul fişierului, care este unic: el este păstrat într-un tabel rezident, numit tabel al fişierelor deschise (*TFD*). La fiecare nouă deschidere în acest tabel este creată o intrare nouă şi mai multe intrări pot desemna acelaşi fişier, deschis

simultan pentru mai mulţi utilizatori. Plus la pointerul de citire-scriere, această intrare conţine modul de deschidere a fişierului şi un pointer la o intrare a *TDA* (fig.6.12).

În sfârşit, pentru a reduce frecvenţa căutărilor în cataloage şi a simplifica desemnarea, un utilizator poate desemna un fişier printr-un **nume local**. Acest nume, care este un întreg din intervalul *0..nmax* (*nmax* având valori până la 20), este alocat la deschiderea fişierului. El serveşte ca indice într-un tabel al numelor locale, păstrat în spaţiul propriu al fiecărui utilizator. Intrările acestui tabel sunt pointeri la *TFD*. Să ne amintim, că un nume local este de fapt un nume de flux, care poate în egală măsură desemna un fişier sau un tub (v.5.2.3.1).

6.6.3.2. Primitive de acces

Funcţiile de acces, descrise mai jos sunt primitive realizate prin apelarea regimului supervizor. Ele sunt utilizate pentru a realiza operaţiile disponibile în limbajul de comandă. Fiecare primitivă returnează un rezultat de tip întreg, care serveşte în special pentru a semnala erorile posibile.

Primitiva *dup* (duplicare) permite crearea unui alt nume local pentru un fişier deja deschis. Ea serveşte în special pentru a conecta tuburile şi pentru a determina, dacă un nume local dat este al unui fişier deschis.

Citirea şi scrierea are loc pornind de la o poziţie curentă, definită de un pointer. Aceste operaţii incrementează în mod automat pointerul cu numărul de caractere citite sau scrise. Primitiva *lseek* permite deplasarea explicită a pointerului.

Denumirea operaţiei	**Efectul**	**Rezultatul**
open(nume fişier, mod de acces)	deschide fişierul cu modul de acces indicat	nume local alocat (-1 dacă eşec)
close(nbloc)	abrogă asocierea între numele local *nbloc* şi fişierul (sau tub) asociat	succes → 0, eroare → -1
dup(nbloc)	dacă *nbloc* este numele local al unui fişier deschis, alocă un alt nume local *nbloc1* sinonimul lui *nbloc* (desemnând acelaşi fişier)	succes → *nbloc1*, eroare → -1
read(nbloc,tampon,ncar)	citeşte *ncar* caractere din fişierul (sau tub) cu numele local *nbloc* în zona *tampon*	numărul de caractere citite efectiv (mai mic decât *ncar*, dacă s-a ajuns la sfârşitul fişierului)
write(nbloc,tampon,ncar)	scrie *ncar* caractere din zona *tampon* în fişierul (sau tubul) cu numele local *nbloc*	numărul de caractere efectiv scrise
lseek(nbloc,depl,orig)	deplasează pointerul de citire-scriere *depl* caractere, pornind de la poziţia definită de *orig:* *0* – începutul fişierului, *1* – poziţia curentă a pointerului *2* – sfârşitul fişierului	poziţia curentă nouă, (-1 în caz de eroare)

Notăm, că este posibil să deplasăm pointerul după sfârşitul fişierului. Efectul este sporirea lungimii fişierului, poziţiile intermediare rămânând goale; aceste "găuri" nu ocupă loc în memorie.

6.6.3.3. Protecţia fişierelor în UNIX

Protecţia este asigurată cu ajutorul listelor de acces conform principiului din 6.5.3. Există trei moduri de acces elementar: citire, execuţie, scriere pentru un fişier; consultare, acces, modificare pentru un catalog.

Pentru controlul drepturilor, utilizatorii nu sunt individualizaţi, ci partajaţi în trei clase: proprietarul fişierului, membrii unui grup de utilizatori, restul utilizatorilor. Lista de acces la un fişier este, deci, de lungime fixă; ea enumără drepturile de acces ataşate fiecărei grupe; ea poate fi modificată doar de proprietarul fişierului sau de administratorul sistemului. O caracteristică originală a sistemului de protecţie este că proprietarul unui fişier poate fi schimbat.

De la începuturi UNIX a fost gândit ca un sistem de operare interactiv. Pentru a începe lucrul un utilizator trebuie „să intre" în sistem, introducând de la terminal numele propriu de evidenţă (contul, account name) şi, poate, parola (password), devenind astfel utilizator înregistrat al sistemului. Noii utilizatori sunt introduşi (înregistraţi) de către administrator. Utilizatorul nu-şi poate modifica contul propriu, în schimb parola este la discreţia lui. Parolele sunt codificate şi păstrate într-un fişier separat. Nici administratorul nu poate restabili o parolă uitată. Fiecărui utilizator înregistrat îi corespunde un catalog propriu al SGF, numit home. La intrare utilizatorului i se conferă acces total la acest catalog, inclusiv şi la toate cataloagele şi fişierele, care se conţin aici. Accesul la alte cataloage şi fişiere poate fi restricţionat.

Deoarece UNIX este un sistem de operare multiuser, o problemă foarte actuală este problema autorizării accesului utilizatorilor la fişierele SGF. Prin „autorizarea accesului" sunt considerate acţiunile sistemului, care conduc la permiterea sau interzicerea accesului unui utilizator concret la un fişier dat în dependenţă de drepturile de acces ale utilizatorului şi

restricţiile de acces, asociate fişierului. Schema autorizării accesului în UNIX este foarte simplă, în acelaşi timp foarte comodă şi puternică, încât a devenit standard de facto pentru sistemele contemporane.

Fiecărui proces în SO UNIX îi sunt asociate un identificator *real* şi un identificator *activ* al utilizatorului. Aceşti identificatori sunt desemnaţi cu ajutorul apelului de sistem setuid, care poate fi executat doar în mod supervizor. Analogic, de fiecare proces sunt legaţi doi identificatori ai grupului de utilizatori – unul real şi unul activ. Aceştia sunt desemnaţi cu ajutorul apelului de sistem privilegiat setgid. La intrarea utilizatorului în sistem programul login controlează, dacă utilizatorul este înregistrat în sistem şi a introdus parola corectă (dacă parola este necesară), formează un proces nou şi lansează în acest proces mediul (shell) utilizatorului dat. Dar, înainte de aceasta login setează, pentru procesul nou creat, identificatorii utilizatorului şi grupului, folosind informaţia din fişierele /etc/passwd şi /etc/group. Restricţiile de accesare a fişierelor devin active după legarea acestor identificatori de proces. Procesul poate avea acces la un fişier sau să execute un fişier executabil numai dacă restricţiile de accesare, păstrate cu fişierul permit aceasta. Identificatorii procesului sunt transmişi tuturor proceselor descendente, extinzând pentru acestea aceleaşi restricţii. În unele cazuri un proces îşi poate modifica drepturile sale utilizând apelurile de sistem setuid şi setgid, iar în alte cazuri sistemul poate modifica drepturile de accesare ale procesului în mod automat.

