SISTEME DE OPERARE

CAP.I DESCRIEREA STRUCTURALA A UNUI SISTEM DE CALCUL

Scopul acestui capitol este de a se realiza o descriere structurala a unui sistem de calcul (calculator) care sa cuprinda atat caracteristicile hardware ale sistemului cat si caracteristicile software.

1. Relatii de ierarhizare intr-un sistem de calcul.

Vom definii:

Sistem calculator : o colectie de componente hardware si software care furnizeaza o forma definita de servicii unui TIP DE UTILIZATORI.

De obicei avem de a face cu ceea ce numim *calculator* - sau mai general *instalatie de calcul* - .
Deci o instalatie de calcul in acest moment o vom considera ca fiind constituita din mai multe *sisteme calculator* determinate de tipul de utilizatori considerati.
De ex.
Intr-o instalatie de calcul cu scop general care ofera posibilitatea de a executare a programelor scrise in limbajul C (de exemplu) vom deosebi cel putin trei *sisteme calculator* distincte, conform tipurilor corespunzatoare de utilizatori.

Sistem calculator	***Tipul de utilizatori***
1. Hardware-ul calculatorului	1. Implementatorii sistemului de operare
2. Hardware-ul calculatorului + Sistemul sau de Operare	2. Implementatorii subsistemului de programere (de ex. C)
3. Hardware-ul calculatorului + Sistemul de Operare + subsitemul de programe C	3. Utilizatorii limbajului de programare C

Orice sistem calculator defineste un *limbaj* in termenii caruia se exprima intreaga activitate ce se executa pe sistemul calculator respectiv.
Altfel spus, un sistem calculator are posibilitatea reprezentarii anumitor **tipuri de date** si **structuri de informatie** si implementeaza o multime de **operatii primitive** asupra tipurilor de date si structuri de informatie pe care le poate reprezenta.
Sa exemplificam pe cazul Sistemului Calculator compus numai din hardware (o unitate de prelucrare si o unitate de memorie).
In acest caz tipurile de date corespund interpretarii cuvintelor de memorie in unitatea de prelucrare ca fiind numere reprezentate in virgula fixa, virgula flotanta, instructiuni etc.
In acest sistem calculator operatiile primitive : sunt operatiile pe care le poate executa unitatea de prelucrare asupra continutului cuvintelor de memorie, operatii aritmetice, operatii de acces la memorie, etc.
Sa presupunem acum ca la unitatea centrala si unitatea de memorie existente se adauga:
-dispozitive periferice;
- un sistem de operare;

In acest fel am construit un alt sistem calculator caruia ii vor corespunde:
- noi tipuri de date si structuri de informatie;
- noi tipuri de operatii primitive relative la noile tipuri de date;

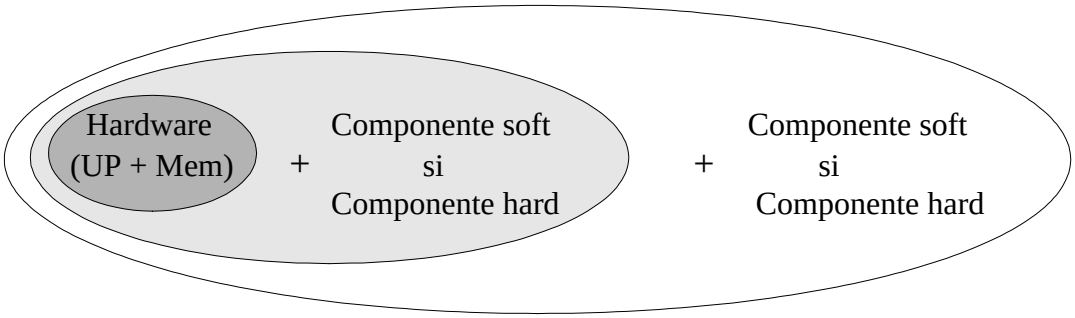
si ofera o forma definita de servicii unui alt tip de utilizatori.

Obs.
Unele caracteristici ale sistemului calculator generator al sistemului calculator actual nu mai sunt accesibile utilizatorului sistemului actual.
De ex. in acest caz mecanismul adreselor absolute sau tratarea intreruperilor nu mai sunt accesibile utilizatorilor sistemului de operare in mod direct.
Utilizatorul Sistemului de Operare va avea acces la acestea prin intermediul noului limbaj care-l defineste sistemul de operare adaugat.

Operatiile primitive asupra structurilor de date in acest nou limbaj pot fi acum foarte complexe considerate la nivelul structurii hardware. De ex. incarcarea si lansarea in executie al unui program.

Daca vom adauga sistemului calculator astfel format un sistem un subsistem pentru un limbaj de programare (mediu de programare) de exemplu "C" se va produce un nou sistem calculator determinat de noul tip de utilizatori (cunoscatori ai limbajului C). In acest nou sistem calculator se defineste un nou limbaj caracterizat de propriile tipuri de date, structuri de informatii si operatii primitive. Utilizatorii au acum acces la limbajul sistemului de operare numai prin folosirea subsistemului de programare C (in acest caz).

In cazul exemplului de mai sus *sistemul generator* pentru sistemul de operare este format din unitatea de prelucrare si memorie operativa.
Sistemul de operare devine sistem generator pentru un sistem de programare,



In conclzie:

Un sistem de calcul se poate considera ca o succesiune de nivele, fiecare nivel fiind un sistem calculator generator pentru sitemul urmator al succesiunii.

Nivelul cel mai interior fiind determinat de *Hardware-ul* calculatorului vom spune ca acesta *este sistemul generator de baza*.

Relatia intre diferite nivele este determinata de faptul ca *un nivel genereaza pe urmatorul.*

Aceasta relatie constituie *ierarhia sitemului de calcul* . Fiecare nivel al acestei ierarhii este un sistem calculator caracterizat de:

tipuri de date;
structuri de informatie;
operatii primitive.

Prin trecerea de la un nivel ierarhic inferior la un nivel ierarhic superior se face abstractie de o serie de proprietati ale nivelului inferior.
De exemplu trecerea de la hardware la sistemul de operare se face prin abstragerea de la proprietatile fizice ale componentelor hardware Aceasta inseamna ca utilizatorii noului sistem calculator format din Hardware si Sistem de Operare pot lua in considerarea numai proprietatile functionale care exista ca elemente in noul limbaj determinat de sistemul de operare.

2. Hardware-ul ca sistem generator de baza

Primul nivel al ierarhiei care o reprezinta sistemul de calcul este sistemul hardware. Relativ la utilizatorii sai sistemul calculator hardware trebuie sa indeplineasca urmatoarele functii:

f1. functia de receptionare a informatiei;
f2. functia de conservare a informatiei;
f3. functia de prelucrare a informatiei;
f4. functia de transmitere a informatiei.

Pentru realizarea acestor functiuni sistemul calculator hardware are o structura care intr-un anumit fel reflecta structura de ansamblu a intregului sistem de calcul.
Sistemul calculator hardware este constituit dintr-o multime ordonata de unitati functionale notata cu U .
Fiecare element u_i U indeplineste una sau mai multe functii de tipul f1÷f4.
Putem considera ca activitatea sistemului calculator U consta in executarea succesiva si/sau paralela a unui sir finit de functii : fi_1, fi_2, ...fi_n , de tipul f1 ÷f4.
Formalizat algebric aceasta inseamna calcularea valorii unei functii

$\Ö = fi_1 \text{ o } fi_2 \text{ o } \ldots \text{ o } fi_n$ unde fi_j {f1,f2,f3,f4} j=1,2,...n

Unde simbolul "o" reprezinta operatia de compunere.
Deci functionarea sistemului calculator la acest nivel inseamna executarea functiei

$\Ö = fi_1 \text{ o } fi_2 \text{ o } \ldots\ldots \text{ o } fi_n$ unde i=1,2,3,4

La executarea acestei functii vor concura diferite unitati functionale $\mathbf{u_i}$ U care in mod obligatoriu trebuie sa fie legate intre ele prin linii de comunicatie prin care se vor transmite informatiile in procesul de lucru al sistemului de calcul.
Pentru un sistem calculator real $\mathbf{u_i}$ sunt dispozitive fizice reale care realizeaz[a] functii de tipul f1÷f4.
Multimea unitatilor functionale $\mathbf{u_i}$ **U** se impart in clase dupa tipurile f1÷f4. pe care le executa.

- Unitatile functionale u_i , care prelucreaza informatia se numesc procesori se noteaza cu P sau UP . Ele realizeaza functii de tipul f3 si formeaze clasa $\mathbf{u_1}$.

- Unitatile functionale u_i care conserva informatia sunt numite memorii si le vom nota cu M, realizeaza functiile de tipul f2 si formeaza clasa $\mathbf{u_2}$.

- Unitatile functionale u_i, care realizeaza schimbul de informatie (receptie sau transmisie) denumite si unitati de schimb US. Realizeaza functii de tipul f1 si f4 si formeaza clasa $\mathbf{u_3}$.

Elementele u_1 , u_2, u_3 ale multimii U sunt <u>functional independente</u>, (Unitatile functionale ale sistemului calculator sunt functional independente.)

Functiile realizate de o unitate functionala se aplica asupra unor informatii transmise unitatii respective de catre alte unitati functionale prin <u>linii de comunicatie</u>.

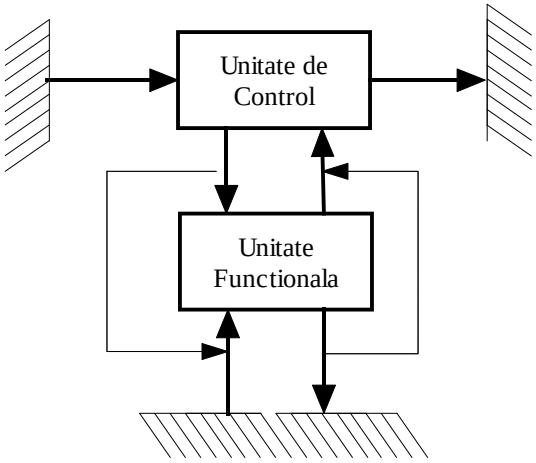
Fiecare unitate functionala u_i U poate fi considerata <u>un subsistem calculator</u> in raport cu sistemul calculator.

Pentru a putea asigura corectitudinea schimbului de informatie intre diferite subsisteme ale sistemului calculator fiecare subsistem este conceput ca fiind compus din:

<u>Unitate functionala</u> (propriuzisa) care se ocupa cu realizarea efectiva a functiei pentru care a fost construit subsistemul respectiv;

<u>Unitate de control</u> care se ocupa cu controlul activitatii unitatii functionale propriuzise si cu legaturile cu celelalte subsisteme.

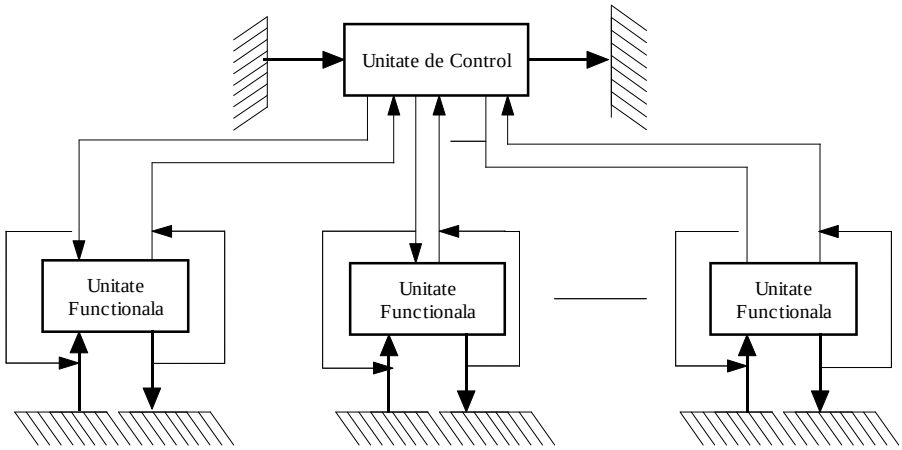
Unitatea functionala (subsistemul calculator) poate fi reprezentata grafic:



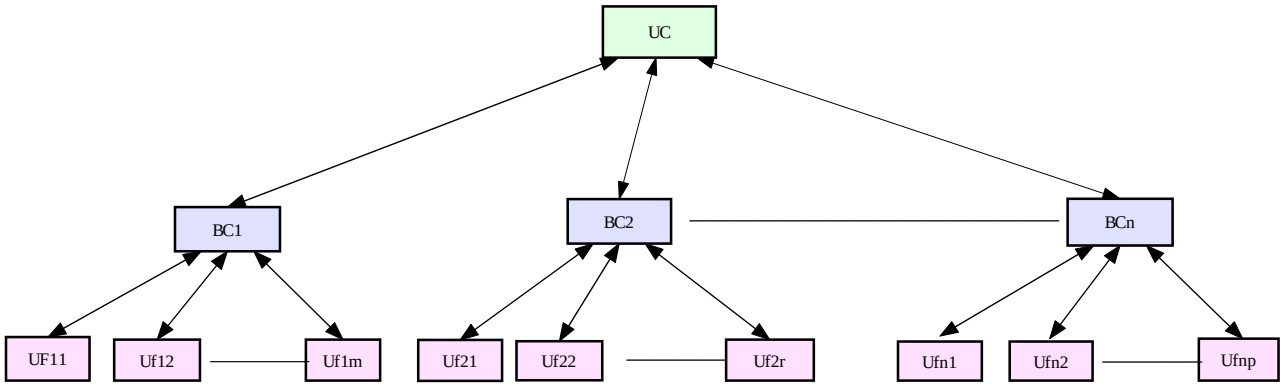
Unitatea de control are rolul de supraveghere a activitatii unitatii functionale legata la ea si de a asigura schimbul de informatie intre aceasta si alte unitati functionale ale sistemului iar unitatea functionala proriu-zisa este cea care realizeaza efectiv functia solicitata (de tipul f1÷f4)
Functionare.
Daca un subsistem functional este solicitat pentru realizarea unei functii de tipul f1 ÷ f4 , cererea soseste la unitatea de control. Unitatea de control va analiza cererea si va activa unitatea functionala pe care o controleaza.
Dupa activarea unitatii functionale, unitatea de control devine libera pentru a putea asigura in continuare conexiuni cu alte subsisteme in timp ce unitatea functionala atasata ei executa sarcina solicitata. Se constata ca in aceasta situatie unitatea de control are o activitate redusa in raport cu activitate unitatii functionale atasate.
Din aceste motive se propune o noua structura a subsistemelor functionale in care se ataseaza la unitatea de control mai multe unitati functionale relativ la una si aceiasi functie fi de tipul f1÷f4.
Fiecare din aceste unitati functionale poate lucra independent, deci mai multe unitati functionale pot lucra in paralel sub controlul unei aceleiasi unitati de control.

In acest fel se asigura subsitemului functional doua proprietati funadamentale:

1. Posibilitatea lucrului paralel. Un subsistem poate realiza una sau mai multe functii de acelasi tip in acelasi timp.
2. Modularitatea subsistemului.Un subsistem poate sa aiba una sau mai multe unitati functionale care lucreaza sub controlul aceleiasi unitati de control.



Aceasta structura se extinde pentru intregul Sistem Calculator Hardware.



Proprietatile de modularitate si paralelism ale subsistemelor functionale se extind la nivelul intregului sistem calculator.

Structura functionala a arborelui care descrie sistemul calculator hard este determinata de urmatoarele conditii:

UC este unitatea centrala a sistemului calculator iar BC_i = 1,2, ...,n sunt subunitatile ei functionale.

Daca BC_i este o radacina in arborele din structura atunci nodurile de pe primul nivel al acestei radacini sunt unitati functionale in raport cu ea.

Fiecare radacina din arborele de structura al sistemului calculator hard determina un nivel care are proprietatile de modularitate si paralelism in raport cu radacina respectiva.

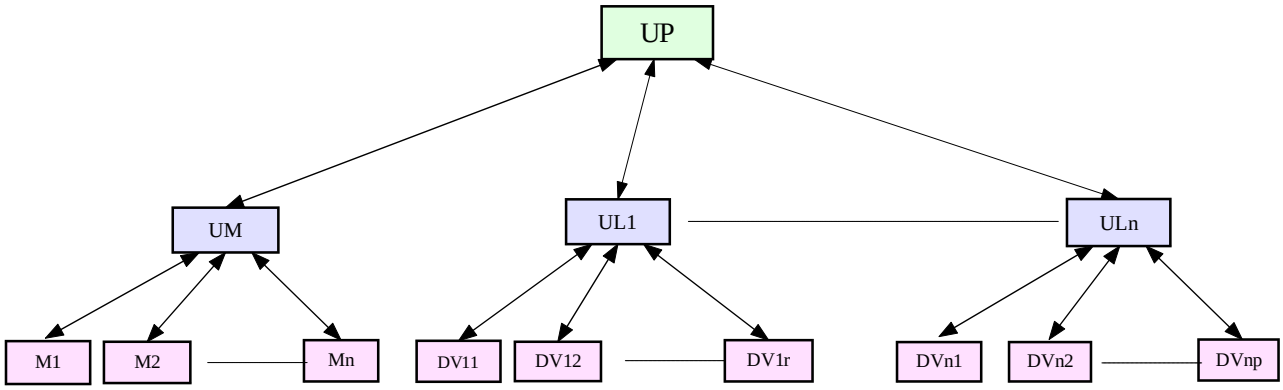
Relatiile dintre nivelele structurii sistemului calculator hard sunt determinate de urmatoarele conditii:

1. Fiecare radacina lanseaza si supervizeaza activitatea tuturor unitatilor functionale de pe primul ei nivel.
2. Radacina poate intrerupe activitatea oricarei unitati functionale asociate ei, daca este nevoie,
3. Fiecare nod de pe un nivel dat, care reprezinta unitatile functionale ale unei radacini, poate cere intreruperea lucrului radacinii pentru a trata conditiile limita in care a ajuns unitatea

respectiva.

4. O radacina la solicitatrea unei unitati functionale apartinind ei (cerere de intrerupere) analizeaza pe rind dupa o regula stabilita toate cererile care sosesc la ea de la unitatile ei in subordine.

Pentru cazul calculatoarelor reale clasele de unitati functionale u_1, u_2, u_3, restring structura arborelui sistemului calculator hardware la urmatoarea structura:

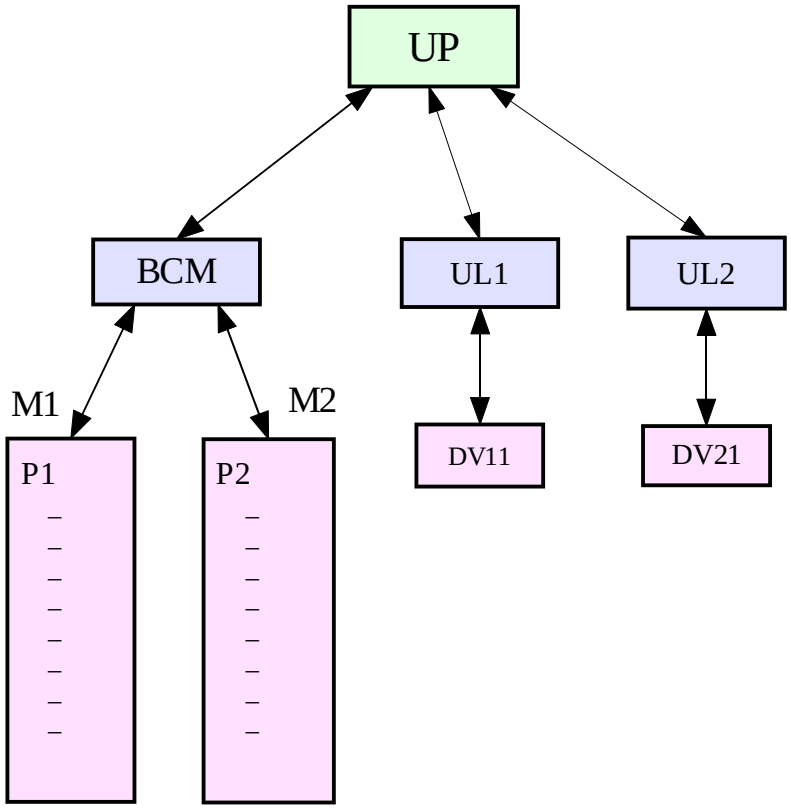


UP	Multimea unitatilor de prelucrare
UL_i	Unitati de legatura (controller) periferice
UM	Controller de memorie
M_i	Blocuri de memorie
DV_{ij}	Dispozitive Periferice

3. **Fluxul informatiei in sistemul calculator**

Vom descrie fluxul informatiei in sistemul calculator particularizat pentru realizarea unor functii date.

Vom presupune ca Sistemul Calculator are o structura care cuprinde o singura UP, doua blocuri de memorie, si doua unitati de legatura cu cite un periferic.



UP are rol de unitate functionala de prelucrare si bloc de control pentru celelalte unitati functionale de pe urmatorul nivel.
Presupunem ca se gasesc stocate in memorie doua programe P1 si P2. Aceste programe contin atat instructiuni ce prelucreaza informatie cu ajutorul UP cat si instructiuni (operatii) de schimb cu dispozitive periferice (operatii de intrare /iesire - I/O).
Presupunem ca P1 se afla incarcat in blocul de memorie M1 si are nevoie sa lucreze cu dispozitivul periferic DV11 adica P1(M1 , DV1)
iar P2 se afla in blocul de memorie M2 si solicita schimb de informatie cu DV2 adica P2(M2, DV2).
Observatii
1. Transferurile de informatii intre dispozitivele periferice si memorie au o viteza mult mai mica decat transferurile intre memorie si UP in procesul de prelucrare.
2. Sistemul accepta lucrul paralel, adica in timp ce UP va prelucra informatia pentru programele P1 si P2 in acelasi timp in care DV1 si DV2 vor executa transferuri de informatie.

Cum se petrec lucrurile?

UP preia din memorie o instructiune (numita si comanda) si o analizeaza. Aceasta poate fi o instructiune de prelucrare sau de schimb de informatie cu un periferic (I/O).
Fluxul informatiei in acest caz:

P1→M1→BCM→UP

Daca este o instructiune de prelucrare la nivelul UP (aritmetica, logica, etc), UP preia operanzii (din memorie sau registrii) executa operatia si depune rezultatul la adresa de destinatie (memorie, registri). UP→BCM→M1
Daca este o instructiune de transfer, UP va transfera sarcina executiei acesteia UL. Pentru aceasta UP va transmite catre UL1 urmatoarele elemente relative la P1:

a. Numele programului care a cerut un schimb de informatiei (adica P1);

b. Adresa de memorie unde (de unde) se face schimbul;
c. Lungimea (cantitatea de informatie ce se transfera);
d. Perifericul prin care se solicita schimbul;
e. Functia de transfer care trebuie executata (citire, scriere, etc,).

Dupa ce aceste elemente au fost transmise UL1 (bloc de control pentru DV1), UP devine diponibila sa execute alte sarcini. De ex. UP va putea sa-si continua activitatea cu executia instructiunilor altui program de ex. programului P2.

In momentul cind UL1 a primit toate informatiile necesare lansarii si derularii schimbului de informatie, se va initia un dialog intre UL1 si dispozitivul periferic solicitat (de ex. DV1). Scopul acestui dialog (test) este de a se afla daca:

DV1 este conectat la sistem?
in caz afirmativ, DV1 poate executa functia solicitata?
in caz afirmativ DV1 este liber?
in caz afirmativ DV1 este gata de lucru?

Daca raspunsul la toate aceste teste este afirmativ atunci UL1 care este o unitate de control pentru DV1 va transmite acestuia elementele necesare desfasurarii schimbului:

a. adresa de memorie unde (de unde) se face schimbul;
b. lungimea (cantitatea de informatie) ce trebuie transferata;
c. functia de transfer ce urmeaza a fi executata (citire, scriere, etc.) precum si conditiile de desfasurare a transferului.

In acest moment UL1 devine la randul ei libera pentru a executa alte functii, in timp ce DV1 va executa efectiv schimbul de informatii direct cu blocul de memorie M1.

Fluxul informatiei pana la initierea transferuluii propriu-zis este urmatorul: UP→UL1→DV1
iar in momentul transferului fluxul este: DV1→M1
sau mai corect: DV1→UL1→BCM→M1.

In acest timpUP executa instructiuni ale programului P2 (de ex.) care la randul sau poate sa solicite printr-o instructiune, un schimb de informatie cu perifericul DV2 (de ex.) . Acest schimb se initiaza si se desfasoara similar cu schimbul prin DV1 descris anterior.

In cazul general al fluxului de informatie intr-un sistem calculator poate fi mult mai complex deoarece:

una sau mai multe linii de comunicatie nu este (sunt) libera/e;
se poate solicita activarea de catre alte subsisteme, unui acelasi subsistem in acelasi timp (cereri simultane);
se solicita activarea unor subsisteme care fie nu sunt prezente, fie nu sunt operabile in momentul solicitarii.

Asigurarea fluxului informatiei in sistemul calculator in functie de evenimentele mentionate anterior va fi asigurata de un nou sistem calculator implementat pe sistemul calculator hardware si anume de catre ***sistemul de operare.***

Regulile generale ale desfasurarii traficului de informatie in sistemul calculator pe care se bazeaza constructia Sistemului de Operare la acest nivel sunt:

1. Cererea de realizare a unei functii se face de jos in sus in arborele de structura al sistemului calculator hard.
2. Intre subsistemele de pe acelasi nivel exista in raport cu radacina, o ordine predeterminata de analizare a cererilor la nivelul superior (prioritate hard).
3. Fiecare subsistem functional se adreseaza blocului sau de control in mod direct.
4. Raspunsurile la cereri sunt transmise in ordinea sosirii lor sau daca sosesc in acelasi timp in ordinea de prioritate hard.