Fie ca exemplu următoarea situaţie. Scrierea în fişierul /etc/passwd este permisă doar administratorului (superuserul poate scrie în orice fişier). Acest fişier conţine, printre altele, parolele utilizatorilor şi fiecare utilizator are dreptul să-şi modifice parola. Există un program special /bin/passwd, care realizează modificarea parolelor. Însă un utilizator obişnuit nu poate face acest lucru chiar utilizând acest program, deoarece scriere în /etc/passwd este interzisă. În UNIX problema dată este rezolvată în modul următor. Pentru un fişier executabil poate fi indicat, că la lansarea lui vor fi setaţi identificatorii utilizatorului şi/sau grupului. Dacă utilizatorul cere lansarea unui atare program (cu ajutorul apelului de sistem exec), atunci pentru procesul respectiv este stabilit identificatorul utilizatorului, care corespunde identificatorului stăpânului fişierului executabil şi/sau identificatorul grupului acestui stăpân. În particular, la lansarea programului /bin/passwd procesului se atribuie identificatorul administratorului şi programul poate scrie în fişierul /etc/passwd. Pentru identificatorul utilizatorului şi pentru identificatorul grupului ID real este identificatorul adevărat, iar ID activ este identificatorul execuţiei curente. Dacă identificatorul curent al utilizatorului corespunde identificatorului administratorului, atunci acest identificator ca şi identificatorul grupului poate fi modificat cu ajutorul apelurilor de sistem setuid şi setgid. În cazul în care identificatorul curent al utilizatorului diferă de identificatorul administratorului, atunci execuţia apelurilor de sistem setuid şi setgid conduce la înlocuirea identificatorului curent cu identificatorul adevărat (al utilizatorului sau al grupului, respectiv).

În UNIX, ca şi în orice sistem de operare multiuser, este susţinut un mecanism standard de control al accesului la fişierele şi directoriile sistemului de fişiere. Orice proces poate accesa un fişier atunci şi numai atunci, când drepturile de acces, ataşate fişierului, corespund posibilităţilor procesului dat.

Protecţia fişierelor contra unor accese nesancţionate se bazează pe trei momente. Unu: fiecărui proces, care crează un fişier (sau un director) este ataşat un identificator al utilizatorului (identificator unic în sistem), care mai apoi va fi considerat ca identificatorul posesorului fişierului nou creat. Doi: de fiecare proces, care încearcă să acceseze un fişier, este legat un cuplu de identificatori – identificatorii activi al utilizatorului şi al grupului utilizatorului. Trei: fiecărui fişier în mod univoc îi corespunde descriptorul propriu – *i-node*. Informaţia dintr-un *i-node* conţine identificatorii posesorului curent (imediat după crearea fişierului identificatorii posesorului curent sunt stabiliţi de către identificatorul respectiv activ al procesului părinte, dar mai târziu pot fi modificaţi cu ajutorul apelurilor de sistem chown şi chgrp). Tot aici se conţin indicaţii despre operaţiile permise posesorului fişierului, ce drepturi au asupra fişierului membrii grupului posesorului şi care sunt drepturile celorlalţi utilizatori.

6.7. Exerciţii la capitolul 6

7. Alocarea resurselor

În acest capitol prezentăm principiile de alocare a resurselor într-un sistem informatic. Sunt abordate două aspecte:

1. utilizarea rezultatelor teoriei firelor de aşteptare pentru analiza câtorva modele simple ale sistemelor informatice, în special pentru alocarea procesoarelor;
2. prezentarea unor metode de evitare a impasurilor (blocarea reciprocă a mai multor procese rezultată din alocarea eronată a unor resurse comune).

7.1. Noţiuni generale

7.1.1. Definiţii

Numim **resursă** orice obiect, care poate fi utilizat de către un proces. Unei resurse îi sunt asociate proceduri de acces, care permit utilizarea resursei şi reguli de utilizare, care constituie "modul de utilizare". Exemple de asemenea reguli au fost aduse în capitolul 4 (acces exclusiv, cititori-redactori, etc.). Nu există vre-o diferenţă fundamentală între noţiunea de obiect şi cea de resursă. Folosirea celor doi termeni semnifică mai mult o diferenţă de domeniu: se utilizează noţiunea obiect atunci când ne interesează specificarea şi realizarea obiectului şi a funcţiilor sale de acces, iar resursă pentru problemele de alocare şi partajare.

Se spune că o resursă este **alocată** unui proces dacă procesul poate utiliza resursa, folosind procedurile ei de acces. Legarea unei resurse de un proces poate fi implicită sau explicită. În primul caz, este suficientă trecerea procesului într-o anumită stare pentru ca el să devină candidat la utilizarea resursei; în cel de-al doilea caz, solicitarea trebuie să fie formulată explicit sub forma unei cereri adresate unui **alocator** al resursei. Alocatoarele şi cererile pot lua forme diverse: de exemplu, un alocator poate fi un proces căruia cererile îi sunt transmise prin emiterea unor mesaje; sau chiar un monitor (v. capitolul 4) o procedură a căruia serveşte la emiterea unor mesaje. Reacţia unui alocator la o cerere poate de asemenea fi diferită: alocarea resursei solicitate, refuz cu sau fără blocarea procesului solicitant. Terminarea utilizării unei resurse de către un proces poate la fel lua diferite forme: **eliberarea** explicită sau implicită a resursei sau **retragerea** forţată a resursei de către alocator. O resursă se numeşte **banalizată** dacă ea există în mai multe exemplare echivalente, în sensul că o cerere poate fi satisfăcuta de alocator folosind indiferent care exemplar al resursei. Aceste noţiuni sunt precizate de câteva exemple.

Exemplul 7.1. **Procesorul.** Procedurile "de acces" la procesor sunt determinate de instrucţiuni şi cuvântul de stare; alocarea procesorului unui proces se face prin încărcarea cuvântului său de stare. Nu există cereri explicite: pentru a fi în stare să depună o astfel de cerere, procesul ar trebui să posede deja procesorul – obiectul cererii! Un proces devine în mod implicit candidat la utilizarea procesorului din momentul când trece în starea *eligibil*. Alocatorul constă din două proceduri: *planificatorul* şi *dispecerul*. ◄

Exemplul 7.2. **Memoria principală.** Alocarea memoriei principale pune în funcţie două mecanisme distincte:

a) Alocarea explicită: este mecanismul de obţinere a unei zone de memorie fie de către un program în curs de execuţie, fie de către sistemul de operare înaintea încărcării unui program nou.

b) Alocarea implicită: pentru acest mod de alocare cererea este emisă în momentul executării unei instrucţiuni, când adresa emisă de procesor este adresa unui amplasament nealocat procesorului. Acest mecanism stă la baza realizării memoriei virtuale. ◄

Exemplul 7.3. **Memoria secundară.** Memoria secundară este alocată prin blocuri banalizate de lungime fixă. Alocarea şi eliberarea poate fi explicită sau implicită (de exemplu, extinderea unui fişier provoacă emiterea unei cereri pentru un bloc suplimentar). ◄

Exemplul 7.4. **Linii de comunicaţie.** Dacă mai multe procese partajează serviciile unui dispozitiv de intrare-ieşire comun (o imprimantă sau o linie de comunicaţie, de exemplu), cererile sunt transmise sub formă de mesaje unui proces, numit server, care administrează acest dispozitiv. Fiecare mesaj conţine informaţiile necesare execuţiei transferului cerut. Mesajele sunt administrate cu ajutorul unui fir de aşteptare cu sau fără priorităţi. ◄