4. Descrierea functionala a unitatii de prelucrare

Unitatea de prelucrare este acea componenta a sistemului care asigura executatrea operatiilor primitive ale sistemului calculator si are si rol de **bloc de control** asupra nodurilor de pe urmatorul nivel in arborele de structura.

Operatiile primitive ale sistemului calculator sunt relativ la tipurile de date si structurile de informatie care se pot defini in sistemul calculator la acest nivel, adica relativ la tipurile de date si structurile de informatie definite pe memoria M a calculatorului. Din aceste motive UP este structurata in functie de tipul de proces necesar pentru executarea efectiva a unei operatii.

Executarea efectiva a unei operatii implica:

- precizarea operanzilor;
- testarea validitatii operanzilor conform operatiei cerute;
- executarea propriu-zisa a operatiei.

Unitatea de prelucrare este compusa din urmatoarele elemente:

1. *Operatorii unitatii de prelucrare*;
2. *Subunitatile de comunicatie ale unitatii de prelucrare*;
3. *Subunitatile de test ale unitatilor de prelucrare*;

Si avand in vedere rolul de bloc de control al UP:

4. *Subunitati de intrerupere ale unitatii de prelucrare*.

Deorece UP este si unitate de control al intregului sistem calculator ea se mai numeste si unitatea centrala.

4.1. Operatorii unitatii de prelucrare (unitatii centrale).

Sunt dispozitive fizice care au rolul de a executa operatiile. Fizic sunt construite din circuite electronice (circuite logice) care executa o functie sau o clasa de functii. Conform tipului de operatii primitive care pot fi executate de sistemul calculator vom deosebi urmatoarele tipuri de operatori:

Operatorul binar (OB): executa operatii primitive asupra tipurilor de date sau structurilor de informatie date sub forma de reprezentare binara (fixa) a marimilor in memoria calculatorului.

Operatorul flotant (numit si Virgula Mobila) (OF): executa operatii primitive asupra tipurilor de date sau structurilor de informatie date sub forma de reprezentare flotanta (Virgula Mobila) a marimilor in memoria calculatorului.

Operatorul zecimal (OZ) executa operatii asupra tipurilor de date sau structurilor de informatie date sub forma de reprezentare zecimala a marimilor in memoria calculatorului.

Operatorii unitatii centrale pot lucra independent sau din motive economice (reduceri de costuri) pot utiliza in comun anumite dispozitive (de ex. : subunitatile de comunicatie si de test).

Atunci cind sunt independenti operatorii UC pot lucra in paralel. Operatorii unitatii centrale au o structura modulara ceea ce inseamna ca pot exista UC care sa nu aiba toate tipurile de operatori. Deci un operator al UC poate fi prezent sau poate sa lipseasca, dar pentru ca Sistemul Calculator sa functioneze trebuie ca UC sa aiba cel putin un operator.

Operatorul binar nu poate sa lipseasca niciodata.

In cazul in care un operator al UC lipseste (de ex. OF) asta nu inseamna ca sistemul calculator nu mai poate executa operatii asupra tipurilor de date sau structurilor de informatie date sub forma de reprezentare corespunzatoare operatorului absent (in ex. nostru tipuri de date sau structuri de informatie in reprezentare flotanta)

Executarea operatiilor asupra tipurilor da date sau structurilor de informatie date sub forma de

reprezentare corespunzatoare operatorului absent se face prin extinderea sistemului calculator.
Aceasta extindere a sistemului calculator se poate face prin:

microprograme: implementarea operatiilor primitive ale operatorului absent ca microprograme construite cu operatiile puse la dispozitie de operatorii prezenti la nivel hardware. De obicei microprogramele sunt realizate cu instructiuni binare deoarece operatorul binar nu poate lipsi niciodata.

subprograme (sau macrouri) la nivelele superioare ale sistemului calculator (de obicei prin bibliotecile asociate mediilor de programare sau Sistemului de Operare) similar ca si la nivelul hard prin operatiile puse la dispozitie de OB.

Prin functia unui operator intelegem multimea operatiilor primitive diferite care pot fi executate de operatorul respectiv.

De obicei operatiile primitive se clasifica in :

1. operatiii aritmetice;
2. operatii logice;
3. operatii salt (conditionat, neconditionat, apel subprograme)
4. operatii de schimb de informatii intre diferitele subsisteme (cu memoria, registrii, dispozitive periferice, etc.)
5. operatii asupra starii programului.

Multimea tuturor operatiilor primitive care pot fi executate de operatorii unitatii centrale este denumita ***multimea operatiilor integrate sau cablate*** *a sistemului calculator si o vom nota cu* Ù***it.***

O operatie cablata (integrata) notata **ø, ø** Ù**it** , actioneaza asupra tipurilor de date si structurilor de informatie definite pe memoria sistemului calculator hardware si este determinata de urmatoarele date:

1. Adresele operanzilor notate **AOi**, i = 1,2,3....
2. Adresa rezultatului **AR**
3. Simbolul operatiei **ø** notat cu **F**. Numarul de operanzi (n-aritatea) este standard si determinat de simbolul F.

O operatie (binara) este definita de o regula:

F: RC1 x RC2 ==> RC3 , unde RCi sunt registrii de comunicatie ai UC.

Din punct de vedere al utilizatorului sistemului calculator, o operatie binara F este definita de o regula notata :

AR: = F(AO1, AO2) ; unde

AR, AO1, AO2 sunt adrese de memorie sau registrii care contin o forma codificata a operanzilor.

Aceasta regula este reprezentata in memorie sub forma unei structuri numite " cuvant " sau " cuvant instructiune " construit din elementele : AR, AO1, AO2 si F , prin concatenarea lor intr-o ordine determinata. In functie de modul cum se obtin adresele operanzilor (tipuri de adresare) in concatenarea care formeaza cuvantul instructiune pot aparea: numere de registrii de baza, index, indicatori de tip de adresare, etc.

Executia operatiei se petrece astfel:

1. Se selecteaza continutul adreselor AO1, AO2 si se transmit in registrii de comunicatie ai operatorului respectiv RC1 si RC2;
2. Se executa functia F: RC1 x RC2 ==> RC3; unde RC3 este un alt registru de comunicatie al operatorului respectiv;
3. Se transfera continutul RC3 (rezultat) la adresa AR.

Un program este format din mai multe instructiuni care se depun in memorie in momentul in care se doreste executarea lui. Deci imaginea unui program in memorie este formata din " cuvinte " (sau " cuvinte instructiune ") care se executa secvential de catre UC.

4.2 Subunitati de comunicatie ale UC.

Legatura intre operatiile primitive ale sistemului calculator si unitatile sale functionale se face prin intermediul registrilor de comunicatie. Sau altfel spus, in timpul executiei operatiilor primitive transmiterea diverselor informatii intre subsistemele care participa la executia operatiilor se face prin intermediul subunitatilor de comunicatie ale UC.

Vom prezenta in continuare din punct de vedere functional cei mai semnificativi registrii de comunicatie:

1.**Registru P** - registrul program sau registrul " counter program ". El contine intotdeauna o adresa de instructiune si anume *adresa urmatoarei instructiuni ce trebuie executata*. Registrul P faciliteaza transmiterea secventiala a instructiunilor catre UC, continand intotdeauna adresa urmatoarei informatii de transmis pentru executie in UC. Deci UC poate ajunge la continutul unei instructiuni si o poate executa prin intermediul registrului P.

2.**Registrii de adresa A**. Au rolul de a memora adresele operanzilor unei operatii. Numarul lor depinde de "n"-aritatile operatiilor primitive.

3.**Registrii de date D**. Au rolul de a memora continutul adreselor operanzilor operatiilor, adica memoreaza chiar operanzii.

Procesul de executie al unei instructiuni mai poate fi descris si astfel:

1. (P) ” DAN
2. (P): = (P)+K
3. DAN ” F, AO1,AO2,AR
4. AO1, AO2, AR ” DCA
5. DCA ” AO1R, AO2R, ARR
6. (AO1R) ” D1
7. (AO2R) ” D2
8. F(D1,D2) ” ARR

unde DAN dispozitiv de analiza
DCA dipozitiv de calcul al adreselor reale
AO11R, AO2R,ARR adrese reale ale operanzilor.
K reprezinta o constanta caracteristica sistemului si este egala cu lungimea cuvintului instructiune.

Din aceasta schema putem vedea mai clar ce inseamna activarea unui program.

Lansarea in executie a unui program la nivelul Sistemului Calculator Hardware inseamna de fapt incarcarea registrului P cu adresa primei instructiuni a programului. Acest lucru se face cu ajutorul unei instructiuni speciale. In continuare executia programului se autogestioneaza.

4.Registrii generali Rgi, i = 1,2,.....

Ei stabilesc o interfata intre unitatea centrala si utilizator. Ei au aceiasi structura ca si cuvintul instructiune si sunt socotiti adrese de memorie cu proprietatea ca unitatea centrala are o mare viteza de acces la ei.

5.**Cuvantul de stare al programului PSW (CSP)**
Asigura interfata sistemului de operare cu programul considerat ca element al limbajului intern. El este caracteristic pentru fiecare tip de arhitectura de UC. De obicei este format din mai multe elemente dispersate in diferite subunitati ale UC.
Relativ la program el reprezinta o structura de informatie bine precizata asupra careia se pot face operatii. Operatiile care se executa asupra PSW se pot executa numai intr-un mod de lucru al UC special numit mod de lucru Kernel (se mai numeste Supervizor sau Privilegiat). Modul Kernel este modul de lucru in care lucreaza Sistemului de Operare (SO).
Fiecarui program in calculator ii corespund un PSW prin intermediul caruia SO face legatura intre programe si sistemul calculator.

Elementele care compun PSW:

Contorul de Program (IP, P, PC);
trigherii (bistabilii) care sunt incarcati la executia unor operatii pentru a indica starea operatiei executate (Flags-uri sau Indicatori de Stare);
masti de intrerupere, moduri de lucru ale UC, etc.

4.2.Subunitati de test ale UC.

UC-ul unui sistem calculator este astfel constituit incat sa poata verifica validitatea oricarei operatii ce urmeaza a fi executate. Fara aceasta verificare s-ar putea ajunge in situatia in care executarea anumitor operatii ar putea conduce la pierderi de informatii.
Activitatea in sistemul calculatorului trebuie sa nu conduca la distrugeri accidentale sau acces neavizat la informatie.
Subunitatile de test sunt necesare pentru toate subsistemele cu rol de bloc (unitate) de control din Sistemul Calculator (UC, UM, US).
Din punct de vedere functional subunitatile de test (SuT) executa o succesiune de operatii logice asupra elementelor componente ale unei instructiuni (comenzi).
In cazul UC se verifica atat operanzii cat si operatia solicitata:
Testul operatiei.
Prin acest test se verifica :*"operatia solicitata apartine sau nu operatorilor UC ? "*si daca da, *"operatia poate fi lansata in executie?"* .
In cazul in care rezultatul acestor teste este afirmativ, se va trece la executia "testului operanzilor".
In caz contrar UC va semnala aceste conditii exceptionale (evenimente) prin intermediul mecanismului (sistemului) de intreruperi .
In cazul aparitiei unei intreruperi se va transfera activitatea catre Sistemul de Operare care este in acest caz utilizatorul Sistemului Calcualtor Hard . In acest fel SO va fi cel care va decide in continuare strategia care va trebuie urmata.
Testul operanzilor.
Se verifica: *"operanzii apartin activitatii (programului) care a solicitat executia instuctiunii ?"* si daca da, " *operatia este definita asupra operanzilor ?"*.
In cazul in care aceste teste sunt pozitive se trece la executia efectiva a operatiei (instructiunii) respective.
In caz contrar ca si in cazul "testului operatiei" se declansaza o intrerupere care transfera activitatea catreSO Acesta va decide asupra strategiei urmatoare.
In cazul UM considerate ca bloc (unitate) de control pentru blocurile de memorie Subunitatile de Test au rolul de a proteja memoria impotriva *distrugerilor accidentale de date* sau *accesului neavizat la informatie*.

In cazul Unitatilor de Schimb de informatii cu perifericele lucrurilor stau similar in sensul ca Subunitatile de Test au rolul de a semnala printr-o intrerupere toate conditiile care ar putea influenta negativ evolutia umatoare a activitatii Sistemului Calculator.

4.3.Subunitati de intrerupere

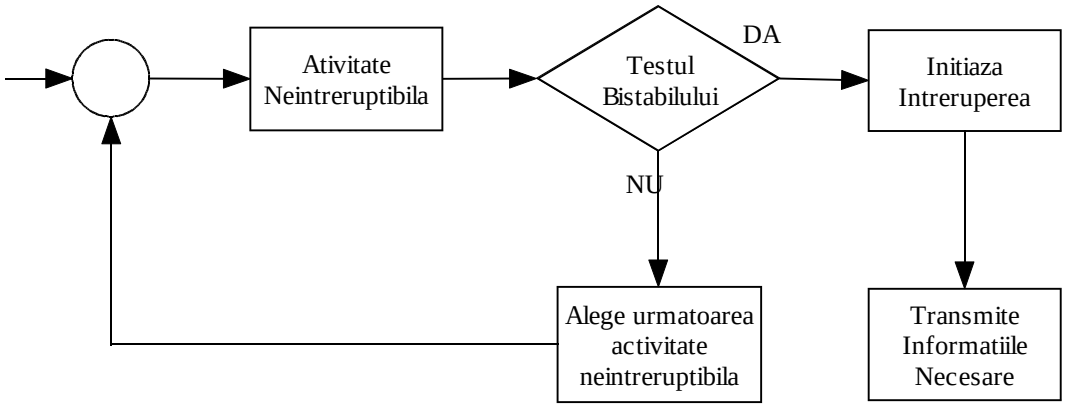
Sunt un mijloc de comunicatie intre Sistemul Calculator Hardware si utilizatorul sau Sistemul de Operare. Exista doua tipuri de Subunitati de Intrerupere:

Subunitati de intrerupere (SuInt) care trimit semnale (cereri) de intreruperi a lucrului unui subsistem " SuInt pentru Initiere.

SuInt care receptioneaza semnalele de intrerupere transmise de alte subsisteme " SuInt pentru Receptionare.

Intr-un subsistem functional in functie de pozitia sa in arborele de structura pot fi ambele prezente. De exemplu in UC sunt prezente ambele tipuri de Subunitati de Intrerupere.

Activitatea unui subsistem al Sistemului Calculator Hardware este formata dintr-o succesiunea de activitati neintreruptibile. Dupa executia oricarei activitati neintreruptibile are loc testarea starii bistabilului (Triggerului) de intrerupere.



Atunci cand UC doreste sa intrerupa lucrul unui alt subsistem care lucreaza sub controlul ei, va comuta bistabilul de intreruperi corespunzator acestui subsitem.

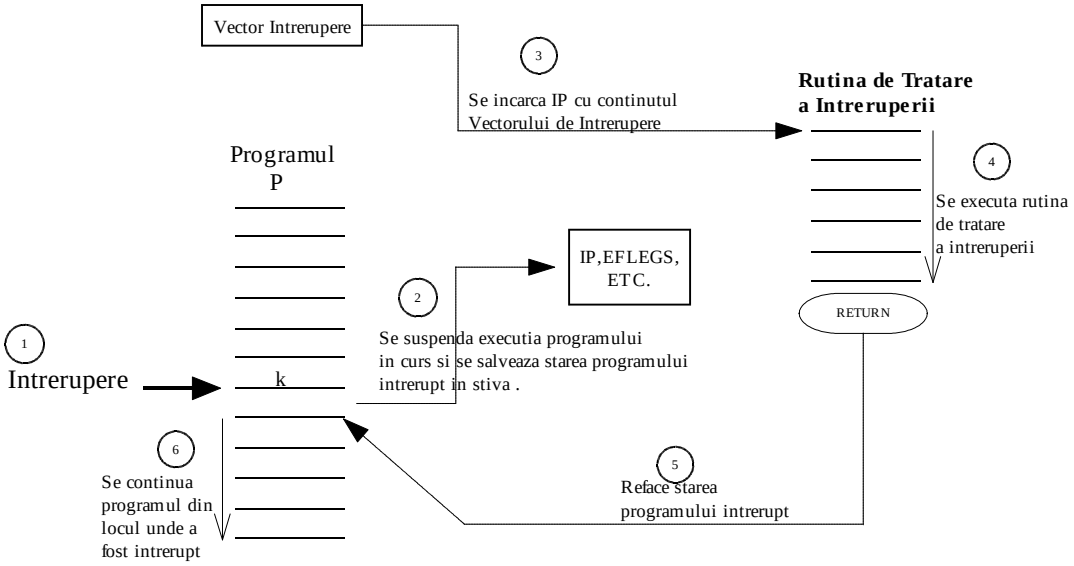
SuI pentru Receptionarea intreruperilor sunt ceva mai complicate (cel putin in cazul UC) intrucat ele trebuie sa execute diverse teste asupra semnalelor receptionate. Vom discuta in continuare SuInt pentru Receptionare la nivelul UC. Pentru celelalte subsisteme (BCM, UL) ele sunt mai simple si nu implica decat mecanisme hardware.

Intreruperile receptionate de UC sunt impartite in clase (tipuri) de intrerupere. Aceste tipuri sunt determinate de locul de unde provin aceste intreruperi. In cadrul unei clase intreruperile sunt clasificate in functie de prioritatea cu care vor intra in actentia UC si determina asa numitele **nivele de intrerupere**. Fiecare nivel de intrerupere corespunde unui eveniment aparut in activitatea SC

Activitatile UC sunt succesiunea de activitati neintreruptibile intre care se face analiza si tratarea intreruperilor.

Prin constructia sistemului calculator fiecarui nivel de intrerupere i se asociaza o adresa de memorie specifica numita **vector de intrerupere** , care reprezinta punctul de intrare in SO relativ la evenimentul care a determinat aparitia intreruperi respective. La aparitia unei intreruperi UC isi intrerupe lucrul obisnuit, salveaza (memoreaza) starea masinii in momentul aparitiei intreruperii

apoi se pregateste sa reia executia de la adresa specificata in vectorul de intrerupere. La aceasta adresa se afla o rutina (subprogram) al SO Deci la aparitia unei intreruperi se continua activitatea UC cu un program al SO care "trateaza" intreruperea respectiva permitand continuarea activitatii Sistemului Calculator in conformitate cu strategia corespunzatoare aparitiei intreruperii respective.
Altfel spus *in momentul aparitiei unei intreruperi Sistemul de oOperare (SO) pe baza starii actuale a SC determina urmatoarea stare functionala in asa fel incat exploatarea SC in ansamblu sa fie cat mai eficienta.*



Intrerupere este un mecanism care se produce la nivelul dispozitivelor fizice. Exista evenimente similare cu intreruperile care se produc la nivelul programului in curs de executie. Aceste evenimente datorate executiei unei anume instructiuni din programul in ececutie sunt numite exceptii program.
Exceptiile program sunt un mijloc de autointrerupere a unui program pentru a solicita executarea unor functii de catre SO Deasemeni exceptiile program se produc si in cazul in care apar evenimente nedorite (erori) in derularea unui program ceea ce necesita interventia SO
Daca am considera programele ca dispozitive fizice atunci " exceptiile program " ar deveni " intreruperi" adica *"exceptia program" este o intrerupere a masinii abstracte reprezentate prin programul care se executa pe un sistem calculator real.*
De exemplu prin sistemele calculator bazate pe arhitectura INTEL " exceptiile program " sunt de 3 tipuri:

erori de program (atunci cand se detecteaza o eroare in program);
generate de software (voluntare);
verificarea masinii (atunci cand se detecteaza o eroare a hardului) - " erori detectate de " machine - check ".

5. Descrierea functionala a memoriei

Memoria serveste pentru realizarea functiei de conservare a informatiei. Ea poate fi considerata ca un intermediar intre celelalte subsisteme functionale (fluxul informatiei in SC se face prin intermediul memoriei).
Memoria contine informatie ce trebuie prelucrata, in memorie se depun rezultatele, operatiile de I/O se desfasoara intre dispozitivele periferice si memorie.
Din punct de vedere fizic subsistemul memorie este construit dintr-o succesiune finita si ordonata de elemente bistabile. Cele doua pozitii ale unui astfel de element se transforma in purtator de informatie , si anume in suportul cifrelor sistemului de numeratie in baza 2. In succesiunea de bistabile care compun memoria vom defini o descompunere in clase numite grupe, astfel ca fiecare grupa de astfel de bistabile sa capete o unica identificare in subsistemeul memorie. Dimensiunea acestor grupe se stabileste la constructia SC tinand cont atat de numarul liniilor de comunicatie dintre memorie si alte subsisteme (considerente constructive) cat si de numarul de litere ale alfabetului care trebuie codificat. Dimensiunea acestor grupe (locatii) determina asa numita "unitate de adresare a memoriei". Grupele sunt ordonate si numerotate cu ajutorul numerelor naturale. Cand ne referim la o grupa ne referim la numarul ei de ordine. Acesta se numeste "adresa de memorie".
Numarul de ordine sau adresa de memorie determina pozitia fizica a unei locatii in cadrul subsistemului de memorie. Unitatea de adresare a memorei reprezinta cantitatea minima de informatie care poate fi manipulata (referita) in cadrul unui transfer de informatie cu subsistemul memorie. Cantitatea de informatie care se transfera printr-o singura comanda intre subsistemul memorie si alte subsisteme ale SC se numeste unitate de transfer a memeoriei.
Unitatea de adresare poate su nu fie egala cu unitatea de transfer a memoriei si este de obicei mai mica.
Datorita proprietatii de paralelism din SC subsistemul memorie trebuie sa satisfaca cateva proprietati:

a. este astfel organizat incat *permite lucrul paralel* a mai multe programe.

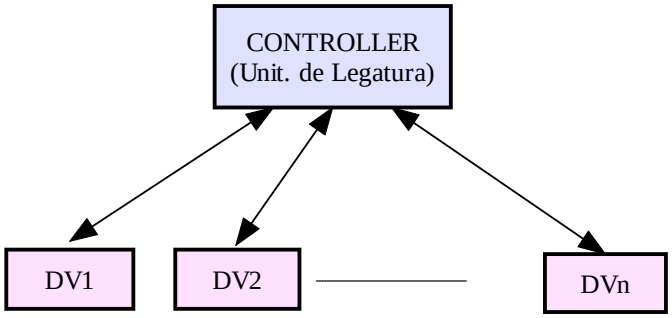
b. trebuie sa permita *alocarea dinamica* a locatiilor ei pentru diferite programe care exista la un moment dat in SC

c. sa permita *extensia modulara* .

Memoria sistemului calculator (numita si memoria operativa) trebuie privita din doua puncte de vedere, din punct de vedere al SC in sine si din punct de vedere al utilizatorului SC
Memoria ca entitate fizica in sistemul calculator se numeste **memorie reala** , iar memoria ca entitate din punct de vedere al utilizatorului va fi numita **memorie virtuala**.
Gradul de deosebire dintre memoria reala si memoria virtuala a unui SC este o masura a gradului de complexitate a organizarii memoriei. Mecanismul prin care se traduce memoria virtuala in memorie reala se numeste **mecanism de adresare** .
Mecanismul de adresare este conceput in asa fel incat sa se asocieze in mod univoc memoria virtuala folosita la scrierea programului cu unul sau mai multe blocuri de memorare fizica. Din modul de lucru a acestui mecanism rezulta diferite moduri de adresare a memoriei.

6. **Descrierea functionala a unitatilor de legatura (controllere) si a dispozitivelor periferice.**

Unitatile de legatura si dispozitivele periferice reprezinta clasa de subunitati functionale care are rolul de a realiza functiile de receptionare si transmitere de informatii in SC.
Sistemul calculator vine in contact cu lumea exterioara prin aceasta clasa de subunitati functionale. Asigura interfata dintre sistemul calculator si lumea externa avand in vedere numai o anumita parte a acesteia, organizata conform scopului lor. Structura informatiei acceptata de SC prin subunitatile de schimb de informatie este una particulara si specifica. Subunitatile functionale care pot receptiona sau transmite informatia astfel structurata se numesc **dispozitive periferice.**
Subunitatile care realizeaza receptionarea respectiv transmiterea informatiei in si din sistemul calculator au o structura cunoscuta (discuatata) deja. Aceasta structura contine un bloc de control (controller sau unitate de legatura) la care sunt atasate mai multe unitati fuctionale (dipozitive periferice) de acelasi tip.



Schimburile de informatie (date) se fac intotdeauna intre DP si UM.
Transferul de informatiei de tip comanda si control se realizeaza intre blocurile de control ale subunitatilor functionale din SC (UC , Controller Memorie si UL).
Unitatea de legatura (controler) este legat de unitatea centrala si unitatea de memorie prin linii de comunicatii.
Controllerul, blocul de control al unei clase de dispozitive periferice lucreaza sub controlul UC si dirijeaza activitatea dispozitivelor periferice atasate ei. Schimbul de informatie cu dispozitivele periferice se realizeaza prin intermediul controllerului.
Ordinul (comanda) de activare a unui dispozitiv periferic se transmite de catre UC.
Printr-un ordin se transmit:

- perifericul implicat in schimb;
- adresa de memorie implicata in schimb (daca este cazul);
- numarul de octeti ce se vor transfera (daca este cazul);
- conditiile de desfasurare a activitatii;
- functia solicitata (receptie, transmisie, pozitionare,etc.)

Aceste informatii sunt stocate in controler (UL) in registrii proprii acestuia (registrii de comanda). Deasemeni UL pastreaza in alti registrii (registrii de stare) starea dispozitivul periferic sau a actiunilor in curs.
Activitatea unui DP nu este intodeauna una de transfer de informatie. Pot fi si activitati care pregatesc un tranfer de informatie, cum ar fi pozitionarea capetelor de citire/scriere la hard disc sau avansul hartiei la imprimanta, etc. Pentru a realiza un transfer propriuzis de informatie trebuie derulate o succesiune de comenzi elementare intr-o ordine precisa. De exemplu citirea unui sector de pe o unitate de disc magnetic inseamna de fapt executarea pe rand a comenzilor de pozitionare , si apoi de citire.

Unitatile de schimb (UL + DP) utilizeaza doua moduri de transmisie propriuzisa a informatiei intre dispozitivele periferice si memorie determinand doua moduri de lucru ale dispozitivelor periferice:

a. modul de lucru registru ” la o comanda se transfera o cantitate de date egala cu lungimea unui registru, informatia fiind preluata efectiv printr-o instructiune de clasica de memorare.

b. modul de lucru DMA (Direct Memory Access)” la executia unei singure comenzi de transfer, se transfera nemijlocit intre memeorie si periferic,o cantitate de date de lungime oarecare si independent de activitatea UC (in paralel).

In modul de lucru registru este evident ca transferarea unei cantitati mai mari de informatie se face prin repetarea aceleiasi comenzi de un numar de ori ceeace in poate fi ineficient.

In ambele moduri de transfer este foarte important modul prin care se poate determina momentul terminarii executiei unei operatii cu un DP.

Momentul terminarii executiei unei comenzi de catre DP poate fi semnalat de catre acesta, fie prin pozitionarea unor indicatori in registrii de stare ai DP respectiv, fie prin declansarea unei intreruperi.

Complexitatea activitatii unitatilor de schimb face ca lucrul direct (nemijlocit) cu DP sa fie foarte laborios si greu de realizat de catre programatori. In plus existenta mai multor utilizatori ai aceluiasi SC face obligatorie existenta unor metode de protejare a informatiei.

Sarcinile acestea sunt preluate de SO.

Controlul si activarea dispozitivelor periferice se face de catre SO. In acest fel se poate realizeaza o gestiune eficienta a DP si asigurand in acelasi timp protectia informatiei impotriva distrugerilor accidentale sau accesului neavizat (nepermis).

Din punct de vedere al SO fiecare DP este considerat ca un program autonom numit<u>proces</u> pentru transferul informatiei intre diferitele parti ale sistemului calculator.

Viteze mica de lucru a DP in comparatie cu viteze altor prelucrari in sistem relativ la memorie si UC, sta la baza prelucrarii paralele a informatiei in SC

SO administreaza activitatea tuturor DP din SC. In functie de caracteristicile fizice ale DP, SO imparte DP in clase:

partajabile ” poate fi folsite in comun de mai multi utilizatori (programe);

fixe ” sunt alocate unui program pe durata de existenta a programului;

Identificarea perifericelor in sistem trebuie sa fie univoc definita si sa indice atat pozitia fizica a perifericuluicat si caracterizarea lui functionala.

Perifericelor li se asociaza un *nume simbolic* prin care perifericuleste cunoscut in lumea externa (de ex. in limbajele de programare). Numelui simbolic ii corespund la un moment dat un anume DP fizic.

Starea unui DP in sistemul calculator este la dispozitia SO sub forma unor tabele gestionate de SO

Trecerea unui DP dintr-o stare in alta este determinata si urmarita de catre SO cu ajutorul sistemului de intreruperi. Evenimentele determinate de terminarea unei functii de transfer, aparitia unei erori etc. sunt semnalate de catre SC catre SO prin aparitia unei intreruperi.

In functie de filozofia SO si de arhitectura SC Hardware organizarea efectiva a activitatii perifericelor este o cracteristica a SO

CAP. II. STRUCTURA SISTEMELOR DE OPERARE

Scopul acestui capitol este de a da o definitie Sitemelor de Operare si de a prezenta structura si principalele concepte ale acestora.

Ce este acela un sistem de operare ?

Pornind de la ierarhia Sistemelor Calculator putem spune ca :

*Un **Sistem de Operare** este o colectie de componente software (programe) care adaugate la Sistemul Calculator Hardware ofera suportul necesar implementatorilor subsistemelor de programare.*

Aceasta definitie poate fi corecta dar ea nu reda explicit complexitatea functiilor indeplinite de Sistemul de Operare pentru a asigura suportul necesar *implementatorilor subsistemelor de programare.* Din aceasta cauza vom fi mai aproape de realitate daca completam aceasta definitie cu enumerarea functiunile globale indeplinite de SO.

Un SO asigura indeplinirea urmatoarelor functii:

un management eficient al resurselor fizice si logice (programe) ale S.C urmarindu-se o incarcare maxima a resurselor fizice ale S.C si minimizarea timpului de raspuns.

protectia informatiei impotriva distrugerii accidentale si accesului neavizat.

o interfata comoda pentru scrierea aplicatiilor - pentru implementarea subsistemelor de programare.

un model abstract pentru periferice - fisierele vor fi considerate ca structuri independente de periferic (se ascund particularitatile dispozitivelor I/O) pentru a se asigura o manipulare uniforma si simpla a datelor indiferent de suportul fizic pe care se afla.

asistarea utilizatorilor printr-un sistem dezvoltat de comunicatie:

utilizator S.O prin sistemul de mesaje si limbajul de comanda;

program - S.O prin apeluri sistem.

1. Concepte ale sistemelor de operare.

1.1. Apeluri sistem.

Apelurile sistem reprezinta mijlocul prin care un program poate solicita executarea unei anumite functii de catre SO. Acestea fac parte din ceeace numim *"sistemul de comunicatie program SO"* .
Apeluri Sistem (System Calls) sunt un set de instructiuni speciale numit set de "*instructiuni extinse"*. Denumirea de *"Instructiunile extinse"* provine din faptul a au aceiasi structura cu Instructiunile Cablate ale SC Hardware dar au coduri de operatii care nu fac parte din multimea operatiilor cablate ale UC ( ). Operanzii acestor instructiuni sunt parametrii asociati functiei solicitate . Executia lor va declansa o intrerupere sincrona (exceptie program) deoarece codul lor nu este cunoscut de SC Hardware. Acest lucru va determina transferul activitatii SC catre SO si executia rutinei SO de tratare a iacestui tip de intrerupere, numita si "handler de intreruperi". Rutina de tratare analizeaza instructiunea care a declansat intreruperea (sincrona), si verifica validitatea cererii. Daca cererea este valida si corecta se va lansa modulul SO care executa efectiv functia solicitata. Aceste functii sunt operatii complexe care nu pot fi executate direct de catre programele utilizator. In general aceste " apeluri sistem" creaza, sterg sau utilizeaza diverse resurse administrate de SO (procese, fisiere, etc.)

1.2. Procese.

Este un concept intilnit in toate SO. In general un proces este orice program in executie, ceea ce inseamna programul in forma executabila, datele programului, IPC, PSW si alte registre si orice alte informatii necesare pentru executia programelor.
In multiprogramare, periodic procesele sunt oprite si lansate in executie altele. Acest lucru inseamna ca este necesar ca toate informatiile legate de procesul suspendat trebuie pastrate astfel incit sa fie posibil sa fie repornite la un moment dat exact in locul in care au fost oprite.

De exemplu daca procesul prelucra fisiere la repornirea sa, pointerii de fisiere sa fie exact in aceiasi pozitie din momentul suspendarii sale.

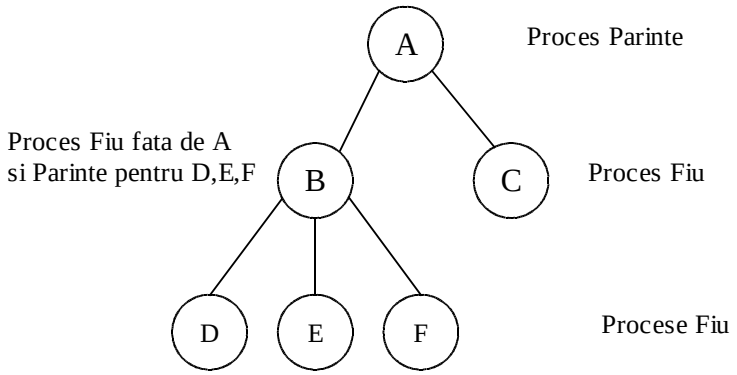
Un proces suspendat consta din:

- spatiul sau adresa (imaginea sa din memorie) si toate informatiile legate de proces. Aceste informatii sunt memorate intr-o tabela a SO (tabela de procese)

Una din functiile importante ale SO este crearea si terminarea proceselor. Acest lucru se realizeaza cu ajutorul " apelurilor sistem"

Ex.: Procesul numit " interpretor de comenzi" sau " Shell" citeste o comanda de la terminal. Daca de exemplu comanda cerea compilarea unui program, atunci procesul " Interpretor de comenzi" trebuie sa creeze un nou proces care va executa compilarea cind procesul "compilare" se va termina, el va executa un "apel sistem" pentru terminarea sa.

Daca un proces poate crea unul sau mai multe procese (fii) care la rindul lor pot crea alte procese se poate ajunge la urmatoarea schema:



Exista o mare varietate de "Apleluri Sistem" . Alte exemple de "Apeluri Sistem":

Apeluri Sistem pentru suplimentarea memoriei pentru un proces;
Apeluri Sistem asteaptarea terminarii unui proces fiu;
Apeluri Sistem pentru suprapunerea programului apelant peste un altul (overlay);
Apeluri Sistem masurarea timpului ; Se contorizeaza scurgerea unei perioade de timp (ceas) o intrerupere la sfirsit;

Comunicarea intre procese pipe
Protectia UID, GID

1.3. Fisiere

Fisierele reprezinta structuri complexe de informatie care au de obicei ca suport de informatie suportul fizic al Dispozitivelor Periferice. Pentru a se putea asigura pe de o parte "securitatea informatiilor" iar pe de alta optimizarea lucrului cu Dispozitivele Periferice, manipularea fisierelor se face prin intermediul SO cu ajutorul "apelurilor sistem".

Particularitatile I/O sunt ascunse;
Model abstract al perifericelor - Fisiere independente de periferic;
Deschiderea fisierelor Read, Write protectia fisierelor;
Directoare, ierarhia directoare
- path name
- root directory
- director de lucru
Fisiere block special
Fisiere caracter- special flux de caractere

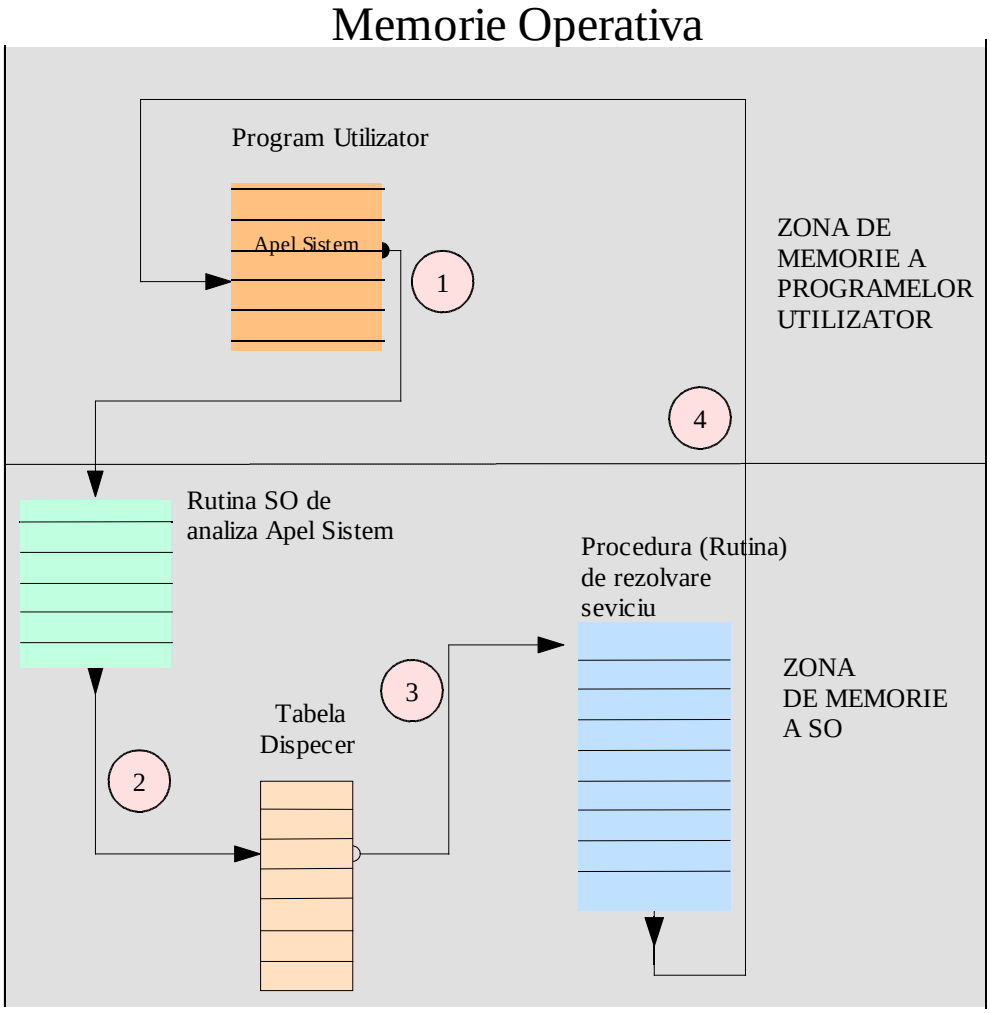
1.4. Shell
Sistem de comunicatie

2. Modele de Sisteme de Operare.

Pina acum am privit SO din afara de ex. d.p.d.v. al interfetei programatorului. Privit in interior SO poate sa aiba structuri diferite.

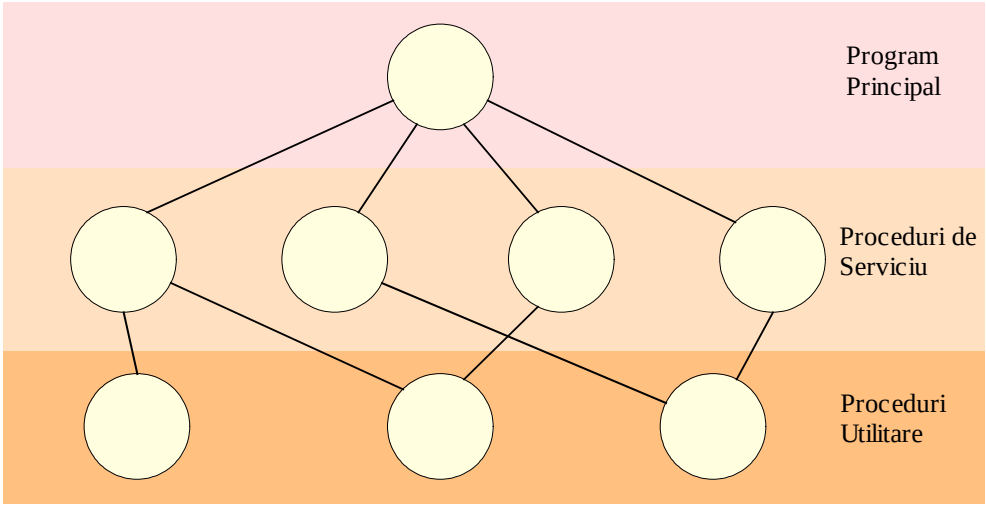
2.1. Sisteme monolitice

Este cea mai banala structura. SO este conceput ca o colectie de proceduri, fiecare din ele poate apela, daca este nevoie, pe oricare alta. Fiecare procedura din acest tip de sistem are o interfata " bine definita" in ceea ce priveste parametrii si rezultatele.
Fiecare procedura este vizibila de oricare alta (in opozitie cu structurile la care procedurile sunt grupate in module sau pachete unde numai punctele de intrare special desemnate sunt " vizibile" din afara modulului)
Se mai numesc si sisteme fara structura sau " This big mess" = marea dezordine (ingramadeala) =
Aceste sisteme furnizeaza o serie de servicii care sunt apelate cu ajutorul " apelurilor sistem" (ca in orice alte SO). Un apel sistem se realizeaza prin plasarea parametrilor in locuri bine precizate (conform conventiilor de apel), registre sau stiva si apoi se executa o " exceptie program", numita aici " apel kernel" sau "apel supervizor". Aceasta instructiune comuta UC din " mod utilizator" (user mode) in " kernel mode" (supervizor mod) transferindu-se controlul SO.
SO examineaza parametrii " apelului " identifica serviciul solicitat si-l executa. Dupa aceasta controlul UC este retransmis programului care a solicitat serviciul.



In acest model pentru fiecare " apel sistem" exista o " procedura de serviciu" care rezolva o anumita functie si eventual o colectie de " proceduri utilitare" de care poate sa aiba nevoie anumite " proceduri de serviciu".
Exemplu de utilitare: extragerea datelor din programele utilizator.

Aceasta organizare sugereaza totusi o structura de baza:



2.2. Sisteme de operare structurate pe nivele.

Sunt o generalizare a structurii anterioare ca o ierarhie de nivele , unul fiind construit pe altul.

Primul SO construit pe acest principu a fost THE (Technische Hogescool Eindhoren Netherlands E.W. Dijkstra 1968)

Sistemul are 6 nivele:

5	Operatorul (utilizatorul) sistem
4	Program Utilizator
3	Gestiunea I/O
2	Comunicatie Operator - Proces
1	Managementul Memoriei
0	Alocarea Procesorului si Multiprogramarea

NIVEL 0 - se ocupa cu alocarea procesului, comutind intre procese atunci cind apare o intrerupere. Furnizeaza bazele functionarii sistemului in regim de *multiprogramare*.

NIVEL 1 managementul (administratoarea) memoriei. Aloca spatiu in UM pentru procese . Administreaza pastrarea " paginilor" de memorie operativa sau intr-o memorie externa atunci cind nu este spatiu suficient in UM.

NIVEL 2 asigura comunicatia intre fiecare proces si operator

NIVEL 3 managementul dispozitivelor periferice (numite si dispozitive de Intrarea/Iesire sau Intput/Output sau I/O) controlind fluxul de informatie la /si de la dispozitivul periferic.

Asigura utilizatorului o imagine abstracta asupra dispozitivelor de I/O, independenta de particularitatile fiecarui dispozitiv de I/O.

NIVEL 4 - Nivelul la care se plaseaza programele utilizator.

NIVEL 5 este nivelul procesului operator sistem.

Aceiasi structura de nivele poate fi considerata ca o succesiune de inele concentrice (sistem MULTICS), unde fiecare inel este mai prioritar (privilegiat) decit cel exterior. Atunci cind o procedura de pe un anumit nivel solicita executia unei functii el foloseste un " apel de procedura" adresat inelului interior, similar " apelurilor sistem".

Avantajul structurii circulare este posibilitatea dezvoltarii cu nivele pentru diverse subsisteme utilizator.

2.3. SO de tip " Masina Virtuala".

Este introdus de IBM si se bazeaza pe observatia ca SO poate fi privit ca o entitate ce furnizeaza doua tipuri de servicii:

multiprogramarea;

o interfata simpla si eficienta pentru utilizator.

Structura sistemului fiind astfel:

Aplicatie User	Aplicatie User	.....	Aplicatie User
VM1	VM2	**.....**	VMn
MONITOR MASINA VIRTUALA			
SISTEM CALCULATOR HARDWARE			

Nivelul " MONITOR MASINA VIRTUALA" realizeaza toate functiile de multiprogramare furnizind suportul pentru mai multe masini virtuale (VMi)

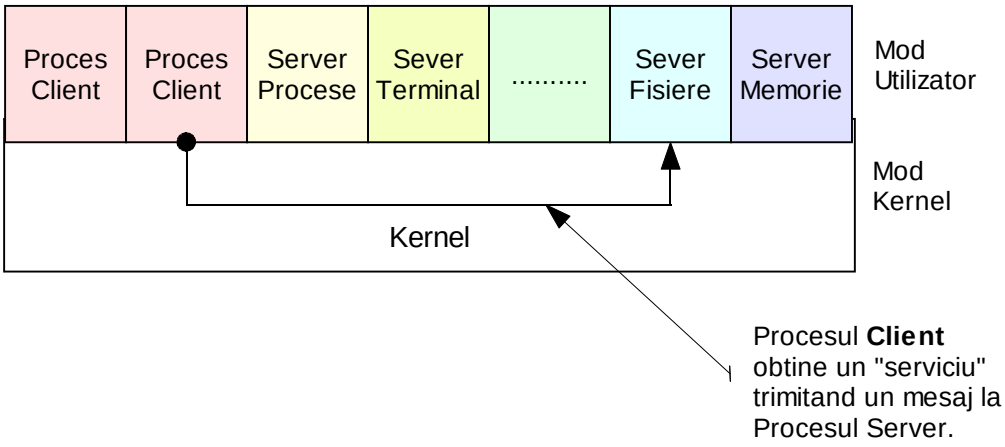
Fiecare masina virtuala (VMi) ofera cite o interfata identica simulind fiecare in parte un S.C ceea ce face posibil ca sistemul de calcul sa fie privit ca mai multe S.C. hardware deci sa fie posibila executia mai multor sisteme de operare.

2.4. Modelul de SO Client- Server

Tendinta S.O actuale este de a se transfera pe cit posibil pe nivele superioare (nivelul programelor utilizator) o parte din codul (programele) S.O astfel incit Kernel-ul (nucleul) sa fie redus ca dimensiune si complexitate.

In sistemele de operare de tip Client-Server o parte din functiuni sunt trecute in procese de tip User (utilizator).

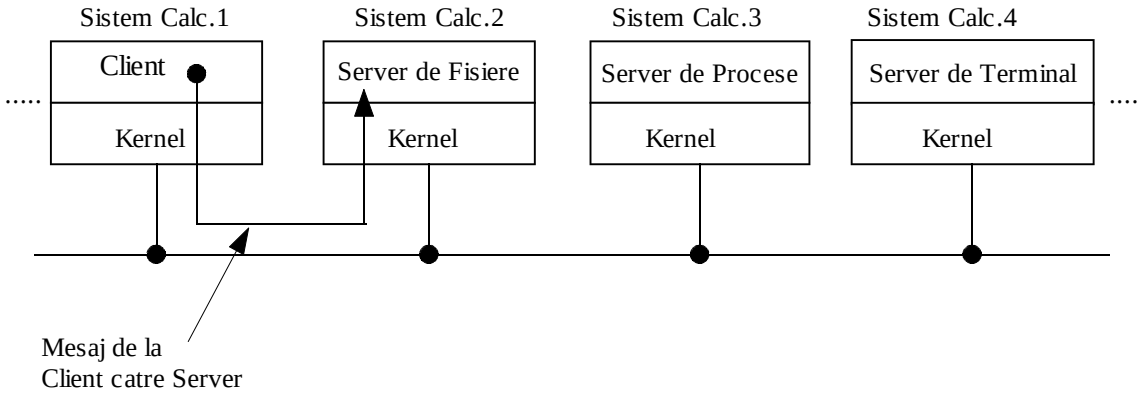
Solicitatrea de executare a unor servicii, de ex.: citirea unui bloc al unui fisier venita din partea unui proces utilizator (proces client) este trimisa la un "proces server" care va executa functia ceruta si va trimite inapoi raspunsul.



In cazul acestui model nucleul SO (kernel) asigura comunicatia intre procesele client si procesele

server. Separind SO in parti componente fiecare din ele administrind numai o parte a sistemului de ex.:fisiere, procese, terminale, memorie, fiecare parte devine mai mica si mai simplu de manipulat. Deoarece procesele server lucreaza in modul de lucru "utilizator" si nu in modul "supervizor" ele nu au acces direct la hardware. In consecinta o eroare in server nu conduce la caderea (blocarea) sistemului in ansamblu ci numai blocarea serviciului respectiv.

Un alt avantaj al acestui model este adaptibilitatea la structurile cu Sisteme Calculator distribuite:



Nucleul SO (Kernel) realizeaza transportul mesajelor de la procesele client catre procesele server si inapoi si o parte din functiunile care nu pot fi executate in " mod user" (de ex. accesul reg. I/O - functiune critica).
Aceasta problema (a functiunilor critice) se poate rezolva in doua moduri:

pastrarea proceselor critice ca procese privilegiate care vor rula in " mod kernel";
impartirea proceselor critice in doua: o parte care ruleaza in " mod kernel" si executa efectiv comenzile asupra dispozitivelor de I/O si o parte care ruleaza in " mod user" si rezolva problemele de administrare,

Impartirea proceselor critice in doua parti, o parte la controlul dispozitivelor periferice si partea de " politica" in administarea unui dispozitiv periferic reprezinta un aspect important al proiectarii S.O.

Tipuri de sisteme de operare.

Initial S.O au fost proiectate pentru a controla un singur sistem calculator. si este la ora actuala cele mai raspindite S.O chiar daca aceste sisteme calculator sunt conectate impreuna in retele de calculatoare.
Sistemele de operare care administreaza un singur calculator se numesc " SO single-procesor" sau " single CPU " sau SO traditionale.
Exista deasemeni SO multiprocesor care adminstreaza sisteme calculator continind unul sau mai multe CPU.
Sistemele de operare care administreaza mai multe S.C conectate in retea se numesc sisteme de operare distribuite.