7.1.2. Probleme în alocarea resurselor

Obiectivul unui sistem de alocare a resurselor este să satisfacă cererile într-un mod echitabil, asigurând în acelaşi timp şi performanţe acceptabile. Dacă toate procesele au aceeaşi prioritate, o alocare este **echitabilă**, atunci când este asigurată o tratare asemănătoare fiecărui proces, de exemplu, aşteptarea medie este identică. Dacă între procese există priorităţi, este dificil să se definească noţiunea de echitate în caz general, dar ea poate fi exprimată prin faptul, că calitatea serviciului (măsurată, de exemplu, prin valoarea inversă a timpului mediu de aşteptare sau printr-o altă expresie semnificativă) este funcţie crescătoare de prioritate, două procese de aceeaşi prioritate beneficiind de aceeaşi tratare. A fost stabilit, că unii algoritmi de alocare conţin riscul de a face un proces să aştepte la infinit: acest fenomen, numit **privaţiune**, este contrar principiului de echitate şi trebuie evitat.

Dacă resursele sunt utilizate de mai multe procese pot avea loc două fenomene nedorite:

- **Blocarea,** deja întâlnită în capitolul 4 în cazul imbricării secţiunilor critice, este stoparea a mai multor procese pentru un interval de timp nedefinit, fiecare dintre procese fiind în aşteptarea eliberării resurselor alocate altor procese.

- **Căderea –** este situaţia în care o cerere excesivă a unui oarecare tip de resurse conduce la degradarea performanţelor sistemului (căderea unei reţele de transport sau a unei centrale telefonice). Situaţii analogice sunt constatate şi în sistemele informatice.

Vom presupune, că alocarea se va face în mod centralizat, adică algoritmii utilizaţi ştiu situaţia de alocare a mulţimii resurselor la orice moment.

Instrumentul principal, utilizat pentru studiul cantitativ a alocării resurselor sunt modelele firelor de aşteptare.

7.1.3. Exemple de sisteme cu fire de aşteptare

Un sistem informatic este constituit din *N* instalaţii identice, fiecare fiind descrisă de modelul M/M/1, cu parametrii λ şi μ (fig. 7.1a). Vom examina influenţa asupra timpului de răspuns al acestui sistem a utilizării comune a resurselor, caracteristicile totale de tratare fiind constante.

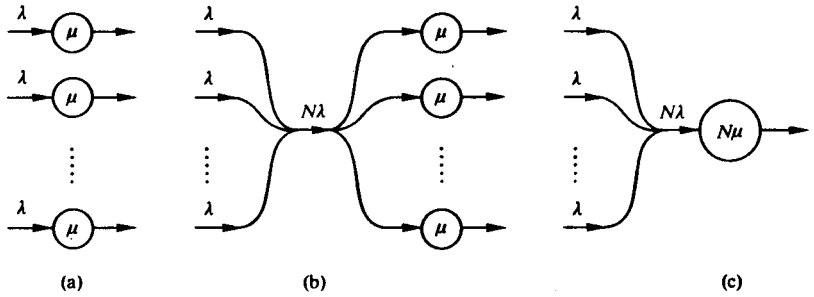


Fig.7.1. Modelul unui sistem de servire în masă

În sistemul din figura 7.1b, clienţii celor *N* fire de aşteptare sunt grupaţi într-un fir unic. Intensitatea sosirii, fiind superpoziţia unor intensităţi poissoniene, va rămâne de tip Poisson cu parametrul $N\lambda$. Sistemul este M/M/N cu parametrii $N\lambda$ şi μ.

În sistemul din figura 7.1c, *N* servere paralele sunt înlocuite printr-un server unic, de *N* ori mai puternic. Puterea totală de tratare rămâne în acest caz constantă, dar timpul mediu de servire a unei cereri se împarte la *N*: avem un sistem de tipul M/M/1 cu parametrii $N\lambda$ şi $N\mu$.

Timpul mediu de răspuns (timpul mediu de aflare în sistem a unei cereri) pentru fiecare caz va fi:

$$t_{ma} = 1/[\mu(1-\rho)]$$
$$t_{mb} = 1/\mu + C(N,N\rho)/[N\mu(1-\rho)]$$
$$t_{mc} = 1/[N\mu(1-\rho)]$$

Aici avem $t_{mc} = t_{ma}/N$ şi putem demonstra, calculând $C(N, N\rho)$, că $t_{ma} > t_{mb} > t_{mc}$. De exemplu, pentru $N=2$, $\mu=1$ şi $\rho=0,8$ vom avea $t_{ma} = 5$, $t_{mb} = 2,8$, $t_{mc} = 2,5$.

Acest rezultat ilustrează câştigul adus de concentrarea resurselor. Intuitiv, fenomenul poate fi explicat astfel: în cazul *a* un server poate fi neutilizat, atunci când avem cereri în alte fire de aşteptare, serverele acestora fiind ocupate. Această situaţie nu se poate produce în cazul *b*, însă dacă există servere neutilizate, posibilitatea lor de tratare nu este folosită, deşi ele ar putea servi la accelerarea tratării, care are loc pe serverele ocupate. Sistemul *c* realizează folosirea deplină a resurselor.

7.2. Modele pentru alocarea unei resurse unice

Vom trata două cazuri: alocarea procesorului şi gestiunea schimburilor cu o memorie secundară paginată.

7.2.1. Alocarea procesorului

7.2.1.1. Introducere

Algoritmii de alocare a procesorului pot fi clasificaţi în funcţie de mai multe caracteristici.

1) Algoritmi cu sau fără retragere

Retragerea presupune alocarea procesorului unui alt proces înainte ca procesul curent să-şi termine execuţia. Ea permite să se ia în consideraţie:

- prioritatea cererilor nou sosite,
- timpul deja consumat de către procesul ales.

2) Noţiuni de prioritate şi modul de determinare a priorităţii

Prioritatea unui proces este o informaţie, care permite clasificarea procesului în cadrul altor procese, dacă trebuie făcută o alegere (intrarea sau ieşirea dintr-un fir de aşteptare, de exemplu). Prioritatea poate fi definită prin mai multe moduri:

- prioritate constantă: definită apriori, funcţie de timpul de servire cerut.
- prioritate variabilă în timp: funcţie de timpul de aşteptare scurs, de serviciul deja acordat, etc.

Variabila principală, care va prezenta interes este timpul mediu de răspuns. Cererea este caracterizată de repartiţia intervalelor de sosire şi de timpul de servire. Aceşti parametri nu sunt întotdeauna bine cunoscuţi şi este important să se evidenţieze influenţa variaţiei lor.

Din practică este cunoscut, că un utilizator suportă cu atât mai greu aşteptarea servirii unei cereri cu cât lucrul trimis de el este de durată mai mică; aceasta este adevărat atât pentru lucrul interactiv, cât şi pentru cel de fond. Din aceste considerente metodele de alocare a procesorului au drept scop *reducerea timpului mediu de aşteptare a lucrărilor celor mai scurte.* Ele diferă conform modului de estimare a duratei lucrărilor şi în dependenţă de privilegiile acordate lucrărilor funcţie de această durată estimată.