Sistemele de operare au patru componente importante:

- managementul proceselor
- managementul memoriei
- managementul fisierelor
- managementul dispozitivelor periferice

Resurse fizice si resurse logice

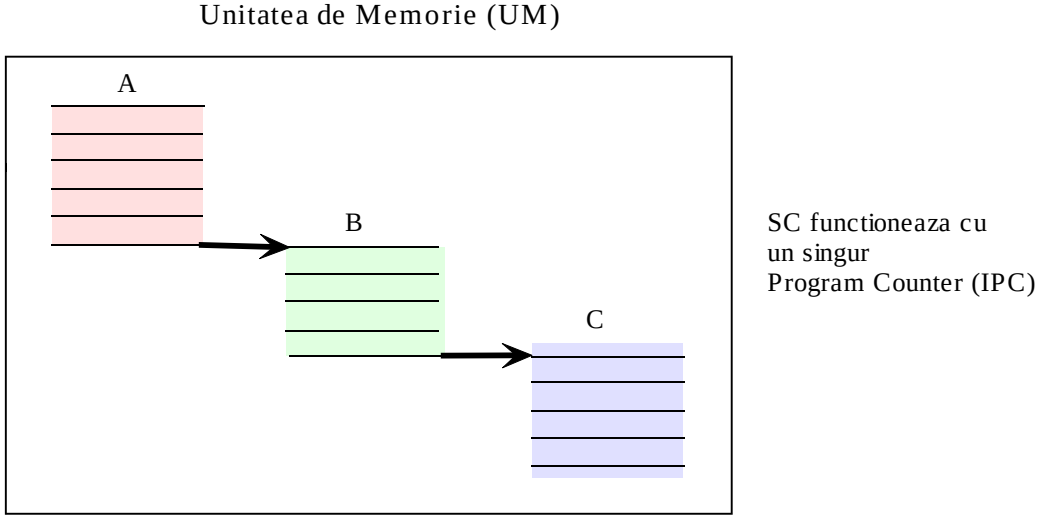
Cap. III PROCESE

3.1. Introducere

Proces: o abstractizare a unui program in executie.
Toate sistemele de calcul moderne executa mai multe actiuni (sarcini, task-uri) aparent in aceleasi timp. Pentru un utilizator care se afla in fata unui sistem el observa ca in timp ce programul sau se executa, sistemul "printeaza (tipareste pe imprimanta)" pentru o alta aplicatie, citeste un CD, etc. Acest mod de lucru al sistemelor se numeste **multiprogramare** . Chiar daca S.C. hard are un singur UC (CPU) , acesta se "comuta" de la un proces la altul, executand fiecare proces in cuante de zeci sau sute de milisecunde. In mod riguros intr-un anumit moment, UC-ul executa un singur proces. Dar de ex. intr-o secunda UC-ul poate sa execute (partial) mai multe procese, creind "iluzia" utilizatorului de executie "simultana" a mai multor procesele. De fapt este un "paralelism" la nivelul UC care comuta controlul rapid de la un proces la altul. Paralelismul real la nivelul SC hard exista numai intre activitatea UC si activitatea dispozitivelor I/O.

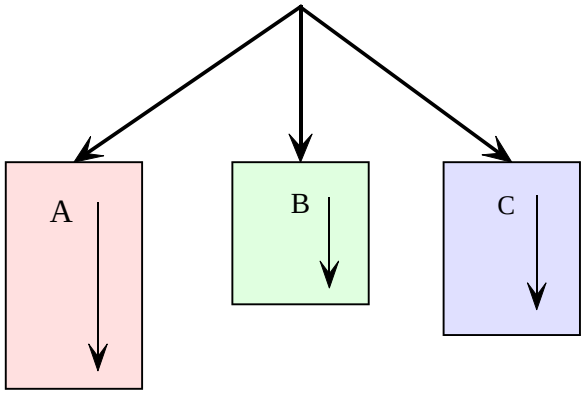
3.2.Modelul proceselor.

Numim procese programele in executie impreuna cu valoarea curenta a PC, registrii si variabile. Toate programele care se afla in executie dintr-un Sistem Calculator (incluzand si programele SO) sunt organizate intr-un numar de "procese secventiale".
Conceptual un proces are propriu sau *UC (virtual).* In realitate UC-ul real executa pe rand secvente din procesele lansate in executie. Comutarea UC-ului real de la un proces la altul se numeste **multiprogramare.**
Consideram ca la un moment dat in SC se afla 3 procese (programe existente in memoria SCH). Fiecare proces are propriul lui flux de executie dat de succesiunea valorilor Program Counter-ului (PC) si lucreaza independent unul de altul.
Aceste procese privite la intervale de timp observabile de catre utilizatorul uman al SC, avanseaza fiecare, cu toate ca la un moment de timp numai un singur proces se executa. Grafic multiprogramarea in cazul a trei procese (A,B,C) se poate reprezenta astfel:

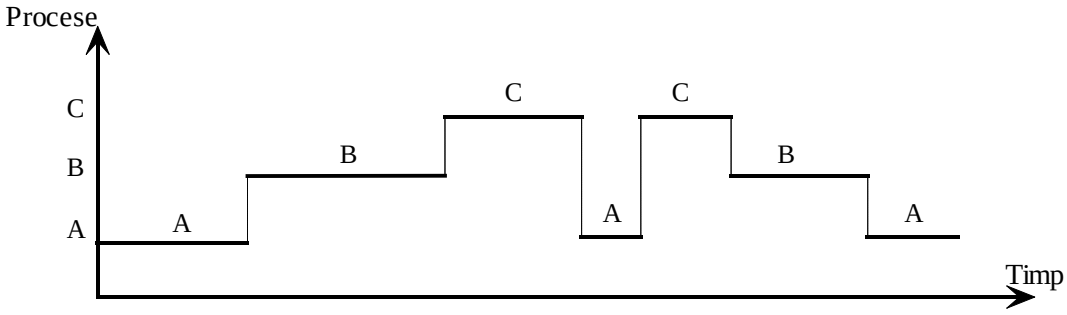


Modelul conceptual a 3 procese independente poate fi privit ca avand 3 Program Counter (PC).

Trei Program Counter



Dar la un moment dat (si orcare ar fi acel moment) un singur proces este activ.



Rata la care un proces se prelucreaza (se executa) nu va fi uniforma si probabil niciodata reproductibila. Daca se reia executia procesului probabilitaea este foarte mica ca el sa primeasca controlul procesorului pentru aceleasi perioade de timp si in aceleasi momente.
Nu se vor scrie niciodata programe cu **"presupunerea"** ca procesul provenit din program se va executa intr-un anume timp.
Ex. banda magnetica.
Pentru a se citi o informatie de pe banda magnetica este necesar ca banda sa ajunga dupa pornire la viteza sa normala de deplasare. De la pornirea motoarelor de antrenare banda ajunge la viteza nominala sa zicem dupa un timp de 2ms. Am putea sa folosim o metoda de asteptare a acestie perioade de 2ms inainte de a da comanda de citire efectiva a benzii folosind executia in ciclu a unei instructiuni a carei durata o stm precis. De ex. daca duarta instructiunii "ciclate" este 2ì s (micro secunde) atunci am putea sa facem un ciclu de executie de 1.000 de ori a acestei instructiuni. Teoretic aceasta metoda are ca efect o interziere a executiei cu exact 1.000 X 2ì s = 2.000ì s=2ms. Numai teoretic, pentru ca intr-un Sistem Calculator care ruleaza in regim de multiprogaramare nu se pot face asemenea "presupuneri". Producerea unor evenimente la momente de timp foarte precise reprezinta caracteristici de "timp real". Daca apar asemenea necesitati (necesitati de "timp real") adica anumite evenimente trebuie sa produca la intervale specificate de timp, trebuie luate masuri speciale iar SO trebuie sa fie prevazut cu functiuni speciale.
Diferenta intre proces si program este subtila si foarte importanta.
Se poate sugera un exemplu la care analogia (poate amuzanta) ne poate ajuta sa intelegem mai bine aceste diferente.
Ex. gospodina care incepe sa faca o prajitura dupa o reteta (program) folosind ingrediente (date de

intrare) si este intrerupta de un eveniment (de ex. Suna telefonul). Prajitura reprezinta rezultatul (datele de iesire).
Procesul este o activitate a UC de un anumit fel. El reprezinta un program, intrarile, iesirile si starea sa.
Un singur procesor poate fi "partajat" intre mai multe procese pe baza unui algoritm. Acest algoritm trebuie sa determine cind se intrerupe activitatea UC cu un proces pentru a continua activiatatea cu alt proces.

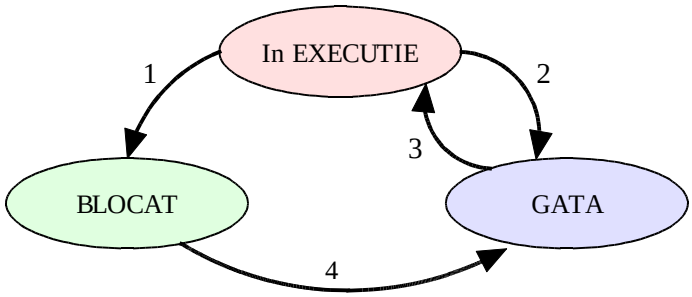
3.3.Ierarhia proceselor.

SO dispune de mecanisme pentru a satisface toate cerintele de care este nevoie pentru a manipula procese.
Aceste mecanisme sunt mecanisme pentru crearea si distrugerea proceselor.
In SO Unix procesele sunt create de functia sistem FORK care creaza un proces copie identic cu procesul care a apelat functia. Dupa apelul Fork procesul "parinte" continua executia in paralel cu procesul "fiu".
Procesul "parinte" poate genera mai multe procese "fiu". La randul lor procesele "fiu" poate genera procese. Astfel la un moment dat poate exista un arbore de procese. Procesele "parinte" si "fiu" pot fie executate in paralel. Sistemul de operare MS-DOS nu permite executia paralela a proceselor "parinte" si "fiu".

Starile proceselor

Fiecare proces reprezinta o entitate independenta, cu propriu sau PC, stare si date. De multe ori este necesar ca un proces sa interactioneze cu un altul. Un proces poate genera date (iesire) care pentru alt proces sunt date de intrare.
Ex. (Unix) **cat fisier1 fisier2 fisier3 | grep elev**
Primul proces (executia programului *cat fisier1 fisier2 fisier3*) concateneaza cele trei fisiere "fisier1", "fisier2" si "fisier3" produce ca rezultat un sir de informatii cu datele celor trei fisiere concatenate. Acest rezultat (iesirea procesului "cat") reprezinta intrare pentru procesul care executa programul "grep" care are ca efect selectarea inregistrarilor (liniilor) care contin cuvantul "elev".
Depinzand atat de complexitatea programelor cat si de prioritatea cu care SO trateaza procesele se poate ajunge in situatia in care procesul "grep" este gata de lucru dar nu are date de intrare. Procesul "grep" ramane blocat pana cand primeste date la intrare.
Aceasta stare de BLOCAT se datoreaza imposibilitatii din punct de vedere logic de a continua executia procesului. "Blocarea" nu se datoreaza unei cauze fizice. Este starea unui proces in care se asteapta date care nu sunt inca disponibile.
Dar este posibil ca un proces care este capabil de a-si continua executia sa fie stopat pentru ca SO a decis sa aloce UC altui proces.
Aceste conditii sunt diferite, primul caz datorandu-se insasi procesului, iar in al doilea caz datorandu-se tehnic strategiei SO .
Cele trei stari ale unui proces pot fi:

1. In **EXECUTIE** ” procesul utilizeaza in acel moment UC (CPU)
2. **GATA DE EXECUTIE** ” procesul ar putea fi in executie dar temporar este stopat alt proces fiind in executie.
3. **BLOCAT** ” procesul nu poate fi executat deoarece asteapta producerea anumitor evenimente externe (de ex. Aparitia unor date ce trebuiesc prelucrate de proces).



Sunt posibile 4 tipuri de **tranzitie** intre aceste stari:

1. Tanzitia din starea de EXECUTIE in starea BLOCAT. Procesul nu-si mai poate continua executia pentru ca asteapta date pentru prelucrat.
2. Tranzitia din starea EXECUTIE in starea GATA. "Planificatorul" stopeaza EXECUTIA procesului pentru a trece un alt proces in stare de EXECUTIE.
3. Tranzitia din starea GATA in starea EXECUTIE. "Planificatorul" alege procesul (care poate fi executat) pentru al trece in executie.
4. Tranzitia din BLOCAT in starea GATA. Atunci cand devin diponibile datele de intrare care au determinat intr-un moment anterior trecerea procesului in starea BLOCAT, procesul avand acum conditii pentru a fi executat.

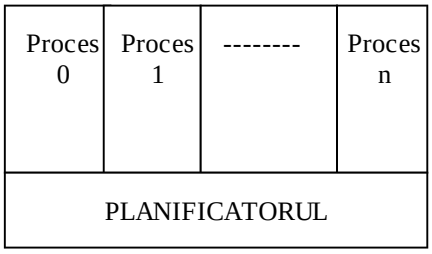
Trecerea proceselor dintr-o stare in alta este realizata de SO.

Tranzitia 1 are loc atunci cand un proces nu poate fi continuat din cauza ca asteapta aparitia unui eveniment extern. De ex. asteapta date ce trebuiesc prelucrate, adica date fara de care nu poate continua executia. In anumite SO, procesele pentru a trece in starea BLOCAT trebuie sa execute un apel sistem BLOCK. Cel mai frecvent un proces trece in starea BLOCAT atunci cand citeste date prin "pipe" sau de la terminal si datele nu au sosit inca. In aceste situatii procesul trece automat in starea blocat.

Tranzitia 2. (EXECUTIE ” GATA) si Tranzitia 3 (GATA ” EXECUTIE) sunt determinate de actiuni ale unui proces parte componenta a SO numit PLANIFICATOR (SCHEDULER). Acest proces nu este apelat vreodata explicit de catre un alt proces. Tranzitia 2 apare atunci cand PLANIFICATORUL hotaraste ca procesul a rulat perioada de timp prestabilita conform strategiei SO , si trebuie sa aloce CPU-ul altui proces. Tranzitia 3 are loc atunci cand conform atrategiei SO s-au indeplinit conditiilee ca procesul sa intre din nou in executie.

Rolul PLANIFICATORULUI este de a decide care proces trece in starea de executie si pentru cat timp. Algoritmul in baza caruia face acest lucru tine de strategia SO. Se cauta sa se tina cont cat de mult posibil atat de eficienta utilizarii S.C. cat si de o servire echitabila (echilibrata - fair) a fiecarui proces.

Tranzitia 4 (BLOCAT ” GATA) are loc atunci cand apare un eveniment extern care confirma terminarea unei activitati asteptata de proces care creaza conditiile ca procesul sa-si continue executia.



Cel mai scazut nivel SO este ocupat de PLANIFICATOR , deasupra caruia se afla o varietate de procese. Insasi procesele SO se afla deasupra PLANIFICATORULUI.
Planificatorul este automat lansat in executie la aparitia oricarui eveniment in sistem.

3.4.Implemntarea proceselor in SO.

Procesele sunt resurse "logice"gestionate de catre SO.
Gestiunea proceslor este realizata prin intretinerea (administrarea) unor tabele numite *Tabele de Procese*

Tabelele de procese sunt structuri de informatie care asigura cate o intrare pentru fiecare proces. Pentru fiecare intrare (deci fiecare proces) se pastreaza in fiecare moment informatii despre starea procesului. Cele mai importante informatii asociate unei intrari sunt:

- Program Counter (IP sau PC);
- Stack Pointer;
- memoria alocata procesului;
- starea (pozitia) fisierelor deschise;
- informatii de contabilitate si planificare si orice alta informatie ce se salveaza in momentul in care un proces trece din starea Executie in Gata in asa fel incat procesul sa poata fi trecut mai tarziu in starea de executie.

Continutul exact la *Tabelei de procese* difera de la un sistem la altul existand informatii "tipice" pentru fiecare scop.
Astfel pentru **managementul proceselor** se regasesc urmatoarele informatii:

- Registrele generale;
- IP;
- PSW;
- Stack Pointer (SP);
- Starea procesului;
- Timpul de cand procesul este pornit;
- Timpul de UC (CPU) consumat;
- Timpul de CPU (UC) consumat de procesele "fiu";
- Proces ID (Identificatorul Procesului);
- Diferite "flags-uri".

Pentru **Managementul memoriei:**

- Pointer la segmentele text;
- Pointer la segmentele de date;
- Pointer la segmentele cod;
- Starea de iesire;
- Starea semnalelor;
- ID-ul procesului;
- Procesul parinte;
- UID-ul si GID-ul;
- Diferite Flags-uri.

Pentru **Managementul fisierelor:**

- Directorul radacina;
- Directorul de lucru;
- Descriptorul de fisier;

UID-ul si GID-ul;
Parametrii de apel al functiei;
Diefriti biti Flags.

Iluzia ca la un moment dat exista in SC mai multe procese secventiale se pastreaza in sistemul calculator hard cu un singur procesor (UC) si mai multe de dispozitive I/O .
Vom vedea in continuare cum se comuta activitatea UC de la un proces la altul si cum se pune in aplicare un anumit tip de planificare a proceselor.
Stim deja ca asociat fiecarei clase de dispozitive I/O i se asociaza o locatie de memorie (de obicei in zona de inceput a memoriei) numita VECTOR DE INTRERUPERE. Aceasta locatie contine adresa unei *proceduri de serviciu* apartinand SO numita rutina de tratare a intreruperii (handler de intrerupere).
Presupunem ca la un moment dat se executa un proces X; anterior un proces Y trecuse in starea BLOCAT datorita unei cereri de date solicitata printr-o operatie de citire de pe un periferic (de ex. disc).
In acest moment starea Sistemului Calculator este urmatoarea:

se desfasoara o operatie de transfer de informatie (I/O) de la perifericul disc in memorie, transfer solicitat de procesul Y (care se afla in starea BLOCAT)
in paralel UC executa procesul X (in starea EXECUTIE).

Presupunem ca transferul de informatie disc memorie solicitat de procesul Y se termina ceea ce declanseaza o intrerupere.
Mecanismul care se va desfasura in continuare numit *mecanism de tratarea a intreruperii* este urmatorul:

1. IP (Program Counterul), PSW si unul sau mai multe registre sunt copiate in stiva ” mecanism hardware;
2. Se incarca registrul Program Counter cu continutul VECTORULUI DE INTRERUPERE ” mecanism hardware;

   Acest lucru face ca sa se continue executia de la adresa care se afla in VECTORUL DE INTRERUPERE.

De aici urmeaza activitati soft (procese ale SO).

3. Incepe executia rutinei de tratare a intreruperii care:

a. salveaza registrele generale in tabela de procese (TP) in intrarea corespunzatoare procesului intrerupt;
b. se copiaza informatiile memorate in stiva la aparitia intreruperii (stiva procesului intrerupt) in TP, in intrarea corespunzatoare procesului intrerupt si se trece indicatorul de stare al procesului intrerupt la valoarea GATA.
c. indicatorul de stiva (Stack Pointer) este fixat la stiva temporara utilizata de SO (pe durata tratarii intreruperii).

Aceste actiuni sunt indeplinite printr-o secventa de program scrisa de obicei in "asamblare".
Mai departe urmeaza o secventa scrisa de obicei in "C" care:

d. determina care proces a lansat cererea de I/O a carei terminare a declansat intreruperea. Si starea acestui proces este modificata din BLOCAT in GATA .

In continuare se lanseaza PLANIFICATORUL care in conformitate cu strategia SO alege urmatorul proces care va fi executat, dintre procesele care se gasesc starea GATA.
n acest moment exista cel putin 2 procese care sunt in starea GATA, procesul intrerupt si procesul care a solicitat operatia de I/O a carui terminare a fost semnalata de Intrerupere.
PLANIFICATORUL va alege in conformitate cu algoritmul implementat un proces care va fi trecut

din starea GATA in starea EXECUTIE . Acest lucru se face printr-o secventa de program (in ansamblare) care incarca registrele IP, PSW, etc cu informatiile memorate in tabela de procese actiune care se mai numeste si *reface contextul programului.*
Se schimba indicatorul procesului de stare din GATA in EXECUTIE. In continuare UC-ul va executa instructiuni ale procesului planificat pentru executie de catre SO.
Prin sistemul de intreruperi se face transferul executiei de la un proces oarecare la procesle SO permitand astfel SO sa asigure administrarea intregii activitati a SC.
Algoritmul de planificare depinde de la sistem la sistem si se face incercand un compromis intre eficienta (incarcare maxima a SC hard) si servire echitabila a proceselor active.

De ex. SO Unix Standard se urmareste maximizarea gradului de paralelism prin acordarea unei prioritati mai mari proceselor cu mai multe cereri de I/O.

Activiatatea Sistemului Calculator este dirijata de evenimente (Event Driven).

Acest lucru semnifica faptul ca activitatea SC este analizata si dirijata (prin SO) numai la aparitia unui *eveniment.*
Aceste evenimente sunt intreruperile *sincrone* sau *asincrone*.
Intreruperile *sincrone* asa cum am mai discutat, sunt datorate unui cod de instructiune inexistent (cod eronat sau *apel sistem*) adica de executia in sine a unei instructiuni.
Intreruperile *asincrone* sunt produse de dispozitive fizice care sunt subunitatile functionale ale SC.
Sursa intreruperilor *asincrone* poate fi:
- subunitatile de I/O ca urmare a activitatii cu aceste dispozitive;
- subunitatile de tip CEAS. Aceste dispozitive dupa scurgerea un perioade de timp (fixata la activarea TIMER-ului) declansaza o o intrerupere.

Teoretic exista probabilitatea ca la un moment dat pentru o lunga perioada de timp sa nu apara nici-o intrerupere *sincrona* sau *asincrona* de la dispozitve de I/O. Aceasta ar insemna ca SO nu ar putea in aceste perioade sa preia controlul activitatii deci nu ar putea sa-si indeplineasca functiile (de ex. Planificarea proceselor).
Pentru a preantampina asemenea situatii insasi SO activeaza dispozitive de tip TIMER ceeace determina aparitia unor evenimente in activitatea SC la perioade fixe de timp. Astfel se "forteaza" aparitia unor evenimente indiferent daca exista sau nu dispozitive de I/O care sa declanseze intreruperi, sau instructiuni care sa produca intreruperi sincrone.
In acest fel este sigur ca periodic, SO va prelua controlul activitatii SC si prin PLANIFICATOR si celelalte componente ale sale va asigura indeplinirea functiilor proprii fiecarui SO.
Algoritmul de planificare depinde de la sistem la sistem si se face incercand un compromis intre eficienta (incarcare maxima a SC hard) si servire echitabila a proceselor active.
De ex. SO Unix Standard se urmareste maximizarea gradului de paralelism prin acordarea unei prioritati mai mari proceselor cu mai multe cereri de I/O.

3.5 FIRE (Thred-uri)

Sistemele de operare traditionale considera procesle ca avand un singur "fir"(secventa) de executie.Adica intreg procesul are propriul spatiu de adrese dar instructiunile procesului sunt privite ca facand parte dintr-o singura secventa (fir) de control.
Apare deseori situatia in care de dorit ca procesul sa fie constituit din mai multe "fire" de control care se gasesc in acelasi spatiu de adrese , executandu-se quasi-paralel ca si cum ar fi procese separate.

3.5.1. Modelul "firelor".

Modelul procesului se baza pe 2 concepte: resurse(grup de resurse) si executia.
Un proces poate fi privit ca fiind gruparea resurselor asociate executiei unui program. Aceste resurse asociate sunt spatiul de adrese de memorie care contine atat instructiunile programului cat si datele sale si alte resurse. Aceste alte resurse se refera la fisierele deschise, procesele "fiu", semafoare, semnale, etc. Punand toate aceste resurse impreuna SO poate administra intr-un mod unitar programele in faza lor de executie.
Un alt mod in care se poate privi un proces este "firul" (secventa) de executie numit simplu FIR. FIRUL are cateva elemente asocite lui cum ar fi: Program Counter (PC) care precizeaza urmatoarea instructiune ce se va executa, registrele (generale) care contin datele de lucru curente si *stiva*, care contine istoricul rutinelor apelate din care nu s-a revenit inca.
Cu toate ca "firul" nu poate fi despartit de proces , ele sunt doua concepte diferite. *Procesele* sunt utilizate pentru gruparea unor resurse (impreuna) iar *firele* sunt entitati care sunt planificate pentru executie la procesor.
Firele aduc in plus fata de modelul *procesului* faptul ca sunt permise multiple executii cu un grad mare de independenta una de alta care au loc in acelasi grup de resurse.
Avand FIRE multiple executate in paralel in cadrul aceluiasi proces este ca si cum am avea mai multe procese executate in paralel pe acelasi SC.
Deosebirea exista insa intre *fire* si *procese*.
In cazul *firelor* multiple de executie in cadrul aceluiasi *proces*, *firele* partajaeaza (folosesc in comun) acelasi spatiu de adrese fizice, acelaesi fisiere, aceleasi semafoare,etc.
In cazul proceselor diferite , procesele partajeaza aceleasi resurse fizice, adica aceiasi UM, aseleasi dispozitive periferice (discuri, imprimante, etc.).
Deoarece *firele* au multe caracteristici comune cu *procesele* ele mai sunt numite si "procese usoare".
Termenul de multithred este utilizat pentru a descrie situatia in care intr-un singur proces avem mai multe fire de executie.