7.2.1.2. Prim sosit, prim servit

Disciplina "Prim sosit, prim servit" (în engleză, "First In, First Out", sau FIFO) utilizează un fir unic, fără prioritate şi fără retragere. Procesul ales este executat până la terminarea sa, iar cel care va urma este procesul din topul firului de aşteptare. Un proces nou intră în coada firului.

Pentru modelul M/G/1 timpul mediu de răspuns este dat de formula Pollaczek-Khintchine:

$$t_m = t_{ms}[1+\rho(1+C^2_{ts})/2(1-\rho)], \qquad C^2_{ts} = Var(t_s)/(t_{ms})^2 \tag{7.1}$$

Figura 7.2 arată variaţia timpului mediu de răspuns funcţie de încărcarea sistemului şi de variaţia timpului de servire. Putem remarca:

- creşterea rapidă a timpului mediu de răspuns, atunci când sistemul se apropie de saturaţie: un sistem este cu atât mai sensibil la o variaţie a şarjei, cu cât este mai ridicat gradul lui de încărcare,
- influenţa **dispersiei** timpilor de servire asupra timpilor de răspuns.

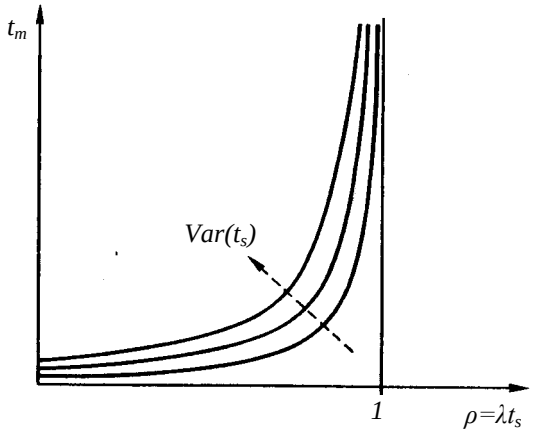


Fig.7.2. Timpul mediu de răspuns pentru disciplina FIFO

Aceste două efecte pot fi întâlnite în toate disciplinele de alocare.

7.2.1.3. Cererea cea mai scurtă servită prima

În acest caz cererile sunt ordonate în cadrul firului de aşteptare conform timpului lor de servire, presupus apriori cunoscut, cererile cu durata de servire cea mai scurtă aflându-se în top.

Timpul mediu de aşteptare t_m funcţie de timpul de servire t_s este dat de formula (7.1):

$$t_m(t_s) = \frac{\lambda t^2_{m_s}(1+C^2_{t_s})}{2[1-\rho(t_s)]^2} \quad \text{cu} \quad \rho(t_s) = \lambda \int_0^{t_s} t dB(t) \tag{7.2}$$

Formulele de aproximare sunt

- pentru valori mici ale t_s, unde $\rho(t_s)$ poate fi neglijat:

$$t_m(t_s) \approx \frac{\lambda t^2_{m_s}(1+C^2_{t_s})}{2} \tag{7.3}$$

- pentru valori mari ale t_s, unde $\rho(t_s)$ este aproape de λt_{ms}, fie ρ:

$$t_m(t_s) \approx \frac{\lambda t_{m_s}^2 (1 + C_{t_s}^2)}{2(1-\rho)^2} \qquad (7.4)$$

Figura 7.3 prezintă dependenţa timpului mediu de răspuns de timpul de servire cerut pentru cazurile FIFO şi Cererea cea mai Scurtă Servită Prima (*CSSP).* Pentru disciplina de servire FIFO acest timp depinde doar de gradul încărcării sistemului ρ, considerat de noi constant. Figura pune în evidenţă tratarea privilegiată a lucrărilor scurte.

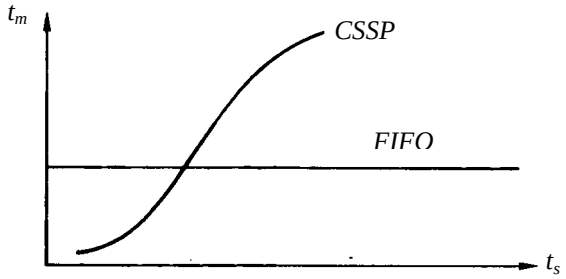


Fig.7.3. Dependenţa timpului mediu de răspuns de timpul de servire în cazul disciplinelor *FIFO* şi *CSSP*

În plan practic disciplina *CSSP* are două dezavantaje: ea prezintă un risc de privaţiune pentru cererile lungi, dacă rata de sosire a cererilor scurte este ridicată şi necesită cunoaşterea apriori exactă a timpului de servire.

O modalitate de evitare a riscului de privaţiune este de a acorda cererilor priorităţi care vor creşte odată cu creşterea timpului lor de aflare în firul de aşteptare, prioritatea iniţială fiind determinată de timpul de servire estimat. De exemplu, în cadrul disciplinei *HRN* (Highest Response Ratio Next) prioritatea unei cereri la un moment de timp τ este dată de relaţia

$$p(t) = [t(\tau)+t_s)]/t_s \qquad (7.5)$$

unde $t(\tau)$ este timpul de aflare a unei cereri în firul de aşteptare (fie $t(\tau)= \tau - \tau_s$, aici τ_s fiind momentul sosirii cererii), iar t_s timpul de servire estimat. Pentru două cereri cu acelaşi timp de aşteptare, prioritate are cererea mai scurtă. În practică, priorităţile sunt recalculate în momente discrete de timp şi firul de aşteptare este reordonat la necesitate.

Ideea directoare a acestui algoritm este de a partaja echitabil procesorul, încercând să se menţină o valoare constantă a raportului $k = t/t_s$, unde t este timpul total de aflare a unei cereri în sistem. Dacă aceasta se respectă, totul se va petrece pentru fiecare lucrare ca şi cum ea ar dispune de un procesor de k ori mai slab (mai puţin rapid) decât procesorul real. Disciplina *HRN* privilegiază lucrările scurte, însă penalizarea lucrărilor de lungă durată este redusă de efectul priorităţii variabile.

Algoritmii descrişi mai jos vizează la fel partajarea echitabilă a procesorului, dar nu necesită cunoaşterea apriori a timpului de servire.

7.2.1.4. Caruselul şi modele derivate

În modelul **caruselului** ("round robin") procesorul este alocat succesiv proceselor eligibile pentru o perioadă constantă de timp τ, numită cuantă. Dacă un proces se termină sau se blochează înainte de sfârşitul cuantei, procesorul este imediat alocat procesului următor. Această disciplină de servire este implementată ordonând procesele într-un fir de aşteptare circular; un pointer de activare, care avansează cu o poziţie la fiecare realocare a procesorului, desemnează procesul ales (fig.7.4).

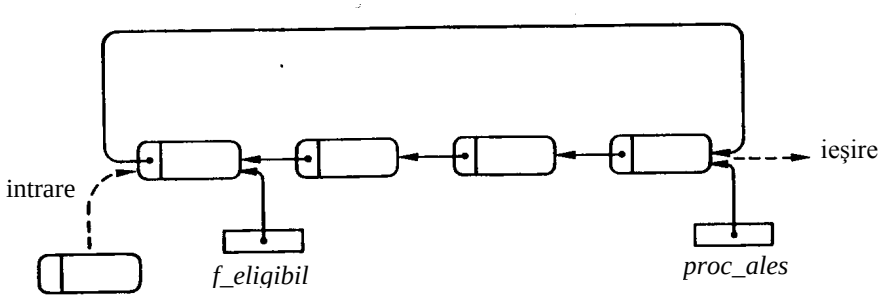


Fig.7.4. Alocarea procesorului: metoda caruselului

Caruselul este un exemplu de disciplină cu retragere. Ea poate fi considerată o aproximare a disciplinei "cel mai scurt primul" pentru cazul când durata execuţiei lucrărilor nu este cunoscută anticipat. Este utilizată în sistemele interactive; valoarea cuantei este aleasă astfel ca majoritatea cererilor interactive să poată fi executate în timpul unei cuante. Evident, valoarea cuantei trebuie să fie net mai mare decât durata comutării procesorului.

Analiza modelului caruselului nu este defel simplă. Doar cazul limită ($\tau \to 0$, iar timpul de comutare a procesorului neglijat) este mai simplu. În acest model, zis **procesor partajat**, totul are loc ca şi cum fiecare proces dispune imediat

(fără aşteptare) de un procesor viteza de procesare a căruia este cea a procesorului real, divizată la numărul curent al proceselor servite. Se poate arăta că timpul de aşteptare în modelul procesorului partajat este

$$t_a(t_s) = \rho t_s/(1-\rho),$$

iar timpul de răspuns este

$$t(t_s) = t_s/(1-\rho).$$

Modelul procesorului partajat realizează, deci, o alocare echitabilă, în care timpul de aflare a unui proces în sistem este proporţional cu durata serviciului cerut.

O îmbunătăţire a modelului caruselului, chemată să reducă timpul de servire a lucrărilor de scurtă durată, este caruselul multinivel. Acest model conţine n fire de aşteptare $F_0, F_1,..., F_{n-1}$. Fiecărui fir F_i îi este asociată o cuantă proprie τ_i, valoarea căreia creşte odată cu i. O lucrare din F_i este servită doar dacă firele de număr inferior lui i sunt vide. Dacă o lucrare din F_i şi-a epuizat cuanta sa fără a se termina, ea intră în firul F_{i+1}; lucrările din F_{n-1} se întorc tot aici. Lucrările noi intră în F_0 (fig. 7.5).

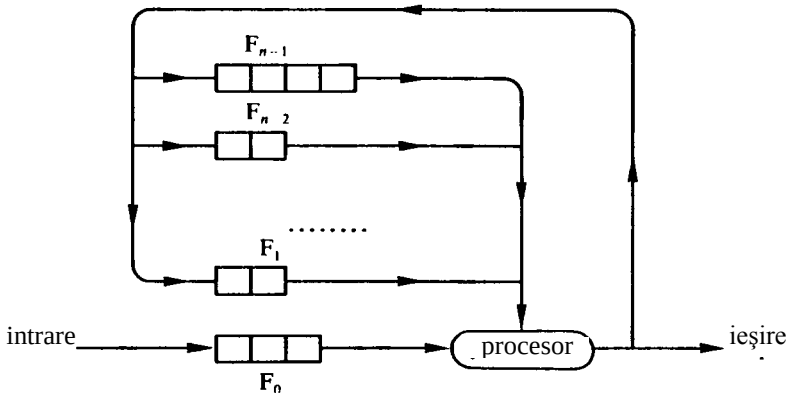
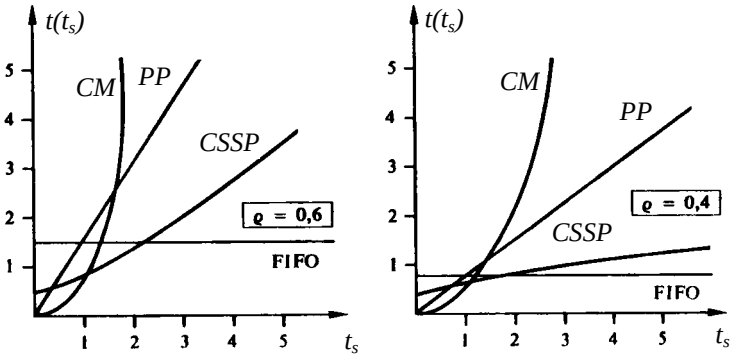


Fig. 7.5. Carusel cu mai multe niveluri

Această disciplină sporeşte privilegiile acordate lucrărilor de scurtă durată în raport cu *CSSP;* ca şi ultima, ea conţine un risc de privaţiune pentru lucrările de lungă durată. La fel poate fi definit un model limită pentru $\tau \to 0$, iar $n \to \infty$, pentru care poate fi calculată valoarea timpului mediu de aşteptare. Figura 7.6 ilustrează caracteristicile disciplinelor de servire examinate [16].



FIFO – prim sosit prim servit
CSSP – cerere scurtă servită prima
PP – procesor partajat
CM – carusel multinivel

Fig. 7.6. Compararea disciplinelor de alocare a procesorului

Putem aprecia, în conformitate cu gradul de încărcare a procesorului, avantajele acordate de diferite discipline de servire lucrărilor de scurtă durată.

7.2.2. Disc de paginare

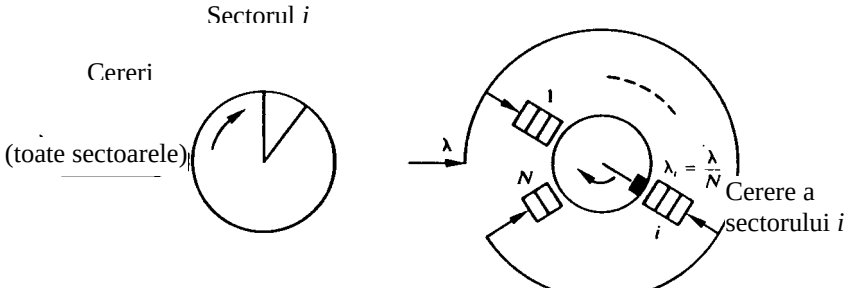
Vom cerceta acum metodele de gestiune a schimburilor între memoria principală şi un disc cu dispozitive de citire-scriere (DCS, capuri) fixe. Astfel de discuri sunt organizate pe piste şi sectoare şi are câte un DCS per pistă. Schimburile sunt comandate de un server care primeşte cererile de transfer; fiecare cerere presupune transferul unui sector cu indicaţia direcţiei de transfer, o adresă în memorie şi una pe disc (pista şi sectorul) [17].

Vom cerceta două discipline de tratare a cererilor:

1) Fir de aşteptare unic. Cererile de transfer sunt ordonate într-un fir unic şi tratate în ordinea sosirii lor,

2) Un fir per sector. Un fir distinct este administrat pentru fiecare sector; se speră să poată fi redus timpul mediu de aşteptare, eliminând timpul mort.

Aceste două discipline sunt prezentate schematic în figura 7.7. Un disc administrat conform schemei *b* este numit disc de paginare, deoarece acest mod de funcţionare este adesea utilizat pentru gestiunea memoriilor paginate.