3.6 Comunicatia interproces (IPC - **I**nter**P**rocess**C**ommunication)

Pe parcursul activitatii SC apare deseori necesitatea ca un proces sa comunice cu altul. Am vazut asta si intr-un exemplu anterior cum datele de iesire a primului proces reprezinta date de intrare pentru un al doilea, intr-o comanda "pipe-line". Acesta este un exemplu de comunicare intre procese. In cotinuare o sa vedem si alte situatii cand este necesara o "comunicare" intre procese.

3.6.1 Conditiile de competitie

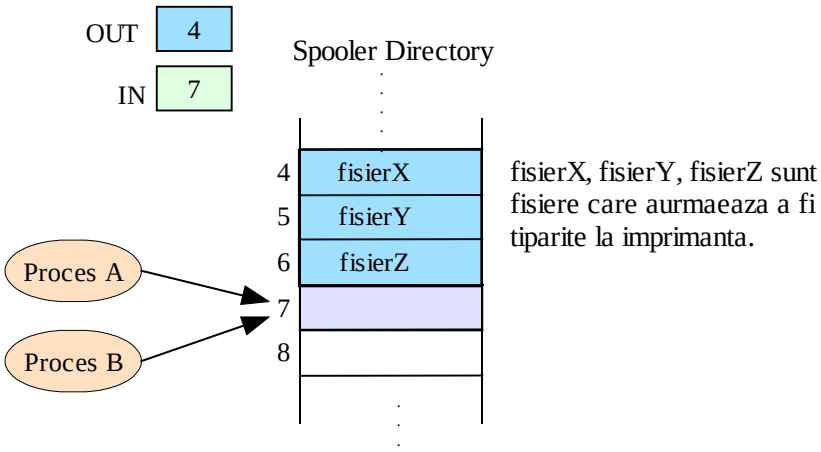
In activitatea sistemului pot aparea situatii in care doua sau mai multe procese "partajeaza" (impart) anumite zone de stocare comune, zone care pot fi citite sau scrise de fiecare proces. Aceste zone de stocare "partajate" pot fi in memoria principala sau pot fi "fisiere partajate".
Problemele care apar la exploatarea de catre mai multe procese a "zonelor partajate" sunt aceleasi indiferent unde sunt ele fizic. Pentru a vedea cum IPC lucreaza in practica vom urmari modul de lucru al "print spooler-ului" - serviciul de tiparire al informatiilor in sistem.
Atunci cand un proces doreste sa tipareasca un fisier, el introduce printr-un ApelSistem, numele fisierului intr-o structura de date de tip lista circulara, numita *"director spooler"*. Un alt proces numit "PRINTER DAEMON" (DEMON vrajitor de imprimanta) periodic verifica daca exista fisiere care trebuie listate, daca da, le listeaza si apoi elimina numele lor din directorul spooler.
Imaginea "directorului spooler" este un numar de locatii (sloturi) numerotate 0,1,2,3 .... fiecare slot putand pastra un nume de fisier. Pentru administrarea "directorului spooler" sunt asociate 2 variabile partajate:

- "OUT "” care indica numele fisierului care urmeaza a fi tiparit;

si

- "IN" care puncteaza (indica) numarul urmatorului slot liber.

Presupunem ca la un moment dat sloturile de la 0 la 3 sunt libere iar 4 la 6 sunt ocupate (cu nume de fisiere ce trebuie tiparite).
Este posibil ca doua procese A si B aproape simultan sa decida sa tipareasca cate un fisier. Din acest motiv poate aparea urmatoarea situatie:
Procesul A citeste var "IN" si memoreaza continutul ei (valoarea 7) intr-o variabila locala. In acest moment daca apare o intrerupere procesul A se suspenda in acest punct. Dupa tratarea intreruperii PLANIFICATORUL da controlul procesului B. Procesul B se pregateste si el pentru a tiparii un fisier. De aceea B citeste variabila IN (contine valoarea 7) si o memoreaza intr-o variabila locala si apoi transfera in locatia 7 a "spooler directory" numele fisierului ce doreste sa fie listat. Dupa un timp se reia procesul A din punctul in care a fost intrerupt. Procesul A depune numele fisierului ce doreste sa fie listat unde ? In pozitia citita din var IN adica intrarea 7 a "directorul spooler". Acest lucru face ca peste numele depus de procesul B sa se memoreze numele fisierului ce trebuie listat al procesului A. Deci solicitarea procesului B se pierde.



Situatii ca aceasta in care *doua sau mai multe procese citesc sau scriu anumite date partajate si rezultatul final depinde de cine si in ce moment precis face citirile sau scrierile* se numesc **CONDITII DE COMPETITIE**.

3.6.2 Sectiuni critice

Ce trebuie facut pentru a se evita CONDITIILE DE COMPETITIE ?
Metoda prin care se evita CONDITIILE DE COMPETITIE se numeste EXCLUDERE MUTUALA.
EXCLUDEREA MUTUALA este mecanismul prin se evita situatia in care daca un proces utilizeaza *variabile partajate* sau *fisiere partajate* si alte procese sa faca acest lucru cu aceleasi variabile sau fisiere partajate in acelasi timp.
Secventa dintr-un program unde se acceseaza zone de memorie sau fisiere partajate se numeste *sectiune critica.*
Deci CONDITIILE DE COMPETITIE se poat evita daca vom putea sa impiedicam ca doua procese sa se afle in *sectiunea critica* in acelasi timp, referitor la aceleasi *variabile* sau *fisiere partajate*.
Pentru a putea avea procese paralele care sa coopereze corect si eficient la utilizarea datelor partajate este necesar pe langa evitarea CONDITIILOR DE COMPETITIE sa solutionam si alte 3 conditii speciale. Astfel putem sa enumeram cele 4 conditii care :

1. Sa nu existe doua procese simultan in *sectiunea critica;*
2. Sa nu existe o dependenta a activitatilor de viteza sau numarul de CPU-uri;
3. Procesele care ruleaza in afara proceselor critice sa nu poate bloca alte procese;
4. Nici un proces nu trebuie sa astepte la nesfarsit pentru a intra in sectiunea sa critica.

3.6.3 Excluderea mutuala cu "Busy Waiting" (in traducere :asteptand ocupat).

In primul rand o sa incercam sa intelegem mai bine acest "Busy Waiting". Busy Waiting se refera de fapt la o stare in care se asteapta terminarea unei activitati fara sa putem executa altceva in aceasta perioada de asteptatare. Acest tip de "asteptare cu ocuparea procesorului" nu este recomandata deoarece conduce la o exploatare ineficienta a SC.
In aceasta sectiune vom studia cateva metode posibile de EXCLUDERE MUTUALA.
Adica vom explica cateva metode prin care daca un proces actioneaza asupra unor zone de memorie "partajata" adica se afla intr-o "sectiune critica" sa se impiedice ca orce alt proces sa intre in "sectiunea sa critica" referitoare la aceleasi zone "partajate".

- Dezactivarea intreruperilor.

Aceata metoda se refera la eliminarea "multiprogaramarii" atunci cand procesul are acces la resursele partajate. Adica dezactivarea intreruperilor pe perioada in care un proces se afla in sectiunea critica. Acest lucru va face ca in aceasta perioada sa nu se mai poata lansa in executie alt proces eliminandu-se posibilitatea ca sa se ajunga in CONDITIILE DE COMPETITIE.
Aceasta solutie este foarte simpla dar are un mare dezavantaj. Daca dintr-o eroare programul utilizator nu mai iese din sectiunea critica atunci nici intreruperile nu se activeaza niciodata. Acest lucru este inacceptabil pentru ca activitatea intregului SC ar putea depinde de un program utilizator.
Deasemeni aceasta metoda in cazul sistemelor multiprocesor nu rezolva problema.

- Blocarea cu variabile Software

Aceasta este o alta solutie prin care se incearca EXCLUDEREA MUTUALA dar asa cum vom explica in continuare aceasta nu este o buna rezolvare.
Considerand ca avem o variabila partajata (lock) care este folosita ca un "fanion" care indica daca un proces este sau nu in sectiunea critica.
Initial aceasta variabila are valoarea "0". Atunci cand procesele trebuie sa intre in "sectiunea critica" mai intai testeaza variabila "lock". Daca valoarea ei este "0", procesul modifica aceasta valoare in "1" si apoi intra in sectiunea critica. Daca variabila "lock" era deja "1" , procesul va astepta pana cand ea devine "0" si apoi intra in sectiunea critica.
Din pacate la o privire mai atenta aceasta metoda conduce la o situatie similara cu cea discutata la imprimarea cu ajutorul "directorului spooler".
Intre momentul citirii variabilei "lock" si momentul "setarii" ei (modificarea valorii) este posibil ca procesul sa fie intrerupt de un altul care sa incerce sa faca acelasi lucru ceeace conduce la indeplinirea CONDITIILOR DE COMPETITIE.

- Alternarea stricta .

Este o rezolvare posibila dar asa cum vom vedea nu este potrivit pentru situatiile cand un proces este mult mai lent decat altul.
Se va folosii o variabila numiata **turn** care face posibila alternarea de la un proces la altul a parcurgerii *sectiuni critice.*

```
While (TRUE) {
        While (turn !=0); /*WAIT*/
                Sectiune critica));
                turn=1;
                sectiune_necritica( );
        }
                ( a )
```

```
While (TRUE) {
        While (turn !=1); /*WAIT*/
                Sectiune critica();
                turn=0;
                sectiune_necritica( );
        }
                ( b )
```

Dezavantajul acestei metode este ca o sectiune necritica poate sa blocheze intrarea altui proces in sectiune critica. In cazul proceselor cu durata mult diferita a *sectiunilor NE-critice* este posibil ca procesul cu duarat mai mica sa astepte nejustificat de mult permisiunea intrarii in sectiunea critica.

Solutia Peterson

Aceasta solutie combina utilizarea "variabilelor software de blocare" cu variabila de alternare "turn" fara insa sa determine o alternare stricta a accesului la variabilele partajate. Aceasta idee a fost elaborata de G.L. Peterson in anul 1981.
Implementarea intr-o rutina scrisa in ANSI C este urmatoarea:

```
# define N          2          /* numar de procese */
# define FALSE      0
# define TRUE       1
int turn;                      /* variabila de alternare: al carui proces este ? */
int interes [N];               /* toate valorile sunt initial 0 (FALSE) */
void  intrare_regiune (int proces)  /* proces = cine a intrat (procesul 0 sau 1) */
{
        int altul;             /*  "alt" proces */
        altul = 1   proces;    /* numarul "celuilalt" proces */
        interes [proces] = TRUE;    /* arata interesul pentru sectiunea critica */
        turn=proces;           /* setez variabila de alternare */
        while (turn==proces && interes[altul]==TRUE); /*astept daca alt proces a fost "interesat"*/
}
void  iesire_regiune (int proces)   /* proces: cine paraseste (0 sau 1) */
{
        interes[proces]=FALSE;      /*indica parasirea reg. Critice*/
}
```

Solutia Peterson combina ideia alternarii cu cea a variabilei de blocare.
Inainte de a utiliza variabilele partajate, adica inainte de a intra in sectiunea critica, fiecare proces trebuie sa apeleze o rutina "intrare_regiune" cu parametrul de apel numarul procesului (0 sau 1). Aceasta rutina va produce pentru procesul appelant o stare de asteptare, daca este nevoie, pana cand se indeplinesc conditiile de acces la avariabilele partajate. Dupa terminarea activitatilor cu variabilele partajate (iesirea din sectiunea critica) procesul trebuie sa apeleze rutina "iesire_regiune" indicand astfel ca alt proces poate sa acceseze variabilele partajate.
Descrierea algoritmului:
Initial nici un proces nu este in sectiunea critica. Presupunem ca procesul "0" apeleaza "intrare_regiune". Atunci *interes[0]* devine TRUE si "turn=0". Daca in acest timp procesul "1" nu este interesat (*interes[1]*=FALSE), procesul "0" va iesi imediat din rutina "intrare_regiune", va intra in sectiunea critica. Apoi procesul 0 va apela rutina "iesire_regiune" care va seta parametrul "interes[0]=FALSE, permitind si procesului "1" sa acceseze sectiunea critica.
Situatia cea mai defavorabila este cea in care ambele procese (0 si 1) apeleaza rutina "intrare regiune" in aceelasi timp. In acest fel ambele procese o sa aiba "interes[proces]" corespunzator, setat TRUE. Se constata ca apar totusi CONDITII DE COMPETITIE (fara EXCLUDERE MUTUALA) la setarea variabilei *turn*, ceeace nu afecteaza controlul corect a accesului in sectiunea critica.Variabila *turn* va lua valoarea corespunzator ultimului proces care a scris in ea (0 sau1). Acest lucru nu inseamna insa ca ambele procese apelante a rutinei "intrare_regiune" vor si parasi aceasta rutina. Una din rutine va fi blocata pe instructiunea *WHILE* si anume procesul care a scris ultimul variabila partajata *turn* (deoarece conditia din instructiunea WHILE va fi adevarata). Celalalt proces (cel care a setat primul variabila *turn*) va iesi din rutina "intrare_regiune" putand sa acceseze "variabilele partajate" (in regiune critica) fara riscul ca si celalalt proces sa acceseze in acelasi timp aceleasi variabile partajate. La iesirea din regiunea critica procesul trebuie a apeleze

rutina "iesire_regiune" care seteaza *interes[proces]* la valoarea FALSE, permitand si celuilalt proces sa intre in regiunea critica.

Solutia TSL

Conditiile de competitie apareau din cauza ca intre momentul citirii variabilei software de blocare si momentul setarii ei, exista posibilitatea ca procesul sa fie intrerupt permitand ca si alt proces sa faca acelasi lucru cu acea variabila.

Exista SC care in setul de instructiuni cablate au o instructiune "TEST AND SET LOCK" (TSL).

TSL RX,LOCK

Executia acestei instructiuni are ca efect citirea continutului unui cuvant memorie (LOCK) intr-un registru (RX) si apoi inscrierea unei valori diferita 0 la adresa de memorie respectiva. Aceste 2 faze sunt indivizibile (actiune atomica). In plus pentru SC multiprocesor prin mecanisme hardware daca un CPU executa o instructiune TSL pe perioada executiei se blocheaza magistrala de memorie blocand accesul altui CPU la acelasi cuvant de memorie (LOCK)..

Utilizarea acestei instructiuni pentru excludere mutuala presupune ca si in cazul metodei Peterson construirea a doua rutine *intrare_regiune* si *iesire_regiune* pentru blocarea respectiv deblocarea accesului altui proces la sectiunea critica.

intrare_regiune

tsl registru,flag	copiaza continutul lui "flag" in registru si seteaza "flag" la 1
cmp registru,#0	a fost flag=0?
jnz intrare_regiune	daca nu a fost "0" ne intoarcem la tsl
ret	daca a fost "0" se iese din rutina lasand "flag diferit de 0"

iesire_regiune

mov flag,#0	pune 0 in flag permitand si altui proces accesul la var.partajate
ret	

ECLUDEREA MUTUALA se realizeaza cu ajutoul unei variabile *flag* citita si modificata printr-o singura instructiune TSL. Cand flag=0 orice proces (dar numai unul la un moment dat) poate sa seteze "flag" la "1" utilizand instructiunea TSL si apoi sa acceseze sectiunea critica . La iesirea din sectiunea critica acelasi proces prin rutina "iesire_regiune" memoreaza "0" in "flag" permitand si altor procese sa aiba acces la sectiunea critica.

Concluzii:

Ambele metode, Petreson (SC monoprocesor) si TSL (SC mono si multiprocesor), sunt corecte ele reprezenatand o solutie de EXCLUDERE MUTUALA. Dar ambele au neajunsul ca sunt de tip "busy waiting" (asteptare prin ocuparea procesului). Solutiile cu "busy waiting" reprezinta o utilizare neeficienta a procesorului.

De remarcat ca la SC cu planificare bazata pe prioritati (stricta), aceste solutii nu rezolva in mod corect problema excluderii mutuale, putand sa produca rezultate neprevazute. In cazul in care in sunt active 2 procese (unul cu prioritate mai mare decat altul) se poate ajunge la asa numita "problema inversarii prioritatilor ".

Aceasta este o problema poate aparea daca doua procese acceseaza aceleasi "resurse partajate" si au prioritati diferite. Daca procesul cu prioritate mai mica se afla in *sectiunea critica* si va fi intrerupt iar PLANIFICATORUL transfera executia procesului cu prioritate mai mare, acesta se va bloca intr-un ciclu infinit pe instructiuniledin *intrare_regiune*.

3.6.4 SLEEP si WAKEUP

Pentru evitarea "Busy Waiting" (asteptarii cu ocuparea procesorului), solutia este ca SO sa puna la dispozitia utilizatorilor sai o serie de apeluri sistem specifice IPC-ului. Aceste "apeluri sistem" trebuie sa "blocheze" procesul care incearca sa intre in *sectiunea critica* , in timp ce alt proces este deja in *sectiunea critica*. Blocarea procesului care nu are indeplinite conditiile de a intra in *sectiunea critica* nu trebuie facuta printr-o instructiune sau mai multe instructiuni care tin ocupat procesorul (timp procesor irosit inutil).
Solutia este ca procesul care nu are indeplinite conditiile de intrare intr-o *sectiune critica*, sa fie intrerupt de catre SO, astfel incat sa fie posibil ca pe toata aceasta perioada sa fie executate alte procese.
O rezolvare simpla se poate da acestei probleme prin perechea de apeluri sistem SLEEP si WAKEUP.
SLEEP - produce blocarea procesului (suspenda procesul) pana cand un alt proces il "trezeste" prin WAKEUP.
WAKEUP - reporneste (trezeste) procesul precizat prin parametrului apelului.

Problema Producator Consumator

Pentru exemplificarea utilizarii apelurilor sistem SLEEP - WAKEUP vom considera o problema clasica, asa numita problema "Producator-consumator" numita si problema "buffer-ului limitat". Aceasta problema apare atunci cand doua procese impart o zona comuna de dimensiune fixa. Unul dintre procese

Producator - care depune informatia in aceasta zona (buffer) iar alt proces
Consumator le preia din buffer.
In acest caz probleme pot sa apara in doua situatii:

- atunci cand *producatorul* trebuie sa depuna o noua informatie in buffer si acesta este plin;
- atunci cand *consumatorul* doreste sa preia date din buffer dar acesta este gol.

Solutia la aceste doua situatii utilizand cele 2 apeluri sistem SLEEP si WAKEUP este urmatoarea:
- *producatorul* sa treaca in starea "blocat" prin apelul functiei SLEEP daca bufferul este plin. El va fi "trezit" (scos din aceasta stare) cu un WAKEUP atunci cand *consumatorul* va extrage din buffer unul sau mai multe elemente.
- *consumatorul* atunci cand doreste sa extraga date din "buffer" si-l gaseste gol va trece in starea blocat (SLEEP) pana cand *producatorul* va depune informatia in buffer.
Aparent lucrurile sunt simple dar si in acest caz pot aparea "conditii de competitie".
Se foloseste o variabila "contor" care tine evidenta ocuparii bufferului, valoarea sa maxima fiind "N" (numarul sloturilor buffer-ului) , Algoritmul este destul de simplu:

Producatorul testeaza "contor" si daca este "N" , va lansa SLEEP, daca nu este N, va introduce un articol in buffer si va incrementa contorul. Daca aceasta devine 1 va lansa WAKEUP pentru a relansa *consumatorul* eventual blocat de N=0.

Consumatorul testeaza daca N=0 (buffer gol). Daca da va trece in starea blocat SLEEP. Daca nu va extrage un articol (element) din buffer si va decrementa contorul. Daca acesta devine N-1 (asta inseamna ca fusese N) va trezi producatorul prin WAKEUP si va utiliza elementul extras.

Implementarea in "C" a acestei probleme este urmatoarea:

```
# define N 100
int contor=0
void producator (void)
{
        int element;
        while (TRUE) {                                  /*Repeta totdeauna*/
                element=produce_element;                /*genereaza urmatorul element*/
                if(contor= =N) SLEEP( );                /*daca baffer plin"  SLEEP*/
                depune_element(element );               /*pune elemnt in buffer*/
                contor=contor+1;
                if(contor= =1) WAKEUP(consumator);      /*a fost buffer gol*/
        }
}

void consumator(void)
{
        int elemnt;
        while(TRUE)  {                                  /*Repeta la nesfarsit*/
                if (contor= =0) SLEEP( );               /* daca buffer gol"  SLEEP*/
                element=extrage_element();              /*extrage elem. din buffer*/
                contor=contor-1;
                if (contor= =N-1) WAKEUP(producator);   /*a fost plin*/
        }
}
```

CONDITIILE COMPETITIE pot aparea si in acest caz din cauza ca variabila *contor* este accesat fara nici-un control de ambele procese. De ex. atunci cand bufferul este gol si procesul *consumator* este intrerupt imediat dupa ce a citit *contor* in vederea testarii (inainte de SLEEP), lansandu-se procesul *producator*. *Producatorul* introduce un element, incrementeaza *contor* devenind "1" ceeace face sa lanseze un WAKEUP(consumator). Dar in acest moment *consumator* nu este in stare blocat (SLEEP), semnalul WAKEUP pierzandu-se. Atunci cand SO va lansa procesul *consumator* el va continua activitatea din punctul in care a fost intrerupt. C*ontor*-ul fusese citit inainte de intreruperea sa si "0" trece in SLEEP.
Mai devreme sau mai tarziu si *producator* va trece in SLEEP (pentru ca va umple bufferul) in timp ce *consumatorul* era in SLEEP.
Amandoua vor fi in starea blocat la infinit unul datorita celuilalt (stare de inter-blocare fara sfarsit DEADLOCK).
Problema se poate rezolva adaugandu-se o noua variabila "wakeup waiting bit" care va fi setata (1) de catre executia lui WAKEUP si testata inainte de SLEEP. Daca inainte de SLEEP, "wakeup waiting bit" este egala cu "1", va fi trecut in zero si nu va mai lansa SLEEP.
"wakeup waiting bit" ” "bit asteptare trezire".
In cazul in care la problema "producator-consumator" participa mai multe procese este nevoie de mai multe variabile "wakeup waiting bit" ceea ce complica problema.

3.6.5. Semafoare.

Conceptul de *semafor* a fost propus de Dijkstra in anul 1965. cu scopul de a avea o variabila intreaga care sa "contorizeze" WAKEUP-urile care nu sunt "acoperite" de SLEEP-uri.
Pentru situatia in care apar mai multe WAKEUP-uri succesive neavand procese in starea SLEEP, se introduce acest nou tip de variabila semafor care va numara "wakeup-urile" ce trebuie luate in considerare in viitor.
Aceasta variabila semafor are valoarea "0" daca nu a fost nici-un WAKEUP netratat si o valoare pozitiva pentru WAKEUP-uri netratate (in suspensie).
Corespunzator acestui tip de variabila se introduc doua tipuri de *apel sistem* DOWN si UP ca o generalizare a lui SLEEP si WAKEUP.

down(*semafor***)** testeaza o variabila *semafor*. Daca este mai mare decat "0" se decrementeaza valoarea *semaforului* si continua executia. Daca *semaforul* este "0" procesul trece in "starea blocat" (SLEEP). Aceasta secventa de actiuni este indivizibila (actiune atomica).

up(*semafor***)** incrementeaza valoarea *semaforului*, permitand la unul din procesele care erau SLEEP pe acest *semafor* de a-si completa operatia **down** inceputa. Astfel dupa o operatie **up** pe un *semafor* , existand procese in starea SLEEP pe acest *semafor* valoarea *semaforului* ramane zero dar vor fi mai putine procese in starea SLEEP. Apelul sistem **up**(*semafor*) este deasemeni o actiune atomica.

In acest fel avem la dispozitie doua functii care asigura o buna comunicare intre procese IPC .
In sistemele care au implementate *semafoare* si functiile DOWN si UP, vom putea extinde utilizarea acestora si pentru operatii de I/O. In acest caz vom asocia cate un semafor pentru fiecare dispozitiv de I/O, initial cu valoarea "0". Imediat dupa lansarea unei cereri de I/O se executa functia DOWN care blocheaza procesul. Atunci cand soseste intreruperea de sfarsit de operatie de I/O, handlerul de intreruperi executa un **up** pe semaforul dispozitivului respectiv, ceea ce face ca procesul sa fie deblocat si trecut in starea "Gata".
Implemantarea functiilor **down** si **up** se face in biblioteca SO ca "apeluri sistem". Pe perioada cat se testeaza semaforul, se actualizeaza si se pune procesul in stare de blocat (SLEEP), se dezactiveaza intreruperile. Acest lucru se intampla pe durata a catorva instructiuni ceea ce nu afecteaza functionarea sistemului.
In sistemele multiprocesor fiecare semafor trebuie protejat cu variabile de blocare cu ajutorul instructiunilor TSL pentru a fi siguri ca numai un CPU examineaza la un moment dat semaforul. Instructiunea TSL produce (busy waiting) pe o scurta durata de timp ceeace nu afecteaza major performanta sistemului.

Rezolvarea problemei *Producator-Consumator* utilizand semafoare.
Problema Producator-consumator se rezolva utilizand 3 semafoare:

semaforul *plin* (ocupat) contabilizeaza numarul de sloturi ocupate;
semaforul *gol* (liber) contabilizeaza numarul de sloturi libere;
semaforul *exclud_mutual* (excludere mutuala) care face sigur ca *producatorul* si *consumatorul* nu acceseaza in acelasi timp bufferul.

plin = 0 initial
gol = N initial
exclud_mutual = 1 initial

Semaforul *exclud_mutual* ia valoarea 0 si 1 ” semafor binar.