(a) (b) Cu sectoare fixe şi DCS mobil

Fig. 7.7. Gestiunea transferelor memorie-disc

Vom compara aceste două discipline, în ipoteza unor cereri poissoniene la sosire şi repartizate uniform între sectoare. Notăm

λ fluxul mediu de sosire a cererilor,
R durata unei rotaţii a discului,
N numărul de sectoare pe pistă,

Durata de derulare a unui sector sub DCS este, deci, de R/N.

1) Fir unic de cereri. Discul poate fi considerat un server M/G/1, durata servirii ia N valori echiprobabile din $[R/N, (N-1)R/N]$. Timpul mediu de servire t_{ms} este în acest caz:

$$t_{ms} = \frac{1}{N}\sum_{i=1}^{N}\frac{iR}{N} = \frac{R(N+1)}{2N}, \tag{7.6}$$

iar coeficientul de variaţie

$$C_{t_s}^2 = \frac{1}{t_{m_s}^2}\sum_{i=1}^{N}\frac{1}{N}\left(\frac{iR}{N}\right)^2 - 1, \text{ sau } 1 + C_{t_s}^2 = \frac{2}{3}\left(\frac{2N+1}{N+1}\right) \tag{7.7}$$

Lungimea medie a unui fir de aşteptare, fiind inclusă şi cererea în curs de servire, este obţinută din formula Pollaczeck-Khinchine:

$$n_m = \lambda t_{ms}\left[1 + \frac{\lambda t_{m_s}(1 + C_{t_s}^2)}{2(1 - \lambda t_{m_s})}\right] \tag{7.8}$$

şi timpul mediu de servire este:

$$t_m = \frac{n_m}{\lambda} = \frac{R(N+1)}{2N}\left[1 + \frac{(2N+1)\lambda R}{3[2N - \lambda R(N+1)]}\right] \tag{7.9}$$

2) *Disc de paginare.* Fiecare fir poate fi tratat independent, utilizând modelul lui Skinner (server cu restabilire), cu

$\lambda_i = \lambda/N$ $t_s = R/N,$ $t_r = (N-1)R/N,$
$T = R$ $S = t_s + t_r = R$

Pentru un fir de aşteptare vid, ca şi pentru începerea tratării următoarei cereri, este necesar să se aştepte o rotaţie completă. Lungimea medie a unui fir de sector este:

$$n_{mi} = \frac{\lambda^2 R^2 / N^2}{2(1 - \lambda R/N)} + \frac{\lambda R}{N}\left(\frac{1}{N} + \frac{1}{2}\right) \tag{7.10}$$

şi timpul mediu de servire:

$$t_m = \frac{n_{mi}}{\lambda_i} = \frac{\lambda R^2 / N}{2(1 - \lambda R/N)} + R\frac{N+2}{2N}. \tag{7.11}$$

În figura 7.8 sunt comparate cele două modele de administrare a discului.

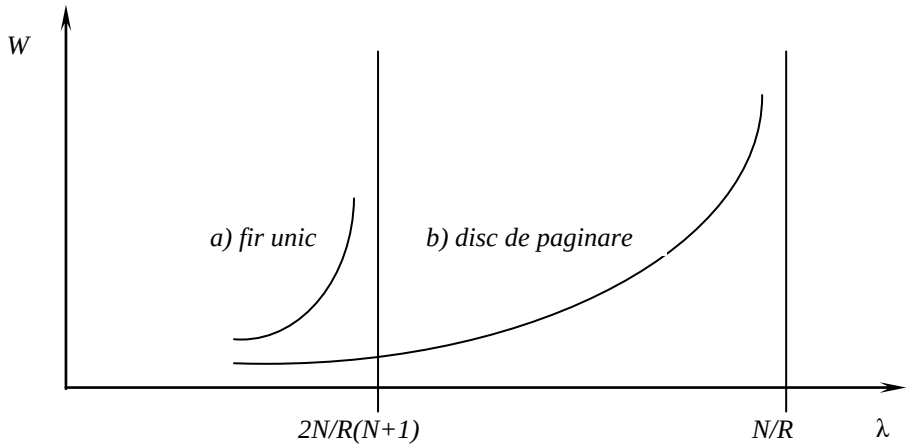


Fig. 7.8. Compararea modelelor de administrare a discului

Compararea pragurilor de saturare pentru cele două discipline permite calcularea câştigului obţinut pentru discul de paginare. Pragul de saturare este definit prin $\rho = 1$. Vom obţine în acest caz: pentru discul cu fir unic: $\lambda R(N+1)/2N = 1$, iar pentru discul de paginare: $\lambda R/N = 1$.

Raportul pragurilor critice pentru λ este, deci, $(N+1)/2$; acest raport este o măsură a câştigului obţinut prin organizarea firelor pentru sectoare.

7.3. Tratarea blocărilor

7.3.1. Enunţul problemei

Un exemplu de blocare reciprocă (impas) a fost prezentat în capitolul 4, când două secţiuni critice, corespunzătoare la două resurse distincte, sunt imbricate eronat, existând un risc de blocare pentru o perioadă de timp nedefinită.

O atare situaţie poate fi generalizată pentru un număr oarecare de procese şi resurse, dacă sunt îndeplinite următoarele condiţii:

1) resursele sunt utilizate în excludere mutuală,
2) fiecare proces trebuie să utilizeze simultan mai multe resurse, şi acaparează fiecare resursă pe măsura necesităţilor sale, fără a le elibera pe cele pe care le posedă deja,
3) cererile de resurse sunt blocante şi resursele nu pot fi retrase,
4) există o mulţime de procese $\{p_0,..., p_n\}$, astfel încât p_0 cere o resursă ocupată de p_1, p_1 cere o resursă ocupată de p_2, ..., p_n cere o resursă ocupată de p_0.

Exemplul 7.5. Două procese p_1 şi p_2 utilizează două tipuri de resurse, r_1 şi r_2. Sistemul conţine 7 unităţi de tipul r_1 şi 6 r_2. Cantităţile respective de r_1 şi r_2 necesare pentru execuţia proceselor sunt *6* şi *3* pentru p_1, *4* şi *5* pentru p_2. Presupunem că sistemul se află în starea următoare:

	r_1	r_2
alocat lui p_1	4	2
alocat lui p_2	2	3
disponibil	1	1

Oricare ar fi ordinea de execuţie a proceselor, nici unul nu va putea obţine totalitatea resurselor necesare, dacă nici o resursă nu este eliberată sau retrasă. Pornind de la această stare a sistemului, impasul este inevitabil. ◄

Problema impasului poate fi abordată în două moduri:

- prin **prevenire**: algoritmul de alocare a resurselor garantează, că o situaţie de blocare nu se poate produce,
- prin **detectare** şi **tratare**: nu sunt luate măsuri preventive; blocarea este detectată doar dacă are loc, iar tratarea şi "însănătoşirea" constă în aducerea sistemului într-o stare, în care procesele sunt deblocate, în caz general, cu o pierdere de informaţii.

7.3.2. Algoritmi de prevenire

7.3.2.1. Algoritmi de profilaxie

Atunci când nu se cunoaşte anticipat nimic despre cererile proceselor, metodele de prevenire constau în înlăturarea uneia din condiţiile 1) – 4) de mai sus.