Fiecare proces va executa un **down**(&exclud_mutual) inainte de a intra in sectiunea critica si **up**(&exclud_mutual) dupa iesirea din sectiunea critica.

```
# define N 100                          /*numarul de sloturi in bufer*/
typedef int semafor;                    /*semafoarele sunt int speciale*/
semafor exclud_mutual = 1;              /*controleazaaccesulla sectiunea critica*/
semafor gol = N;                        /*numara sloturile libere*/
semafor plin = 0;                       /*numara sloturile ocupate*/

void producator (void)
{
        int element;
        while (TRUE){                   /*intotdeauna adevarat"  ciclu infinit*/
                element=produce_element(); /*generare elemnt de pus in buffer*/
                down(&gol);             /*decrementez nr. de sloturi libere*/
                down(&exclud_mutual);   /*exclud_mutual=0 blochez acces alt proces*/
                pune_element(element);  /*pune element in buffer*/
                up(&exclud_mutual);     /*exclud_mutual=1,permite acces alt proces*/
                up(&plin);              /*incrementez nr. de sloturi ocupate*/
        {
}

void consumator(void)
{
        int element;
        while (TRUE){                   /*intotdeauna adevarat* - ciclu infinit*/
                down(&plin);            /*decrementeaza nrumar sloturi ocupate*/
                down(&exclud_mutual);   /*fac exclud_mutual=0 blochez acces alt proc*/
                element=extrage_element(); /*extrage element din buffer*/
                up(&exclud_mutual);     /*exclud_mutual=1, permite acces altui proc. */
                up(&gol);               /*incrementez nr. de sloturi libere*/
                utilizeaza_elemnt(element); /*utilizez elementul*/
        }
}
(gol si plin "  pentru sincronizare)
```

3.6.6. Metoda cu contoare evenimente

Rezolvarea problemei producator-consumator se poate face si altfel decat prin "excludere mutuala". Cu ajutorul unei variabile speciale *contor evenimente* se pot rezolva probleme de asemenea tip fara a se face o excludere mutuala a proceselor producator respectiv consumator.
Metoda este aparent mai simpla dar ea trebuie sustinuta de catre SO care trebuie sa puna la dispozitie apeluri sistem specifice (read, advance, await) care opereaza cu contoare de eveenimente.
Asupra variabilelor *contor evenimente* se pot face trei operatii:

1. **read**(E); Citeste (intoarce) valoarea curenta a contorului
2. **advance**(E); Incrementeaza automat E cu "1".
3. **await**(E,V); Asteapta pana cand E capata o valoarea egala cu "V" sau mai mare. Unde "E" variabila contor eveniment.

Intotdeauna contoarele de evenimente cresc. Niciodata nu descresc.

Pentru rezolvarea problemei *producator-consmator* se utilizeaza doua *contoare evenimente*. Primul contor evenimente "in" contorizeaza cumulativ numarul de elemente pe care producatorul le-a pus in buffer de la pornirea programului. Celalalt contor "out" contorizeaza cumulativ numarul de elemnte pe care consumatorul le-a extras din buffer.

Valoarea din "in" trebuie sa fie mai mare sau egala cu "out" dar nu cu mai mult decat dimensiunea buffer-ului (N).

```
# include "prototypes.h"
#define N 100
typedef int event_counter;
event_counter in=0;
event_counter out=0;

void producator(void)
{
        int elemnt,secventa=0;
        while(TRUE) {
                produc_element(&element);
                secventa=secventa+1;
                await(out,secventa-N);
                pune_element(element);
                advance(&in);
                }
}
```

```
        void consumator(void)
{
        int element, secventa=0;
        while(TRUE) {
            secventa=secventa+1;
            await(in,secventa);
            extrage_element(&element);
            advance(&out);
            utilizeaza_elemnt(element);
            }
}
```

Producatorul dupa producerea unui element incearca sa-l puna in buffer. Prin apelul sistem **await(out,secventa-N)** este pus in asteptare daca nu este loc in buffer, adica daca out (numarul elementelor extrase) nu este cel putin egal cu "secventa-N".

Producatorul va fi pus in asteptare ori de cate ori numarul elementelor generate ajunge cu "N+1" mai mare decat numarul elementelor consumate (extrase din buffer).

In mod simetric *consumatorul* va fi pus in asteptare prin **await(in,secventa)**, daca numarul elementelor depuse in buffer (in) nu este cel putin egal cu numarul elementelor extrase - 1.

In acest exemplu apelul sistem **read** nu este utilizat.

3.6.7. Monitoare

Utilizand *semafoarele* sau *contoarele de evenimente* se poate asigura o comunicare intre procese sigura aparent si usor de realizat. Totusi programarea in aceste cazuri trebuie facuta cu mare atentie. Lucruri aparent fara prea multa importanta putand conduce la asa numitul "dead lock" (inter blocare) in care ambele procese care utilizeaza resurse partajate raman blocate la infinit. De ex,: inversarea apelurilor **down**(&gol) cu **down**(&exclud_mutual) din procedura *producator* pot conduce la o asemenea situatie (atunci cand buferul este plin si deci variabila gol=0).
Metoda *monitoarelor* implica existenta unei colectii de primitive de nivel inalt pentru sincronizare. *Monitorul* este o colectie de proceduri, variabile si structuri de date grupate impreuna intr-un tip special de modul. *Monitoarele* sunt constructii ale limbajului de programare pe care compilatorul le trateaza special in ceea ce priveste apelurile la astfel de proceduri. Numai un singur proces poate fi activ in *monitor* la un moment dat.
Atunci cand un proces apeleaza o procedura de tip monitor, primele instructiuni ale procedurii vor verifica daca alt proces este activ intr-o procedura de tip monitor. Daca **da** ultimul proces care a apelat monitorul este blocat pana cand celalalt proces paraseste procedura *monitor*. Revine in sarcina compilatorului de a implementa "excluderea mutuala" intre punctele de intrare in monitor, ceea ce face mai putin posibil erorile de programare.
Programatorul nu se mai ingrijeste de cum se asigura "excluderea mutuala" acest lucru revenind in sarcina compilatorului fiind suficient sa stie ca nu va fi posibil niciodata ca doua procese sa execute sectiiunile lor critice in acelasi timp.
Deci *monitoarele* asigura excluderea mutuala dar acest lucru nu este suficient fiind necesare deasemeni de mijloace prin care procesele sa fie blocate atunci cand nu pot sa lucreze (sincronizare). In problema *producator-consumator* aceasta revine in a putea bloca procesul atunci cand bufferul este plin (in cazul procedurii *producator*) respectiv buffer gol (in cazul procedurii *consumator*).
Pentru acest lucru se introduce un nou tip de variabila "variabila de conditie" si asociat cu acest tip, doua tipuri de operatii:

- **wait** (*variabila conditie*): determina blocarea procesului.
- **signal**(*variabila conditie*): deblocheaza procesul blocat pe variabila de conditie respectiva.

Operatiile **wait** si **signal** sunt similare cu SLEEP si WAKEUP, cu deosebirea fundamentala ca nu se poate intampla sa se piarda nici-un **signal** asa cum se intampla cu WAKEUP deoarece nu este posibil ca un proces sa fie intrerupt (de catre planificator) intr-un WAIT neterminat (*monitorul* asigura excluderea mutuala).
Monitorul este un concept la nivelul limbajelor de programare implementat in *compilator.* La ora actuala sunt putine limbaje de programare care implementeaza acest concept. Un asemenea limbaj este Concurent Euclid (Holt,1983).
Ca si in cazul *semafoarelor,* SO pe durata accesului la variabilele de *conditie* a unui proces va bloca accesul altui proces la aceleasi variabile cu ajutorul unei instructiuni TSL. In acest fel se evita conditiile de competitie.
Deoarece in cazul monitoarelor excluderea mutuala pentru sectiunile critice se face automat, programarea devine mai simpla probabilitatea aparitiei erorilor de programare fiind mai mica.

Monitor Producator_consumator (pseudo cod sau Pascal)

```
        condition plin,gol;
        integer contor;
        procedure introduce;
                begin
                        if contor = N then wait (plin);
                        introduce_elemnt;
                        contor:=contor+1;
                        if contor = 1 then signal (gol);
                end;
        procedure extrage;
                begin
                        if contor = 0 then wait (gol);
                        extrage_elemnt;
                        contor:=contor-1;
                        if contor = N-1 then signal (plin);
                end;
contor:=0;
end monitor;
```

```
        procedure producator;
        begin
                while true do
                begin
                  produce_element;
                  producator_consumator.introduce;
                end
        end;
```

```
        procedure consumator;
        begin
                while true do
                begin
                  producator_consumator.extrage;
                  utilizeaza_element;
                end
        end;
```

Concluzii.

Limbaje de prgramare ca C sau Pascal nu au implementat conceptul de *monitor* dar nici conceptul de *semafor*. Cu toate acestea mecanismul *semafor* este usor de implementat in asemenea limbaje prin niste rutine foarte simple scrise in asamblare. Singura coditie este ca SO sa suporte acest concept adica sa exista in biblioteca sistemului apeluri sistem pentru manipularea semafoarelor.

Atat metoda *monitoarele* cat si cea a *semafoarelor* sunt metode proiectate pentru rezolvarea problemelor de "excludere mutuala" pentru SC cu unul sau mai multe procesoare. Adica aceste metode sunt utilizabile in SC care au singura unitate de memeorie operativa. Acest lucru se datoreaza faptului ca numai in asemenea sisteme este posibila folosirea instructiunile TSL pentru "protejarea" variabilelor *semafor* sau de *conditie*.

In SC distribuite care constau din multiple CPU-uri (procesoare), fiecare cu propria sa memorie, conectate prin *retele*, aceste concepte nu mai sunt aplicabile.

Niciuna din aceste metode nu furnizeaza un mecanism de schimb de informatii intre SC diferite(sisteme distribuite).

3.6.8 Transmiterea mesajelor.

Reprezinta o metoda de comunicare intre procese (IPC) utilizand doua primitive **send** si **receive** care sunt apeluri sistem (ca si semafoarele).In cazul "transmiterii mesajelor" apar probleme specifice iar cerintele de proiectare pentru utilizarea acestui mecanism vor fi altele decat in cazul semafoarelor sau contoarelor de evenimente.
Prmitivele puse la dispozitie de SO :

send(destinatie,&mesaj); si
receive(sursa,&mesaj);

realizeaza transmiterea (**send**) unor *mesaje* (un sir de informatii indiferent reprezentarea) de la un proces numit *sursa* la un proces (sau mai multe) numit *destinatie*, respectiv receptionarea (**receive**) de catre procesul *destinatie* a unui mesaj trimis de un proces *sursa*.
Daca in momentul executiei de catre un proces a apelului sitem **receive** nu exista un *mesaj* pentru procesul respectiv, procesul se blocheaza pana la sosirea unui mesaj adresat lui.

3.6.8.1 Elemente de proiectare pentru sistemul de transmitere mesajelor.

Utilizarea mesajelor in IPC, in mod special atunci cand procesele sunt pe SC diferite conectate prin retea, creaza alte probleme decat in cazul altor metode de IPC. Iata cateva probleme specifice care pot aparea in cazul IPC utilizand *mesaje*.

1. Trebuie identificate mesajelor pierdute (mesajele care nu ajung la destinatie).

Pentru a se evita pierderea mesajelor trimise de la un proces la altul se poate imagina un sistem de confirmare a receptionarii unui mesaj. Receptorul va trimite imediat dupa receptionarea unui mesaj, un mesaj de achitare(aknowledge) (confirmare), cu scopul de a informa *sursa* ca a primit *mesajul*.
Este posibil insa ca acest mesaj de confirmare (achitatre) sa se piarda la randul sau, ceeace ar determina sursa sa retransmita mesajul. In acest fel poate aparea o dublare a mesajului primit de un proces *destinatie*. Pentru eliminarea acestei situatii se poate adauga la mesaj un numar de secventa care va permite la receptie eliminarea mesajelor multiple. Aceasta confirmare respectiv adaugarea numarului de secventa mesajului este transparenta pentru utilizator ea realizandu-se la nivelul "apelului sistem".

2. Numirea (identificarea) proceselor trebuie sa fie unica si neambigua.

Pentru o identificare unica si neambigua se aplica o regula de de construirea numelui. De exemplu o schema de atribuire a numelui unui proces des utilizata este concatenarea numelui procesului cu numele SC-ului si eventual un "domeniu" din care face parte SC-ul.

- nume_proces@nume_masina
- nume_proces@nume_masina.domeniu

Aceasta schema este simpla si asigura "unicitatea" identificarii proceselor in sistemele distribuite.

3. Trebuie asigurata autenticitatea mesajelor.

Foarte important in IPC cu schimb de mesaje este asigurarea "securitatii" mecanismului.
De exemplu cum poate un SC SERVER sa fie sigur ca o anumita cerere (de ex de transfer a unui fisier) primita de la un proces, este chiar de la acel proces (cu un anumit nume) si nu de la un "impostor". Si invers, cum poate un proces CLIENT sa stie ca informatiile primite sunt chiar de la un real SC SERVER. Adica la receptionarea unui mesaj sa fim siguri ca el nu a fost trimis de la un proces care printr-un mijloc oarecare nu a facut o :"substituire" de identitate.
Acest lucru poate fi evitat prin mecanisme de "criptare" a mesajelor utilizand o "cheie" de criptare cunoscuta numai de utilizatorii autorizati.

4. Transmiterea mesajelor este lenta.

Copierea unui mesaj de la un proces la altul este intotdeauna mai putin rapida decat oricare din metodele anterioare.

Stiind acest lucru trebuie avut grija sa facem limitarea lungimii mesajelor. Altfel spus sa se incerce intodeauna ca lungimea mesajelor sa fie cat mai mica posibil.

3.6.8.2 Problema Producator-consumator cu "pasarea" mesajelor

In continuare vom da o solutie de rezolvare a problemei producator-consumator utilizand functiile **send** si **receive**.

Presupunem cateva conditii indeplinite:

toate mesajele au aceiasi lungime

mesajele trimise dar neprimite inca de receptor vor fi stocate automat de SO

utilizarea unei noi structuri de date numite "mailbox (casuta postala)" care poate stoca un numar (specificat) de mesaje.

In acest caz destinatia/receptia inseamna numele *mailbox* (nu al procesului). Utilizarea *mailbox*-ului elimina neajunsurile in cazul in care producatorul si consumatorul au viteze de lucru diferite. Un mesaj trimis la o destinatie este pastrat in *mailbox* pana cand destinatarul il accepta.

Mailbox-ul este locul in care sistemul pastreaza mesajele trimise prin send. Numarul mesajelor pastrate in *mailbox* se stabileste la crearea lui. Atunci cand se folseste transmiterea mesajelor utilizand *mailbox*-ul *send* si *receive* vor avea ca parametru numele *mailbox*-ului. Atunci cand un proces va trimite un *mesaj* catre un *mailbox* care este plin, procesul va fi suspendat pana cand va fi extras (de catre alt proces) un *mesaj* din *mailbox*-ul respectiv.

In sistemul UNIX pentru IPC se folosesc mecanismele de "pipe" care sunt aproape identice cu utilizarea "mail-boxului ". Singura deosebire este ca mecanismul "pipe" nu furnizeaza o delimitare intre mesajele trimise. Aceasta inseamna ca utilizatorul *destinatie* va primi un "sir" de informatii fara ca sistemul sa-i furnizeze explicit de unde pana unde tine un anume mesaj. Bineanteles ca procesul are posibilitatea sa-si faca singur "separarea" mesajelor daca procesul sursa trimite si lungimea fiaecarui mesaj.

Rezolvarea problemei Producator-consumator cu *mesaje* se face prin trimiterea de catre procesul consumator a "N" mesaje goale pe care producatorul care le completeaza cu date si le retransmite consumatorului. Fiecare dintre procese are propriul *mailbox*. Producatorul va trimite mesaje catre mailbox-ul consumatorului, iar consumatorul va trimite mesaje catre mailbox-ul producatorului.

Atunci cand *producatorul* are deja produs un "element" are nevoie de un mesaj gol de la *consumator*. Daca mesajul (gol) se afla deja in mailbox atunci il completeaza si-l retransmite consumatorului. Daca nu exista niciun mesaj (gol) in mailbox, asteapta unul iar atunci cand il primeste il va completa si apoi il va transmite catre mailbox-ul *consumatorului*.

```
#define N 100                                   /*numarul de sloturi in mailbox*/
void producator(void)
{
        int element;
        mesage m;
        while (TRUE) {
                element=produce_element();      /*generare elemnt*/
                receive(consumator,&m);         /*astept un mesaj gol*/
                construiesc_mesaj(&m,element);  /*construiesc mesaj*/
                send(consumator,&m);            /*trimite elemnt la consumator*/
        }
}

void consumator(void)
{
        int element,i;
        mesaj m;
        for (i=0; i<N; i++) send(producator,&m)  /*send N mesaje goale*/
        while(TRUE) {
                receive(producator,&m);         /*primeste mesaj continand elemente*/
                element=extrage_elemnt(&m);     /*extrage element din mesaj*/
                send(producator,&m);            /*trimite inapoi mesaj gol*/
                utilizez_elemnt(element);       /*utilizarea elemntului*/
        }
}
```

Solutia furnizata in acest caz foloseste "bufferarea", adica o zona de memorie (mailbox) pentru stocarea mai multor mesaje la un moment dat.
Se poate imagina si o rezolvare a problemei producator-consumator fara "bufferare" ceeace implica un mod de lucru "strict" intre producator si consumator. Mai explicit, producatorul dupa ce produce un element si lanseaza **send** el va fi blocat pana cand consumatorul lanseaza **receive** moment in care consumatorul preia acest element si invers consumatorul va fi blocat pana cand producatorul furinzeaza un element prin **send**. Aceasta metoda se mai numeste si "rendez-vous", este mai usor de implementat decat prin metaoda cu "bufferarea" mesajelor dar este mai putin flexibila deoarece producatorul si consumatorul sunt fortate sa lucreze in tandem pentru fiecare element.
Transimterea (pasarea) mesajelor este in mod curent utilizata in sitemele cu programare paralela.
Unul din cele mai cunoscute sisteme de transmiterea mesajelor este MPI (Message-Passing Interface) foarte des utilizat in calcule stintifice.

3.6.9. Echivalenta primitivelor

De-a lunngul timpului au fost foarte multe metode de IPC. Unii le-au numit "secventiatoare" altii "serializatoare" etc.
Se constata insa ca cea mai mare parte a metodelor propuse sunt similare una cu alta.
In cele discutate anterior am studiat patru primitive diferite de IPC. Fiecare din ele a castigat mai multi sau mai putini adepti.
Toate aceste metode sunt din punct de vedere semantic echivalente (cel putin pentru SC cu un singur CPU). Utilizand oricare din ele putem construi o alta din ele.

Putem arata o echivalenta esentiala a semafoarelor, monitoarelor si mesajelor.

utilizand Semafoare putem implementa Monitoare si Mesaje
utilizand Mesaje putem implemnta Semafoare si Monitoare
utilizand Monitoare putem implementa Semafoare si mesaje

3.7. Probleme clasice de IPC

Literatura specifica SO abunda in probleme foarte interesante care sunt pe larg discutate si analizate. Unele din cele mai cunoscute probleme sunt:

1. Problema "Dineului Filozofilor".
2. Problema "Cititorilor si Scriitorilor".
3. Problema "Frizerului care doarme".

In general aceste probleme sunt utilizate pentru evaluarea diverselor metode de IPC.

3.8. Planificarea proceselor

Am vazut ca la aparitia unui eveniment (o intrerupere) in SC, SO va analiza procesele (starea lor) si va decide ca unul dintre ele sa fie executat de CPU aceasta insemnand sa i se aloce resurs fizica "procesor".
Partea sistemului de operare care se ocupa de alocarea resursei "procesor" se numeste PLANIFICATOR (SCHEDULER) iar algoritmii utilizati se numesc *"algoritmi de planificare*".
Cei mai simplii algoritmi de planificare au fost folositi in sistemele "batch process-ing "(prelucrare pe loturi). In sistemele de tip multi-utilizator (multi-task interactiv) algoritmii se complica. La ora actuala majoritatea sistemelor lucreaza multi-utilizator fiecare utilizator lucrand interactiv. Vom studia in continuare asemenea algoritmi utilizati in sisteme multi-user interactiv, cele mai frecvente azi.
Planificarea proceselor trebuie sa implementeze politica SO ce trebuie urmata in cazul alocarii resursei "procesor" (UP).
Criteriile urmarite de algoritmii de planificare sunt specifici tipului de SO.
Regulile generale la toate SO sunt:

1. Corectitudinea (echitabilitate). Fiecare proces sa primeasca resurse UP in mod corect (echitabil,drept).
2. Eficienta: realizarea unei ocupari maxime tuturor a resurselor fizice a SC (in principal UP).
3. Respectarea politicii SO: indeplinirea strategiei propuse de catre SO (politica SO).

Regulile specifice pe care se bazeaza strategia SO interactive sunt:

Proportionalitatea: asigurarea unui raspuns proportional cu solicitarile.
Timpul de raspuns: minimizarea timpului de raspuns al proceselor utilizatorilor interactivi.

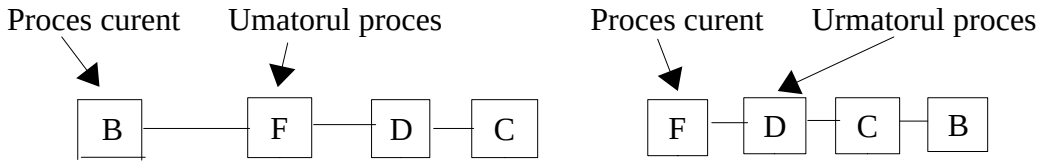
In timp ce regulile specifice in sistemele de tip "Timp-Real" sunt:

Respectarea "termenelor impuse" (deadline): evitarea pierderii datelor.
Predictibilitatea: evitarea degradarii alitatii in sistemele multimedia.

Unele din aceste criterii pot fi "divergente ". De exemplu timpul de raspuns si eficienta sistemului.
Fiecare proces este unic si nepredictibil . Unele procese folosesc intens dispozitivele de I/O mare parte din timp asteptand date de la dispozitivele periferice. Altele utilizeaza intens UC, avand putine operatii de I/O. Atunci cand planificatorul lanseaza in executie un proces, el nu stie niciodata cand procesul se va bloca (pentru operatia de I/O, semafor sau alte motive). Pentru a fi sigur ca un proces nu ruleaza prea mult timp, toate SC dispun de dispozitive electronice de tip ceas (TIMER, CLOCK) care produc intreruperi la perioade fixe de timp (de ex. la 20 ms.). Fie la o intrerupere produsa de un dispozitiv periferic fie la o intrerupere de ceas, SO preia controlul si decide daca procesul care era in executie isi continua executia sau este suspendat temporar alocand *procesorul* altui proces.
Strategia conform careia un proces care din punct de vedere logic poate fi executat este temporar suspendat este numita planificare preemtiva.
In contrast exista strategia planificare nepreemtiva sau executie completa in care un proces odata lansat in executie nu mai este suspendat in mod "voluntar" atunci cand el poate fi in executie.
In planificarea preemtiva un proces poate fi suspendat intr-un moment arbitrar, fara nici-un avertisment astfel ca un alt proces poate fi executat. Aceast mod de lucru al SO creeaza posibilitatea de aparitie a *conditiilor de competitie.* Acest mod de lucru a determinat introducera semafoarelor, contoare de evenimente, monitoare, mesaje sau alte metode pentru evitatarea *conditiilor de competitie.* Pe de alta parte politica de a "lasa" un proces sa ruleze oricat de mult, poate conduce la o situatie de nedorit, aceea ca alte procese sa astepte foarte mult timp (timp nedefinit) pana sa capete resursa *procesor*. Chiar daca planificarea nepreemtiva este simpla si usor de implemntat ea nu este eficienta conducand la o utilizare nerationala a SC.

3.8.1 Planificarea ROUND-ROBIN.

Conform acestui algoritm fiecarui proces ii este permis sa "ruleze" o *cuanta* de timp prestabilita. Daca procesul ruleaza inca la expirarea acestei *cuante* de timp procesorul este alocat in continuare unui alt proces. Daca procesul trece in starea *blocat* inainte de expirarea *cuantei* deasemeni procesorul va fi alocat unui alt proces.
Algoritmul ROUND-ROBIN este usor de implementat. Planificatorul administreaza o lista a proceselor aflate in starea "GATA ".



Atunci cand *cuanta* alocata procesului "B" expira, procesul este suspendat si trecut in coada listei, *procesorul* fiind atribuit urmatorului proces din lista.
Problema care se pune in cazul acestui algoritm este stabilirea cat mai judicios a lungimii *cuantei* de timp. Acest lucru este foarte important pentru ca are efecte majore asupra eficientei utilizarii SC. Comutarea de la un proces la altul consuma un timp legat de administrare,salvarea, restaurarea contextului procesului si actualizarea tabelelor SO. Acest timp il vom numi timp de comutare context. Daca de ex. acest timp este de cca 5 msec si daca cuanta de timp ar fi 20 ms, timpul de comutare context ar reprezinta 20% , ceea ce este considerata o utilizare ineficienta a SC. La cealalta extrema, daca alegem o cuanta de 500ms timpul de comutare context reprezinta 1% ceea ce pare foarte bine. Dar acest lucru conduce la un timp de raspuns foarte lung.
O cuanta potrivita este intre 10ms si 100ms (in functie de viteza *procesorului care* determina in principal timpul de comutare) realizand un compromis rezonabil intre eficienta si timp de raspuns. Pentru sistemele mai rapide se alege o cuanta mai mica pentru ca la acestea si timpul de comutare context este mai mic.