Metoda **alocării globale** vizează eliminarea condiţiei "2": fiecare proces cere toate resursele de care are nevoie în bloc; el poate elibera o resursă dacă nu mai are nevoie de ea. Metoda dată conduce la imobilizarea unor resurse în mod neproductiv, şi aduce adesea în practică la suprimarea oricărui paralelism în execuţie; totuşi această metodă nu trebuie refuzată din start.

Metoda **claselor ordonate** vizează eliminarea situaţiei unei aşteptări circulare (condiţia "4"). Resursele sunt divizate în clase $C_1, C_2,..., C_m$. Un proces poate primi o resursă din clasa C_i numai dacă el a primit deja toate resursele necesare în clasele $C_j, j<i$. Toate procesele cer resursele de care au nevoie în aceeaşi ordine, cea a claselor.

Presupunem, că avem o aşteptare circulară, adică există procese $p_0, p_1,..., p_{n-1}$ astfel încât p_i aşteaptă o resursă r_i alocată procesului $p_{i\oplus 1}$ (simbolul $\oplus$ desemnează suma modulo n). Fie $c(r_i)$ numărul clasei resursei r_i. Deoarece procesul $p_{i\oplus 1}$ aşteaptă resursa $r_{i\oplus 1}$ şi posedă resursa r_i, avem $c(r_i)<c(r_{i\oplus 1})$, de unde, datorită tranzitivităţii, obţinem $c(r_0)<c(r_0)$. Aşteptarea circulară este astfel imposibilă şi blocarea evitată.

7.3.2.2. Algoritmul bancherului

Algoritmul de prevenire poate fi îmbunătăţit dacă există informaţii anticipate despre cererile de resurse. Dacă fiecare proces prezintă în avans **o declaraţie de intenţii (anunţ)**, adică valoarea unei borne superioare ale cererilor sale pentru numărul de resurse de fiecare tip, algoritmul **bancherului** permite evitarea blocării reevaluând riscul acestora la fiecare alocare. Acest algoritm se bazează pe noţiunea de **stare fiabilă**: o stare de alocare este numită fiabilă dacă, pornind de la această stare este posibil să se asigure funcţionarea sistemului fără blocări în ipoteza cea mai pesimistă, cea în care fiecare proces cere efectiv cantitatea maximă presupusă de fiecare dintre resurse. Algoritmul bancherului va aloca o resursă doar dacă această alocare păstrează starea fiabilă a sistemului.

Considerăm un sistem care conţine n procese şi m clase de resurse. Starea sa este reprezentată de următoarele structuri de date:

Res, Disp: **array***[0..m-1]* **of** *integer*
Anunt, Aloc: **array***[0..n-1, 0..m-1]* **of** *integer*

unde

Res[j]: este numărul total de resurse din clasa *j*,
Anunt[i, j]: numărul maxim de resurse din clasa *j* necesare la un moment de timp procesului *i*,
Aloc[i, j]: numărul maxim de resurse din clasa *j* alocate la un moment de timp procesului *i*,
Disp[j]: numărul de resurse din clasa *j* disponibile la un moment de timp dat.

Avem, prin definiţie, pentru un moment oarecare de timp:

$Disp[j]=Res[j]-\sum_{i=0}^{n-1} Aloc[i, j]$ pentru $j=0, 1,..., m-1$.

Notăm prin *A[i,*]* vectorul format din linia *i* a matricei *A*. Dacă *U* şi *V* sunt doi vectori de aceeaşi lungime *k*, vom conveni să scriem:

$U \le V$ dacă şi numai dacă $U[i] \le V[i]$ pentru $i=0, 1,..., k-1$
$U<V$ dacă şi numai dacă $U \le V$ şi $U \ne V$.

Specificaţiile sistemului impun următoarele restricţii:

$0 \le Aloc[i,*] \le Anunt[i,*] \le Res$, pentru $i=0,...,n-1$
$Aloc[i,*] \le Res$
Disp ≥ 0 vectorul nul.

Aceste relaţii exprimă, pentru fiecare clasă de resurse, că anunţul nu poate depăşi numărul resurselor disponibile, că cantitatea resurselor alocate unui proces nu poate depăşi anunţul său şi, în sfârşit, că sistemul nu poate aloca mai multe resurse decât există în total. O stare a sistemului se numeşte **realizabilă**, dacă este conformă acestor specificaţii.

Fie E_0 o stare realizabilă a sistemului. Încercăm să găsim un proces p_i, care, dacă el era executat singur pornind de la starea E_0, utilizând totalitatea resurselor specificate de anunţul său, procesul ar putea ajunge până la sfârşit. Un asemenea proces trebuie să verifice , în starea E_0:

$Anunt[i, *]-Aloc[i, *] \le Disp.$

Presupunem că a fost găsit un astfel de proces, fie p_{i0}, şi că el este executat până la sfârşit. El eliberează atunci toate resursele sale şi sistemul trece într-o stare realizabilă E_1, definită prin

$Disp_{E1}=Disp_{E0}+Aloc[i_0,*]$.

Operaţia poate fi repetată, căutând un proces nou p_{i1} şi tot aşa cât este posibil. Suita de procese, obţinută în acest mod se numeşte **fiabilă**. O stare a sistemului se numeşte **fiabilă**, dacă, pornind din această stare, putem construi o suită fiabilă completă (adică, care conţine toate procesele sistemului).

Urmând această definiţie, un sistem într-o stare fiabilă nu este în blocare, deoarece poate fi definită o ordonare a proceselor, care permite execuţia tuturor proceselor. Putem demonstra afirmaţia reciprocă (exerciţiu): dacă un proces nu este în blocare, el se află într-o stare fiabilă. Algoritmul bancherului se bazează pe această proprietate; el este executat la fiecare cerere de resursă de un proces. Fie *Cerere[i, j]* numărul de resurse cerute din clasa *j* în starea curentă a sistemului, de către procesul *i*.

```
if Aloc[i,j]+Cerere[i,j]>Anunt[i,j] then
    <eroare>                              -- cererea totală > anunţul
else
    if Cerere[i,j]>Disp[j] then
```

```
            <pune procesul p în aşteptare>
        else                                        -- simularea alocării
            <definire stare nouă prin:
            Aloc[i,j]:=Aloc[i,j]+Cerere[i,j]
            Disp[j]:=Disp[j]-Cerere[i,j]>
        endif;
        if starea nouă este fiabilă then
            <efectuare alocare>
        else
            <restaurare stare primitivă>
            <pune procesul p în aşteptare>
        endif
    endif
```

Locul central al acestui algoritm îl constituie testarea fiabilităţii unei stări, care se reduce la încercarea construirii unei suite fiabile. Este vorba de un algoritm combinator, care presupune iniţial o complexitate de ordinul *n!*, însă complexitatea poate fi redusă la n^2 datorită următoarei proprietăţi, care elimină necesitatea întoarcerii înapoi în caz de eşec: *dintr-o stare fiabilă orice suită fiabilă poate fi prelungită până la o suită fiabilă completă*.

Cu alte cuvinte, dacă tentativa de extindere a unei suite fiabile parţiale eşuează, starea sistemului nu este fiabilă şi este inutil să se încerce construirea unei alte suite.

Ţinând cont de această proprietate, testarea fiabilităţii este realizată cu ajutorul algoritmului de mai jos:

```
Dispcurente  : array[0..m-1] of integer;
Rest         : set of procese;

Dispcurente:=Disponibil;
Rest:={toate procesele};
posibil:=true;
while posibil do
    căutare p din Rest astfel ca
        Anunt[i,*]-Aloc[i,*]≤Dispcurente;
    if găsit then                          -- simularea execuţiei procesului p
        Dispcurente:=Dispcurente+Aloc[i,*];
        Rest:=Rest-{p}
    else
        posibil:=false
    endif
endwhile;
stare fiabilă:=(Rest=<vid>)
```

Acest algoritm poate fi simplificat, dacă există o singură clasă de resurse.

7.3.3. Algoritmi de detectare şi tratare

Pentru detectarea unei blocări putem utiliza testul de fiabilitate a unei stări, înlocuind pentru fiecare proces cererea maximă *Anunt[i,*]-Aloc[i,*]* cu cererea efectivă la instanţa considerată. Dacă în aceste condiţii este imposibil să se găsească o suită fiabilă completă, sistemul se află în blocare. Mai precis, procesele blocate sunt elementele mulţimii *Rest*, la terminarea algoritmului.

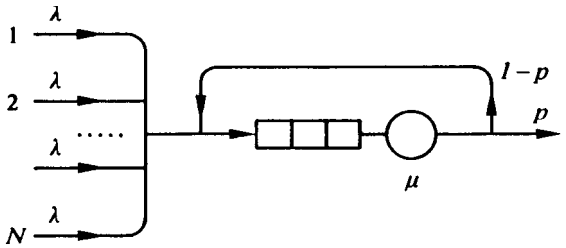
Lecuirea blocării vizează reluarea execuţiei sistemului; aceasta poate fi făcut doar recuperând resursele deja alocate. O metodă mai dură constă în distrugerea proceselor blocate, începând cu cele pentru care preţul distrugerii este cel mai mic (conform unui criteriu definit), până la procesele pentru care execuţia poate fi reluată. Această metodă nu este fără pericole, dacă procesele distruse au lăsat date într-o stare incoerentă. De asemenea este preferabil să se salveze periodic starea proceselor, definind **puncte de reluare**, corespunzătoare unor stări semnificative. În caz de retragere forţată a resurselor unui proces, acesta poate fi repus în starea corespunzătoare ultimului său punct de reluare.

Tratarea blocărilor prin distrugere sau întoarcerea la o stare anterioară prezintă un risc de privaţiune, dacă criteriul de preţ ales conduce întotdeauna la desemnarea aceluiaşi proces. Pentru remediere au fost propuse diferite metode: acordarea unor "privilegii" care ar proteja procesele protejate contra distrugerilor forţate sau factor de preţ crescător odată cu creşterea numărului de retrageri.

Atunci când retragerea este autorizată şi dacă ea are loc fără pierdere de informaţii, poate fi utilizată o formă particulară de detecţie şi lecuire, evitarea continuă. Această metodă este folosită pentru sistemele repartizate [18].

7.4. Exerciţii la capitolul 7

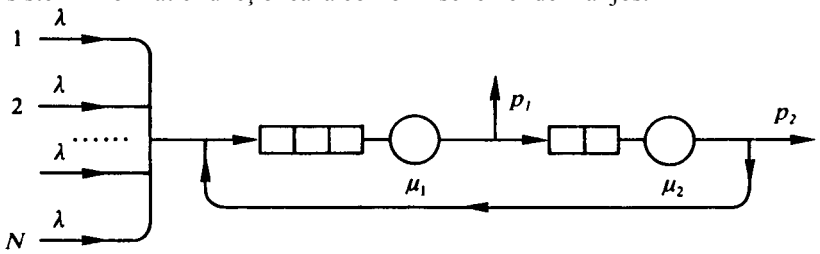
Exerciţiul 7.1. Un sistem informatic funcţionează conform schemei de mai jos.



Fiecare terminal este o sursă de cereri poissoniene pentru care timpul de servire are repartiţie exponenţială. Calculaţi:

- debitul la intrarea serverului,
- numărul de terminale, care aduce sistemul în starea de saturaţie, presupunând constante ceilalţi parametri.

Exerciţiul 7.2. Un sistem informatic funcţionează conform schemei de mai jos.



Cu aceleaşi ipoteze ca şi în exerciţiul precedent, calculaţi:

- debitele la intrarea celor două servere,
- numărul de terminale, care aduce sistemul în starea de saturaţie, presupunând constante ceilalţi parametri,
- relaţia care ar trebui să existe între parametrii sistemului pentru ca cele două servere să fie echilibrate (acelaşi grad de încărcare).

Exerciţiul 7.3. Să se arate, că disciplina "Lucrarea cea mai scurtă prima" minimizează timpul mediu de răspuns pentru o mulţime de lucrări. **Indicaţii**: Se va stabili timpul total de execuţie pentru o mulţime de lucrări, apoi se vor permuta două lucrări.

Exerciţiul 7.4. Două procese ciclice *p* şi *q* folosesc două resurse comune *A* şi *B*. Fiecare proces traversează o fază în care are nevoie de ambele resurse simultan, cerându-le una după alta, conform necesităţilor (deoarece alt proces poate avea nevoie de o singură resursă) şi eliberează ambele resurse la sfârşitul fazei. Dacă ordinea de achiziţie a resurselor este diferită pentru cele două procese, este oare posibil programarea lor pentru evitarea blocării fără retragere sau rezervare globală?

Exerciţiul 7.5. Considerăm un sistem de prelucrare pe loturi monoprogram, cu tampoane de intrare-ieşire în memorie, conform schemei din exerciţiul 3.13. Zonele de memorie, care servesc drept tampoane de intrare şi de ieşire sunt alocate şi eliberate în mod dinamic, la necesitate, pornind de la o zonă comună de lungime constantă.

Identificaţi riscul unei blocări şi propuneţi un algoritm pentru a o evita.

Exerciţiul 7.6. Trei procese partajează trei resurse nebanalizate cu acces exclusiv şi fără retragere. Utilizând procedurile *alocare(resursă)* şi *eliberare(resursă)*, elaboraţi algoritmul programelor pentru aceste procese:

- cu risc de blocare globală a celor trei procese,
- cu risc de privaţiune pentru doar unul din procese, fără blocare.

Fiecare proces nu va utiliza mai mult de două resurse.

Exerciţiul 7.7. Demonstraţi, că dacă un sistem nu se află în blocare, este posibil să se construiască o suită completă de procese.

Exerciţiul 7.8. Demonstraţi, că dacă un sistem se află într-o stare fiabilă, este posibil să se prelungească orice suită fiabilă de procese într-o suită fiabilă completă.

Exerciţiul 7.9. Să se elaboreze un program detaliat pentru alocarea resurselor conform algoritmului bancherului, introducând structurile de date necesare.

Exerciţiul 7.10. Arătaţi că, în caz general, complexitatea algoritmului bancherului este de ordinul n^2m, unde n este numărul total al proceselor şi m este numărul total de clase de resurse.

Exerciţiul 7.11. Completaţi algoritmul bancherului presupunând, că procesele pot fi create şi distruse dinamic şi, că un proces poate modifica anunţul său. Precizaţi, care sunt operaţiile care trebuie să fie supuse unei autorizări a sistemului de alocare a resurselor.