3.8.2 Planificare dupa prioritati.

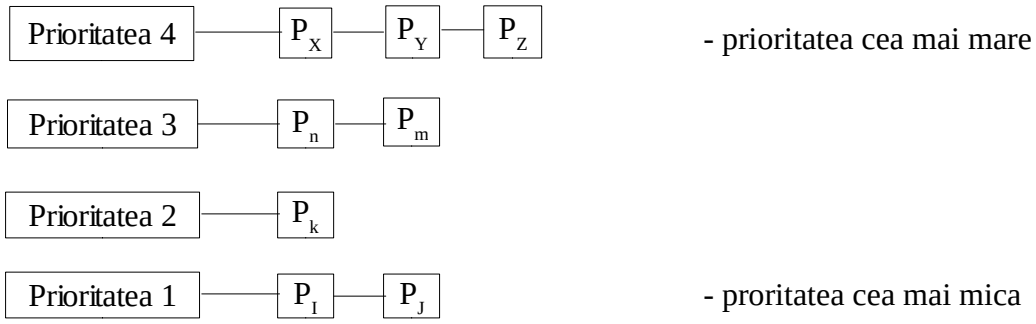
Planifcarea Round-Robin trateaza procesele ca fiind de egala importanta. O alta posibilitate este de a atribui fiecarui proces o anumita prioritate, dupa importanta sa. Planificatorul va lansa de fiecare data procesul cu prioritatea cea mai mare.
Pentru prevenirea situatiei in care procesele cu cea mai mare prioritate ruleaza un timp nedefinit de obicei acest algoritm se completeaza cu un criteriu care tine cont de timpul cat un proces a avut alocat *procesorul*. PLANIFICATORUL poate descreste prioritatea unui proces in executie la fiecare intrerupere de ceas daca acelui proces ii vor fi alocate cuante de timp succesive. Astfel la un moment dat prioritatea procesului in executie poate sa scada sub prioritatea unui proces cu prioritate mai mica ce asteapta sa fie lansat in executie, permitand acestui proces care asteapta sa fie lansat si el in executie. De aceea prioritatile pot fi atribuite proceselor static si dinamic.
Asignarea prioritatilor static se face la lansarea in executie a proceselor, aceste prioritati nefiind modificate ulterior. Asignarea dinamica se face in timpul executiei procesului dupa anumite reguli. Acestea tin cont de cat timp a fost alocat pentru fiecare proces.

Un algoritm simplu de atribuire dinamica a prioritatii este de a atribui fiecarui proces o prioritate egala cu 1/f unde "f" fractiunea de timp din durata cuantei cat a fost rulat un proces in ultima cuanta care i-a fost atribuita.

De ex. daca procesul a fost rulat 2 ms (din 100 ms durata *cuantei*) prioritatea = 50 pentru ca f = 2/100 = 0,2 . Sau daca un proces a beneficiat de intreaga cuanta de 100ms atunci el va capata prioritatea 1 pentru ca f = 100/100= 1.

De multe ori algoritmul de planificare dupa "prioritati" se aplica unor clase de procese, in interiorul unei clase aplicandu-se algoritmul "Round-Robin ".

Procese in starea GATA



Planificare cu 4 clase de prioritati

Atat timp cat exista procese in starea "gata" in clasa de prioritati 4 (Px,Py,Pz), se vor trata aceste procese conform algoritmului "Round-Robin" (pentru procesele din aceiasi clasa). Daca nu exista procese in clase de prioritate 4 se vor rula procesele din clasa 3 (Pn, Pm) cu algoritmul Round-Robin, s.a.m.d.

3.8.3 Planificarea cu cozi multiple.

Acest tip de planificare se poate aplica la SC la care comutarea intre procese este foarte lunga (timp de comutare mare). Acest lucru se intimpla la SC la care nu se poate pastra in memorie la un moment dat decat un singur proces. Fiecare *comutare* presupune transferarea (evacuarea) procesului curent pe hard disc si apoi citirea procesului ce urmeaza la rand, de pe disc si transferarea lui in memorie.La acest tip de algoritm se urmareste minimizarea numarului de evacuari.
Acest tip de algoritm la fiecare analiza a starii proceselor va trece un proces deja rulat dintr-o clasa cu prioritate mai mica intr-o clasa cu prioritate mai mare. Fiecarei clasa de prioritate ii corespunde un anumit numar de cuante alocate succesiv. In acest fel un proces va va fi considerat initial in clasa cea mai mica de prioritati si va primi o cuanta, apoi la a doua alocare va primi 2 cuante corespunzator clasei urmatoare, apoi 4, apoi 8, s.a.m.d in ce in ce mai multe.
De ex. consideram un proces care are nevoie pentru executie de 100 cuante. Initial procesul este introdus in clasa in care este executat o cuanta apoi este evacuat. Urmatoarea data i se aloca 2 cuante succesive si apoi iarasi este evacuat, apoi 4 cuante, apoi 8,16,32 si 64. In ultima clasa el neavand nevoie decat de 37 cuante 1+2+4+8+16+32+37=100. Deci numai 7 evacuari. Intr-o planificare de tip Round-Robin procesul ar fi avut nevoie de 100 evacuari. Procesele pe masura ce avanseaza in clasele de prioritate ele vor fi rulate din ce in ce mai rar.
Multi alti algoritmi folosesc clase de prioritati in functie de tipul de activitate care se desfasoara in cadrul procesului la un moment dat. Procesle vor fi introduse dinamic intr-o clasa sau alta.
Astfel se constituie clasa terminale, clasa I/O disk, clasa cuante scurte si clasa cuante lungi.
Cand un proces, de ex., asteapta un caracter de la terminal (tastatura) el va fi trecut in clasa de

prioritate mare (terminal). Cand procesul asteapta un transfer de pe disc el este trecut in clasa urmatoare cu prioritate mai mica. Atunci cand procesul si-a utilizat cuanta de timp de mai multe ori la rand, fara insa sa treaca in starea blocat (datorata unui I/O cu terminalul sau discul), el va fi transferat mai jos in coada de asteptare.

3.8.4 Primul Job cel mai scurt

Daca se cunoaste in avans durata de executie a unui job este preferabil sa se aplice algoritmul care planifica primul jobul cu durata cea mai scurta.
De ex. Daca la un moment dat sunt active 4 procese a, b, c, d cu duratele:
a=8 cuante, b=4 cuante, c=4 cuante, d=4 cuante.

a	b	c	d
8	4	4	4

Si daca aceste procese se vor rula in ordinea: a, b, c, d,
Timpul total de executie pentru toate procesele va fi:8+(8+4)+(8+4+4)+(8+4+4+4) sau 4a+3b+2c+d iar media per proces: (4a+3b+2c+d)/4 = 14
Deci ponderea cea mai mare la medie o are jobul "a ". Pentru ca media sa fie cat mai mica se alege "a" cel mai mic.

b	c	d	a
4	4	4	8

Timp mediu = 11
Problema la acest tip de algoritm este de a se putea estima, apriori, durata proceselor.

3.8.5 Planificarea garantata

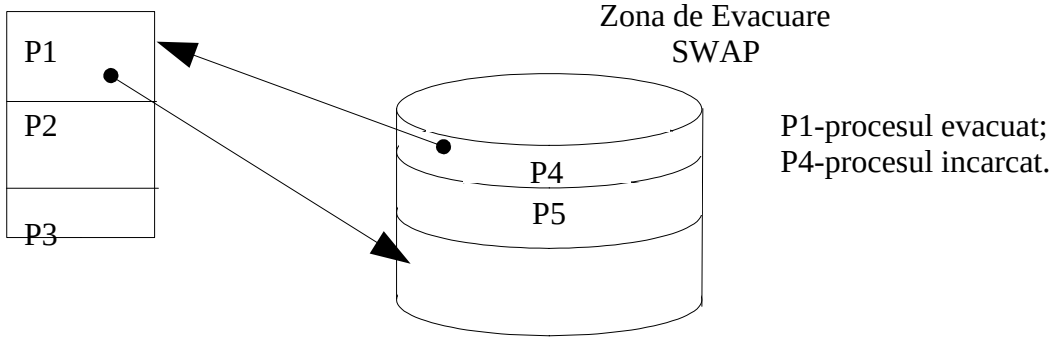
O alta abordare a planificarii proceselor pentru executie este de a se asigura (garanta) pentru fiecare proces o anumita fractiune din partea procesorului (ca timp).
Daca exista de ex. "n" utilizatori care lucreaza cu sistemul la un moment dat, PLANIFICATORUL trebuie asigure ca fiecare proces va primi (aproximativ) 1/n din puterea CPU.
Pentru a realiza acest lucru SO trebuie sa tina evidenta (sa contabilizeze) pentru fiecare proces,a timpului de CPU pe care l-a consumat fiecare proces de la inceputul executiei sale si deasemenea timpul total cand procesul a inceput executia.
Planificatorul calculeaza timpul CPU la care fiecare utilizator are dreptul numit "**timp cuvenit**". Acest timp se calculeaza ca fiind timpul de la deschiderea sesiunii impartit la numarul proceselor "n". Stiind si timpul pe care un utilizator l-a consumat efectiv (real) cu prelucrarile proceselor sale putem sa calculam raportul dintre "timpul real" CPU consumat impartit la "timpul cuvenit".
Daca acest raport este unitar asta inseamna ca acel utilizator se afla "in grafic ".
Daca raportul este subunitar (de ex. 0,5) inseamna ca utilizatorul a primit resursa CPU numai jumatate din timpul cat a avut dreptul iar un raport 2 inseamna ca utilizatorul a primit de doua ori mai mult CPU decat i se cuvenea.
Acest algoritm planifica pentru executie procesele utilizatorilor la care raportul este mai mic pana cand acest raport ajunge mai mare decat a altor procese. Acest algoritm se aplica la sistemele de tip "Timp Real" la care fiecare proces trebuie sa se termine intr-un anumit timp prestabilit (deadline).

3.8.6 Politica si mecanismul planificatorului

Pana acum am presupus ca procesele active la un moment dat in sistem apartin la diferiti utilizatori. Acest lucru este adevarat de multe ori, dar ce se intampla daca un proces ("parinte") lanseaza mai multe procese "fiu" care ruleaza sub controlul lui. Acest lucru se intampla in sisteme de gestiune a bazelor de date unde un proces (principal) lanseaza mai multe procese "fiu" , fiecare din procese "fiu" putand executa o anumita functie specifica: analiza cererilor, acces la disc, etc.
Exista posibilitatea ca fiecare proces fiu sa aiba o anumita importanta si prioritate de care insa planificatorul nu are cunostinta. Planificatorul nu tine cont de faptul ca procesele pot avea fiecare o anumita importanta si va planifica pentru executie procesele intr-o ordine care sa nu fie cea mai buna din punct de vedere al aplicatiei.Solutia la aceasta problema este de a se separa MECANISMUL DE PLANIFICARE de POLITICA DE PLANIFICARE.Aceasta se poate face daca algoritmul de planificare este parametrizat astfel incat parametrii algoritmului sa poata fi fixati de catre procesul utilizator.De ex. daca SO utilizeaza algoritmul de planificare bazat pe prioritati procesele trebuie sa aiba posibiliatea cu ajutorul unor functii sistem sa stabileasca sau sa schimbe prioritatea proceselor fiu. In acest fel procesul "parinte" poate controla in detaliu modul cum vor fi planificate pentru executie procesele sale "fiu ". Acesata posibilitate asigura o adaptare mai buna a activitatii de planificare la cerintele specifice unei aplicatii.
Atunci putem spune ca "Mecanismul planificarii proceselor se afla in SO iar politica este stabilita de procesele utilizator".

Evacuare-reincarcare (SWAPING)

In sistemele cu multiprogramare se poate ajunge foarte des in situatia de a se lansa in executie mai multe programe decat pot fi cuprinse in memoria operativa. Surplusul de programe (procese) se vor incarca pe disc intr-o zona speciala de pe disc numita zona de evacuare (SWAP).



Pentru a putea fi executate procesele trebuie sa se afle in memorie. Atunci cand un proces este planificat pentru executie si nu se afla in memorie se declanseaza un mecanism de transfer a unui a unui proces aflat in memoria operativa (evacuare) si apoi transferul de pe disc in memoria operativa (incarcarea) a procesului ce trebuie lansat in executie.
Mecanismul de transfer a proceselor intre memorie si disc se numeste **evacuare-reincarcare (Swaping).**

3.8.7 Planificarea pe 2 nivele

In cele discutate anterior am presupus ca procesele care erau in starea "gata" se aflau in memoria principala. In cazul in care memoria necesara proceselor active la un moment dat depaseste dimensiunea memoriei operative (principale) o parte din procesele active pot fi pastrte pe disc. Aceasta situatie are implicatii majore asupra "planificarii proceselor" pentru ca incarcarea unui proces in memorie de pe disc dureaza o perioada de timp comparabila cu cuanta de timp alocata executiei unui proces. In aceasta situatie este convenabil sa se utilizeze o planificare pe "doua nivele".
Din totalitatea proceselor active la un moment dat o parte se afla in memoria operativa si o parte pe disc. Periodic "planificatorul de nivel inalt" evacueaza din memorie un proces (cel mai vechi in memorie) transferandu-l pe disc si incarca de pe disc in memorie un proces (cel mai vechi de pe disc).
"Planificatorul de nivel jos" se ocupa numai de submultimea proceselor aflate in memorie pe care le planifica la executie conform unui anumit algoritm.
"Planificarea de nivel inalt" se ocupa numai de procesele active aflate pe disc. Pentru "planificarea de nivel inalt" se pot utiliza diverse informatii pentru a decide care proces este evacuat si care incarcat in memorie.

cat timp s-a scurs de cand un proces a fost evacuat?
cat timp de UC a fost alocat procesului?
cat de mare este procesul?
ce prioritate are procesul?

Si aici se poate utiliza algoritmul "Round-Robin" sau orice alt algoritm discutat anterior.

CAP. IV DEADLOCKS (INTERBLOCARE)

In sistemele cu multiprogramare pot aparea probleme serioase atunci cind doua procese solicita (initiaza) simultan (sau aproape simultan) utilizarea unei aceiasi resurse.
Acest lucru s-a vazut pe larg atunci cind s-a discutat despre "conditiile de competitie".
Putem da si un alt exemplu.
Presupunem ca doua procese care doresc sa tipareasca catre un fisier de dimensiuni mari aflate pe "benzi magnetice"

Procesul A solicita permisiunea sa utilizeze imprimanta si i se acorda acest drept.
Procesul B solicita (cvasisimultan) permisiunea sa utilizeze banda magnetica si capata acest drept.

Mentionam ca ambele periferice , imprimanta si banda magnetica, sunt periferice nepartajabile adica nu pot fi utilizate simultan de doua procese.
In continuare procesul A solicita utilizarea "benzii magnetice" dar cererea este nesatisfacuta pina cind "banda magnetica" nu este eliberata de catre procesul B.
Procesul B va solicita accesul la imprimanta, iar acesta nu-i va fi permis pentru ca este atribuita procesului A. In acest moment ambele procese sunt blocate si nu mai ies niciodata din aceasta stare.
Aceasta situatie este denumita "DEADLOCK" (interblocaj)
"Deadlock"- urile se produc cind unui proces i se atribuie utilizarea, "exclusiva" la resursele fizice sau logice ale sistemului calculator.
Resursele fizice sunt dispozitivele fizice ale S.C.
Resursele logice se refera la anumite informatii, inregistrari in bazele de date, fisiere, etc.
In general numim resurse orice care poate fi utilizat de un proces la un moment dat.
Din alt punct de vedere resursele sunt de doua tipuri:

preemtibile
nepreemtibile

O resursa "preemtibila" este acea resursa care poate fi atribuita unui proces (retrasa altui proces) fara sa se perturbe activitatea S.C.
Memoria este un exemplu de resursa "preemtibila".
Daca consideram un exemplu la care memoria sistemului disponibila utilizatorilor este 512K, sistemul dispunind de o imprimanta. Daca pe acest sistem se executa doua procese de 512K care fiecare au de tiparit pe imprimanta informatii. Procesul A presupunem ca solicita imprimanta si aceasta ii este atribuita, apoi incepe o secventa de calcul a valorilor ce vor fi imprimate.
Inainte ca sa se termine secventa de calcul a valorilor ce trebuiesc tiparite, procesul (A) este oprit de catre "planificator" si evacuat pe disc, deoarece s-a consumat cuanta de timp alocata procesului A.
In continuare este incarcat in memorie si lansat in executie procesul B. Procesul B cere atribuirea imprimantei pentru a tiparii propriile date.
Aceasta a fost insa deja atribuita procesului A. deci procesul B aflat in memorie este blocat pina la eliberarea imprimantei.
Aparent avem o situatie de "deadlock" deoarece procesul A are alocata imprimanta iar procesul B se afla in memorie.
Dar este posibil de a se face o tribuire "preemtiva" a memoriei de la procesul B, prin evacuarea procesului B din memorie pe disc si incarcarea in memorie (atribuirea resursei memorie) a procesului A. Acum procesul poate fi executat, continuind cu prelucrarea datelor si tiparirea lor pe imprimanta, dupa care imprimanta este eliberata putind fi alocata altui proces.

O resursa nepreemtibila este o resursa care nu poate fi retrasa neconditionat procesului caruia i-a fost atribuita fara ca sa se perturbe prelucrarea.
De exemplu daca procesul a inceput tiparirea datelor sale, retragerea imprimantei inainte de terminarea tiparirii lor pentru a o atribui altui proces va avea un rezultat negativ (lista obtinuta fiind incompleta si neutilizabila).
Imprimanta este o resursa nepreeemtibila.
In general "deadlock"-urile apar la utilizarea resurselor nepreemtibile. Deadlock-urile potentiale care pot aparea la utilizarea resurselor "nepreemtibile" pot fi de obicei rezolvate prin realocarea resurselor de la un proces la altul.
Secventa de evenimente ce trebuie sa se execute la utilizarea unei resurse este:

1. Solicitarea resursei si atribuirea ei;
2. Utilizarea resursei;
3. Eliberarea resursei;

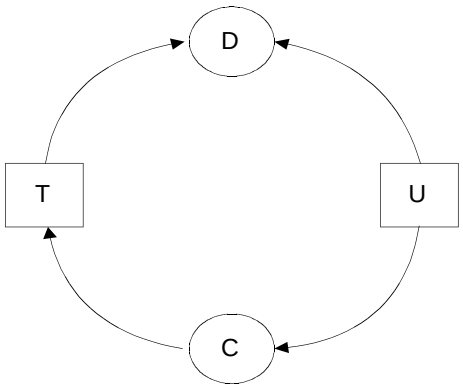
Daca resursa nu este disponibila cind este solicitata, procesul care solicita resursa este pus in asteptare.
In unele S.O procesul este automat blocat atunci cind solicitarea nu poate fi satisfacuta si repornit (trezit) atunci cind resursa devine disponibila. In alte S.O cererile nestatisfacute determina aparitia unei erori (cod de eroare) si procesul va testa periodic disparitia erorii, adica eliberarea resursei solicitate.
Deadlock-ul (interblocarea0 poate fi definit formal astfel:
Un grup de procese sunt interblocate daca fiecare din procesele din acel grup este in asteptarea unui eveniment care nu poate fi produs decit de un singur proces din grup.
Deadlock-urile sunt modelate cu ajutorul grafurilor orientate.
Grafurile au doua tipuri de noduri:

procese (simbolozat cu un cerc)
resurse (simbolizat cu un patrat)

Un arc dinspre un nod *resursa* (patrat) catre un nod *proces* (cerc) semnifica ca resursa a fost deja solicitata si alocata si in prezent este ocupata de proces.

Un arc de la *proces* catre *resursa* semnifica faptul ca procesul este blocat asteptind eliberarea resursei in vederea alocarii ei.

Folosind aceste simbolizari se poate reprezenta o situatie de "deadlock" (interblocare)



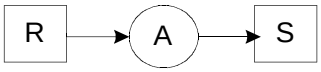
Se remarca un ciclu in care sunt implicate resursele si procesele interblocate.

T,U - Resurse
D,C - Procese

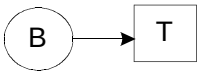
Detectarea "Deadlock-urilor" si eliminarea lor.

Aceasta tehnica porneste de la premiza ca este posibil sa apara in timpul prelucrarii situatii de "deadlock ".(sisteme fault-tolerant)
Deci se porneste de la premiza ca "deadlock"-urile pot aparea si daca apar se incearca detectarea lor si apoi se incearca intreprinderea actiunilor de iesire din aceasta situatie.
Vom alege un exemplu, 7 procese (A G), resurse (R W)
Starea sistemului este urmatoarea:

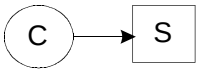
1. Procesul A are alocat R si solicita S;



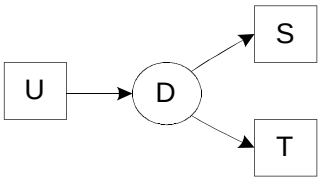
2. Procesul B nu are alocat nimic si solicita T;



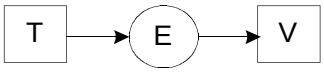
3. Procesul C nu are alocat nimic dar solicita S;



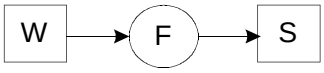
4. Procesul D are alocat U si solicita S si T;



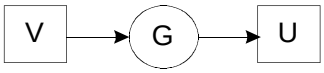
5. Procesul E are alocat T si solicita V;



6. Procesul F are alocat W si solicita S;

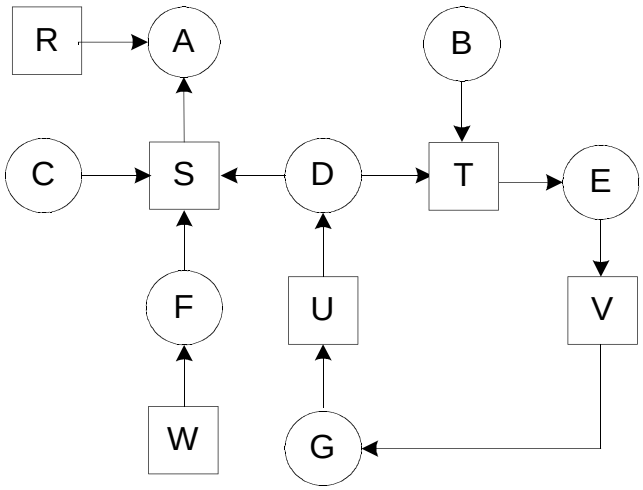


7. Procesul G are alocat V si solicita U;

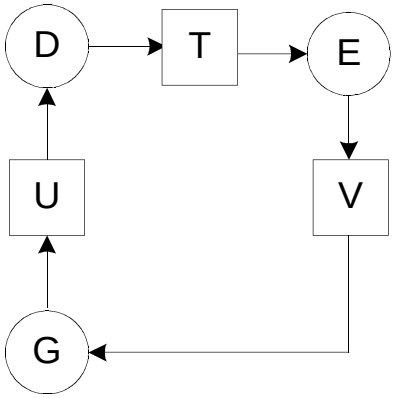


Se pune intrebarea : Sistemul este interblocat (deadlock)?
Daca da, care sunt procesele implicate in interblocare?

Pentru a rezolva aceasta problema vom construi graful resurselor :



In acest graf gasim un ciclu.



Remarcam ca procesele implicate in starea de interblocare sunt D,E si G.
Este relativ simplu de a remarca "vizual" procesele interblocate, dar este necesar un algoritm formal pentru detectarea "interblocarilor". Acest algoritm se bazeaza pe unul din algoritmii de detectare a ciclurilor in grafurile orientate.
Presupunind ca algoritmul nostru de detectare a interblocarii (deadlock) a detectat o interblocare (deadlock).
In continuare este necesar sa elimine aceasta interblocare si de a face sistemul sa functioneze din nou (procesele implicate in interblocare).

Iesirea din starea de interblocare prin PREEMTIUNE

In anumite situatii este posibil de a "retrage" temporar o resursa procesului caruia i-a fost alocata si atribuita altui proces. Acest lucru insa este destul de dificil de realizat si presupune ca este posibila sa se "retraga" o resursa de la un proces si sa se aloce altui proces, si apoi sa se realoce inapoi resursa procesului fara ca acesta sa fie perturbat.
Acest mod de iesire din starea de interblocare este deseori dificil de aplicat.

Iesirea din interblocare prin ROLLBACK (Intoarcere inapoi).

Aceasta metoda se bazeaza pe salvarea periodica a starii proceselor.
Periodic S.O salveaza "contextul procesului" , copiind intr-un fisier toate informatiile legate de proces (imaginea memorie, starea resurselor, etc) astfel incat reconstituind aceasta stare procesul sa poata fi relansat exact din punctul in care s-a salvat contextul procesului.
Pentru a putea fi lansat din oricare moment anterior (salvat), noile salvari ale contextului procesului se fac in fisiere diferite, prezervindu-se toate imaginile anterioare ale procesului.
Atunci cand este detectata starea de interblocare este usor de detectat care resursa este necesara.
Pentru a iesi din starea de interblocare, procesul care are nevoie de resursa care determina interblocarea este derulat inapoi (Rolled back) intr-unul din punctele salvate anterior si reluata executia din acel punct.

Iesirea din interblocare prin "omorarea proceselor".

Cea mai simpla (dar brutala) metoda de iesire din interblocare este de a termina (kill) unul sau mai multe procese care sunt implicate in interblocare.
Una din posibilitati este de a termina fortat (kill) oricare din procesele din ciclu, celelalte procese putand continua activitatea. Daca interblocajul nu se elimina se va termina fortat (kill) un alt proces din ciclu, s.a.m.d. pina cand se elimina interblocajul.
In acest procedeu trebuie ca procesul care se termina fortat sa fie ales cu grija pentru ca el trebuie sa elibereze resursele asteptate de alte procese.
Atunci cand este posibil se "termina fortat" acel proces care elibereaza resursele necesare eliminarii interblocarii si care poate fi reexecutat de la inceput fara efecte nedorite. De exemplu o compilare poate fi reluata oricand fara "pagube" deosebite.

BIBLIOGRAFIE

1. Rus, Theodor - Structuri de Date si Sisteme Operative
   Ed. Academiei 1974
2. Tanenbaum, Andrew S. - Modern Operating Systems. Second Edition
   Prentice Hall 2001
3. Tanenbaum, Andrew S. - Operating System. Design and Implementation. Second Edition
   Prentice Hall 1997
4. Silbershatz, Abraham - Operating System Concepts. 4-th Edition
   Adison Wesley Publishing 1994
5. Musatescu, Carmen - Sisteme de Operare
   Reprografia Universitatii Craiova 1997,1999
6. Kay, Robinson - Practical UNIX Programming
   Prentice Hall 1996
7. King, A Inside Windows 95
   Microsoft Press 1994

CAP.V ADMINISTRAREA MEMORIEI PRINCIPALE

Scopul acestui capitol este descrierea metodelor utilizate de SO pentru administrarea (managementul) memoriei principale a SC.

5.1. Introducere.

Memoria principala numita si *memoie operativa* sau pe scurt *memoria SC* reprezinta subsistemul calculator care indeplineste functia de *conservare a informatiei* in cadrul SC hardware.
In capitolele anterioare am vazut ca *procesorul* (CPU) poate fi partajat (atribuit) mai multor procese active la u moment dat, conform unui algoritm de planificare. Acest mod de lucru are drept scop cresterea a gradului de utilizare CPU si reducerea timpului de raspuns al SC.
Pentru a putea partaja *procesorul* intre mai multe procese active simultan in SC este necesar ca acestea sa se afle in memorie. Acest lucru se poate face daca vom putea pastra si gestiona simultan mai multe procese in memorie. Adica trebuie ca resursa " memorie" sa fie partajata.
Acest lucru conduce la ideea ca este necesar sa existe o componenta a sistemului de operare care sa administreze memoria. Algoritmii (schemele) de management ale memoriei depind foarte mult de caracteristicile hardware al memoriei (caracteristici de proiectare) si de tipul sistemului de operare (scopul caruia ii este destinat SC).
Exista algoritmi de management al memoriei care presupun ca intregul proces trebuie sa se afle in memoria fizica pentru a putea fi executat. Aceasta restrictie determina ca dimensiunea procesului trebuie sa fie mai mica sau cel mult egala cu dimensiunea memoriei fizice disponibile. Exista insa si algoritmi de management al memoriei care nu impun aceasta restrictie fiind suficient existenta in memorie in momentul executiei numei a partii din proces in vecinatatea instructiunii curente.
Indiferent de tipul algoritmului utilizat, dimensiunea memoriei operative a SC influenteaza direct performanta sistemului in ansamblu.

5.2. Fundamente

Memoria sistemului calculator reprezinta o succesiune de locatii (cuvinte, octeti, bytes) fiecare din aceste locatii avand propria adresa. Memoria stocheaza (memoreaza) imaginea programului care nu este altceva decat o succesiune de instructiuni si date pe care o numim *proces*.
Procesorul (CPU) extrage informatii din memorie si depune rezultatul prelucrarii in memorie.
Un ciclu instructiune tipic extrage (fetch) o instructiune din memorie apoi instructiunea este decodificata dupa care poate determina extragerea operanzilor din memorie (in functie de tipul instructiunii). Dupa ce se executa operatia asupra operanzilor rezultatul poate fi stocat (memorat) inapoi in memorie.
Unitatea de memorie prin unitatea sa de control, primeste numai un flux (secventa) de adrese de memorie.
Unitatea de memorie:

- nu este implicata in generarea adreselor (indexare, indirectare, adrese literale etc.);
- nu deosebeste instructiunile de date.

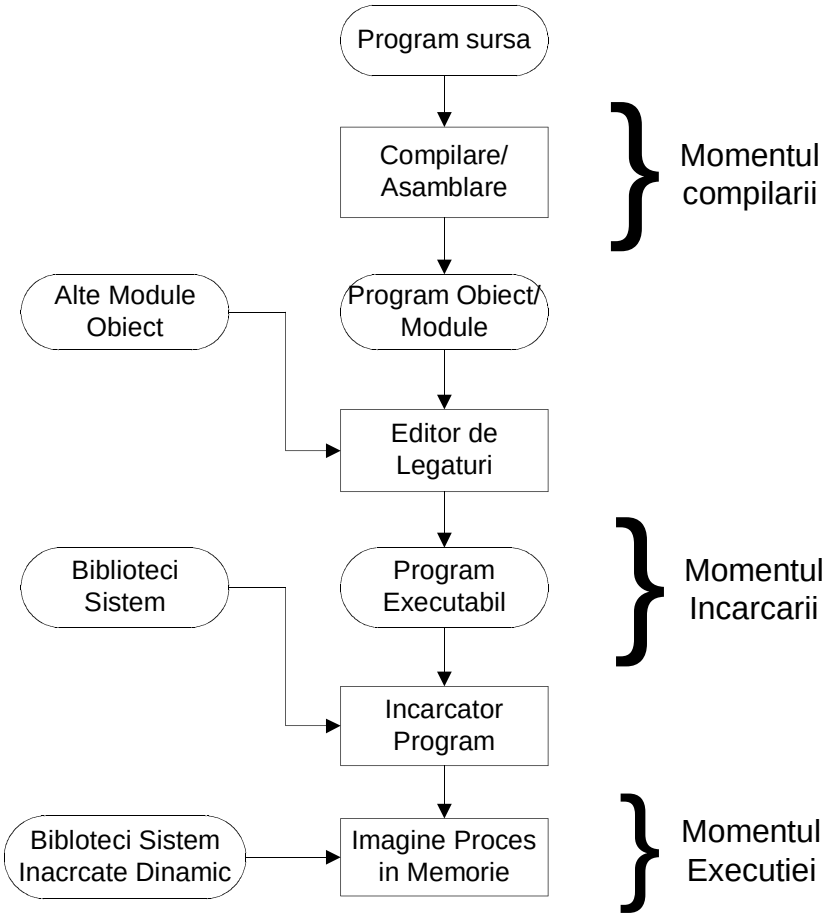
In continuare o sa ignoram modul cum sunt generate adresele de catre program (procesor) si vom fi interesati numai de secventa de adrese de memorie generate de programele in executie (procese).
Spatiul de adrese "virtuale" ocupat de un program pana la incarcarea sa in memorie se numeste *spatiu virtual de adrese* sau *spatiu logic de adrese* iar adresele individuale ale instructiunilor sau datelor se numesc *adrese virtuale* sau *adrese logice*.
Spatiul ocupat de un proces in memoria fizica se numeste *spatiul fizic de adrese* iar adresele, *adrese fizice* (sau absolute) de memorie.

5.2.1. Legarea adreselor.

Acest paragraf descrie modul in care se face asocierea (legarea) adreselor logice de adresele fizice concept gasit in literatura si sub denumirea de legarea instructiunilor si datelor de adresele fizice.
Un proces trebuie sa fie incarcat in memorie pentru a fi executat. De obicei procesele se afla pe disc sub forma de fisiere executabile (programe in format executabil).
Multimea programelor aflate pe disc care asteapta sa fie incarcate in memorie pentru executie formeaza o " coada (lista) de intrare".
Asa cum am mai discutat in capitolele anterioare procedura normala face ca unul din procesele din "coada de intrare" sa fie selectat pentru a fi incarcat in memorie.
Incarcarea in memorie a unui program (proces) poate sa insemne:

relocarea adreselor transformarea adreslor logice in adrese fizice;
legarea referintelor la punctele de intrare ale procedurilor apelate.

In acest fel procesul poate fi executat accesand instructiunile si datele sale pana la terminarea procesului cand spatiul ocupat de proces este declarat disponibil.
Un proces utilizator poate fi incarcat oriunde in memoria fizica. Chiar daca prima adresa a memoriei fizice este 0000 prima adresa a procesului utilizator nu este necesar sa fie 0000.
Vom urmari evolutia adreselor (reprezentarea lor) pe parcursul transformarii unui program de la formatul sursa la formatul executabil.



In cele mai multe cazuri programele sursa (utilizator) parcurg mai multe etape (unele optionale) pana cand devine program executabil.

1. Adresele in programele sursa sunt in general simbolice. Prima faza este compilarea. Compilatorul transforma deobicei aceste adrese in adrese relative si relocabile (de ex. 28 bytes de la inceputul modulului);
2. Editorul de legaturi si incarcatorul vor transforma adresele relocabile in adrese absolute (fizice) (ex. 264020). Aceasta transformare reprezinta o proiectie a unui spatiu de adrese in altul.

"Legarea" instructiunilor si datelor de adresele de memorie fizica se poate petrece oriunde in arborele parcurs de program. De exemplu:

a) In momentul compilarii. Daca se cunoaste din momentul compilarii locul in care va fi incarcat programul in momentul executiei atunci se poate genera direct cod in adrese "absolute".
De exemplu daca se stie din momentul compilarii ca programul (procesul) va fi incarcat in memorie la adresa (de ex, X=2820244) atunci codul (programul) generat de compilator va incepe cu locatia X si se va continua de aici. Daca insa mai tarziu ar trebui incarcat programul la alta adresa Y, programul trebuie recompilat.

b) In Momentul incarcarii programului in memorie. Daca nu cunoastem din momentul compilarii locul in care se incarca programul in memorie, compilatorul va genera cod "relocabil". In acest caz legarea adreselor de instructiuni si date, este aminata pina in momentul incarcarii programelor in memorie. Daca se va schimba adresa de inceput a programului in memorie, atunci este necesar sa se reincarce codul programului.

c) In Momentul executiei. Daca procesul este necesar sa fie mutat in memorie in timpul executiei dintru-un loc in altul, atunci "legarea" se va face in momentul executiei procesului. Pentru aceasta este insa necesar sa existe mecanisme hard speciale care sa faca posibil acest lucru.

5.2.2. Incarcarea dinamica rutinelor.

Pentru a obtine o utilizare mai buna a spatiului de memorie se poate utiliza "incarcarea dinamica". Utilizarea acestei tehnici de "incarcarea dinamica" face ca o rutina sa nu fie incarcata in memorie pina ce ea nu este apelata (folosita). Legarea (asocierea) apelului unei rutine de rutina propriuzisa se face in mod precis inca din momentul *compilarii/editarii de legaturi* a programului. In mod obisnuit la *editarea de legaturi* se face inluderea (incorporarea) rutinelor in formatul executabil (incarcabil) al programului. La incarcarea programului in memorie in mod automat vor fi incarcate in memorie odata cu programul si toate rutinele referite in acesta.
Prin tehnica de *incarcare dinamica* se amana incarcarea rutinei in memorie pana in momentul utilizarii ei.
Toate rutinele sunt pastrate pe disc in format "relocabil". La lansarea unui program in executie el va fi mai intai incarcat in memorie si apoi este lansat in executie. Aceasta incarcare initiala se face fara cuprinderea "rutinelor" cu incarcare dinamica. Atunci cand procesul va executa un apel de rutina mai intai va verifica daca rutina "apelata" se afla in memorie. Daca rutina nu se afla in memorie este activat "incarcatorul" pentru a incarca in memorie si reloca adresele rutinei apelate. In plus se executa o actualizare a tabelelor de simboluri.
Avantajul este ca rutinele neapelate nu sunt pastrate in memorie in mod inutil. Exista o mare probabilitate ca la o anume executie a unui program sa nu fie folosit (apelate) toate rutinele pentru care exista *apeluri* in program.
De exemplu rutinele de tratare a erorilor sunt lansate in executie numai atunci cand apare o eroare ceeace poate sa se intample foarte rar.
Aceasta tehnica nu necesita suport special al SO sau mecanisme hard suplimentare (suport hard).

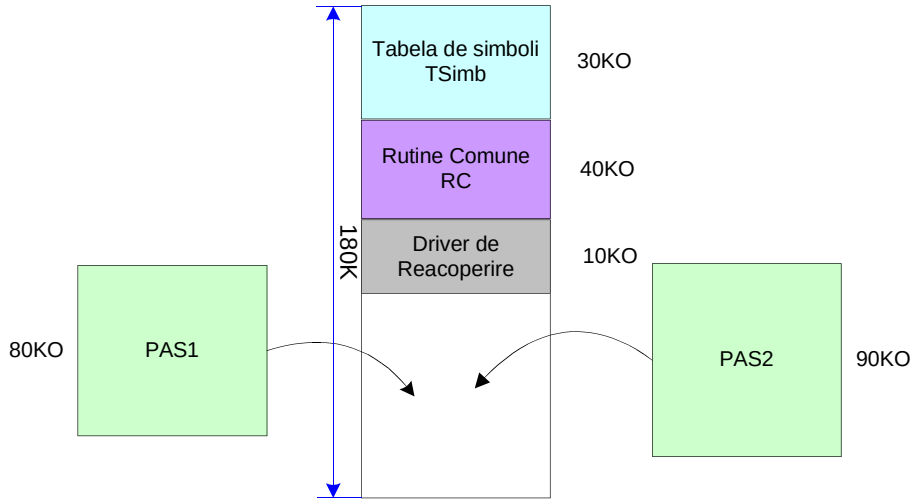
5.2.3. Legarea (asocierea) dinamica a bibliotecilor.

Aceasta *legare dinamica* se refera de obicei la bibliotecile sitemului de operare si ale mediilor de programare (bibliotecile compilatoarelor). Multe SO trateaza bibliotecile sistem ca si orice modul obiect adica sunt incluse in imaginea executabila a programului (legare si incarcare statica a rutinelor).
Conceptul de *legare dinamica* (a bibliotecilor) este similar cu *incarcarea dinamica*. In cazul *legarii dinamice* apelul unei rutine nu va fi legat (asociat) de o anumita rutina pana in momentul executiei apelului. Deci in acest caz va fi amanata legarea apelului unei subrutine de adresa fizica reala pana in momentul executie.
Aceasta tehnica este aplicata bibliotecilor sistem.
Bibliotecile de rutine cu legare dinamica sunt rezidente in memorie ele putind fi partajate (utilizatein comun) de toate procesele care au nevoie de ele. de aceea aceste biblioteci se mai numesc si *biblioteci partajabile*.
Atunci cand se utilizeaza *legarea dinamica*, in imaginea executabila a programului in fiecare loc in care se apleleaza o rutina de acest tip (cu legare dinamica), se plaseaza o secventa de cod de mici dimensiuni *(*numita *jalon*). Acest *jalon* semnaleaza un apel de rutina dintr-o biblioteca cu *legare dinamica.*
La prima executie, atunci cand se intalneste aceasta secventa *jalon*, executia ei va determina inlocirea *jalonuilui* (intreaga secventa) cu un apel la adresa fizica reala de memorie (din acel moment) a rutineia respective. La urmatoarea trecere prin acelasi loc (deci la urmatorul apel) se va face direct apelul la rutina rezidenta. In acest fel se pastreaza o singura copie a rutinelor pentru toate programele ce apeleaza rutina respectiva.
Acest procedeu se poate extinde si in cazul in care anumite biblioteci exista in versiuni diferite sau atunci cand se inlocuieste o biblioteca cu alta, nemafiind necesara recompilarea programelor.
Toate versiunile bibliotecilor se incarca in memorie si fiecare program va utiliza biblioteca cu care a fost compilata.
Sistemul de operare Windows foloseste aceasta tehnica, aceste biblioteci numindu-se *Linked Library* iar modulele de biblioteca de acest tip avand extensia "**DLL**" (**D**ynamic **L**inked **L**ibrary).
Sistemul de operare Unix (sau compatibile) foloseste si el aceasta tehnica, modulele cu aceasta caracteristica avand extensia "**ol**" (**o**bject **s**hared library) iar bibliotecile se numesc *Shared Library*.

5.2.4 Reacoperirea (Overlay)

In toate cazurile discutate anterior am presupus ca un program pentru a fi executat este necesar sa fie incarcat in totatlitate in memorie (cu exceptia rutinelor incarcate sau legate dinamic).
Exista doua modalitati de a se evita aceasta constrangere. Una se refera la situatia cand SC hardware dispune de macanisme mai evoluate de adresare care permit utilizarea memoriei "paginate" sau "segmentate" iar cea de a doua este o modalitate implementata prin mecanisme soft numita "REACOPERIRE" sau "OVERLAY".
In continuare vom explica cum se realizeaza "REACOPERIREA" .
Ideea de baza de la care se porneste este de a se pastra in memorie numai instructiunile si datele de care este nevoie la un moment dat. Atunci cand alte instructiuni si date sunt necesare, ele vor fi incarcate in spatiul ocupat de instructiuni si date de care nu mai este nevoie in acel moment.
Aceasta tehnica presupune insa ca programatorul poate sa "structureze" programul sau in "segmente" de cod sau date. Pentru fiecare din aceste segmente programatorul va stabili daca este necesar permanent in memorie si care sunt segmentele care pot lucra logic independent unul de altul.

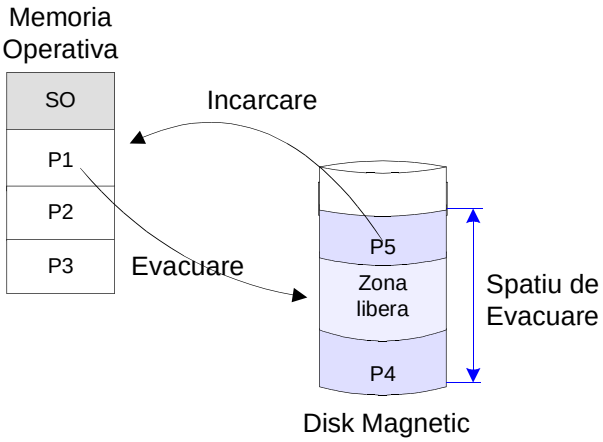
In momentul executiei se vor incarca in memorie la un moment dat numai acele segmente care sunt necesare.
Putem exemplifica aceasta metoda aplicata la un program "asamblor" in doi pasi.
In primul pas "asamblorul" construieste tabela de simboli iar in pasul al doilea va genera codul in limbaj masina. Avand in vedere aceasta derulare a prelucrarii vom impartii acest program in mai multe segmente:
-codul pasului 1 (PAS1);
-codul pasului 2 (PAS2);
-tabela de simboli a programului (TSimb);
-rutine comune (apelate atat in PAS1 cat si in PAS2) (RC);
a caror lungimi presupunem ca sunt: PAS1=80Ko, PAS2=90Ko; TSimb=30Ko; RC=40Ko.
Intr-o preulucare obisnuita , fara "reacoperire" ar fi nevoie ca memoria operativa disponibila sa fie de cel putin 240Ko. Daca memoria disponibila este de mai mica (de ex. numai 180Ko) executia programului ar fi imposibila.
Utilizand tehnica de "reacoperire" vom putea totusi executa acest program intr-un spatiu de memorie disponibil de numai 180Ko. Va fi necesar insa sa existe un mic program numit "driver de reacoperire" (overlay driver) care sa permute intre memorie si discul magnetic segmentele programelor in functie de structura lor atunci cand ele trebuie sa se afle in memorie.
In structura programului asamblor segmentul de cod "PAS1" si "PAS2" sunt logic independenete nefiind necesara prezenta lor simultana in memorie. Bazat pe aceasta caracteristica vom pastra in memorie la un moment dat numai : PAS1, TSimb, RC si in alt moment: PAS2, TSimb, RC.
Altfel spus in memorie se vor afla pe toata durata de executie a programului segmentele TSimb si RC si alternativ PAS1 si PAS2. PAS1 si PAS2 vor folosi alternativ o aceiasi zona de memorie.
Programul "driver de reacoperire" care permuta segmentul PAS1 cu PAS2 va fi prezent intodeauna in memorie.



Aceasta metoda conduce insa la o crestere a duratei de executie a programelor datorita timpului consumat pentru mutarea segmentelor de program care se reaocopera intre disc si memorie.

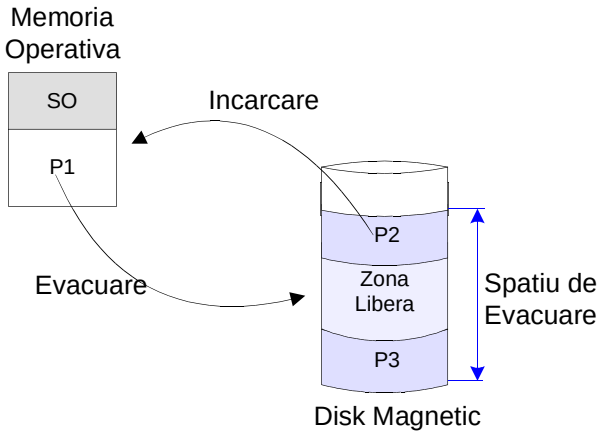
5.3 Evacuarea reincarcarea proceselor (Swaping)

(Swaping = Interschimbare)
Pentru a putea fi executate, procesele trebuie sa se afle in memorie. La un moment dat memoria necesara pentru toate procesele active este mai mare decat capacitatea memoriei operative a SC. In aceasta situatie procesele pot fi "evacuate" temporar pe un suport de stocare (memorie externa de obicei disc magnetic) si apoi "reincarcate" in memorie pentru a-si continua executia.
Putem presupune un exemplu cu un sistem care lucreaza in multiprogramare iar planificarea proceselor pentru executie se face cu un algoritm de tip "round-robin" :
Un singur proces se afla in executie la un moment dat. Dupa expirarea "cuantei" de timp alocata procesului, acesta este trecut de catre planificator in starea "gata", urmand ca un alt proces sa treaca in starea "executie". Daca procesul care urmeaza sa treaca in starea de executie nu se afla in memorie, din lipsa de spatiu, atunci se declanseaza evacuarea- reancarcarea (swaping-ul). Aceasta activitate de "evacuarea-reancarcare" se executa la initiativa "managerul de memorie" printr-un modul numit "dispecer" care copiaza unul dintre procesele aflate in memorie pe suportul de evacuare. Apoi reincarca in memorie de pe suportul de evacuare, procesul care urmeaza pentru executie. Incarcarea procesului in memorie se va face in spatiul eliberat de procesul evacuat.



P1, P2, P3, P4, P5 procesele active.
P1, P2, P3 sunt prezente in memorie.
P4, P5 se afla pe suportul de evacuare.
Atunci cand procesul P5 este planificat pentru executie, "Dispecerul" va evacua pe disc P1 (de ex.) si apoi va incarca in memorie P5.

O situatie particulara apare atunci cand dimensiunea fiecarui proces activ este apropiata (dar mai mica) decat dimensiunea memoriei operative. Aceasta inseamna ca numai un singur proces se poate gasi la un moment dat in memorie. Cand fiecare proces isi consuma cuanta de timp alocata pentru executie, el va fi "inlocuit" in memorie cu urmatorul proces ce trebuie executat.



P1,P2, P3 procesele active.
Numai un singur proces poate fi prezent in memorie la un moment dat.
P1 este prezent in memorie.
P2, P3 se afla pe suportul de evacuare.
Atunci cand P2 este planificat pentru executie, se va decalansa "swaping-ul" lui P1 cu P2.

Ideal ar fi ca "evacuarea reancararea" sa se faca suficient de rapid astfel incat timpul de "asteptare" al CPU pana cand ajunge in memorie urmatorul proces pentru a fi executat sa fie foarte mic.
"Cuanta" de timp alocata proceselor trebuie sa fie suficient de mare astfel incat perioada de timp intre "evacuari- reancarcari" sa fie rezonabila.
O varianta a acestei politici de "evacuare-reincarcare" este utilizata in planificarea proceselor bazata pe "proiritati".
Daca un proces de prioritate mare devine activ, atunci "managerul de memorie" poate evacua din memorie un proces de prioritate mai mica pentru a face loc in memorie pentru a fi executat procesul cu prioritate mai mare. Atunci cand procesul de proiritate mare se termina, procesul de proiritate mai mica evacuat anterior poate fi reancarcat in memorie pentru a-si continua executia. Aceasta varianta de evacuare-reincarcare se mai numeste si "roll-out, roll-in" (dute-vino).
Tehnica de "evacuare-reancarcare" se foloseste in sistemle care lucreaza in multiprogramare pentru a se mari gradul de incarcare al CPU.
Totusi utilizarea swaping-ului creaza cateva probleme.
Aceste probleme sunt legate de:

1) Locul in care se reincarca in memorie un proces evacuat anterior.

Pentru procesele la care relocarea (legarea adreselor fizice de instructiuni si date) s-a facut la compilare sau la incarcare, la evacuare-reancarcare va fi obligatoriu ca ele sa fie reancarcate in memorie la aceiasi adresa. De aceea este mai avantajoasa metoda de legare in timpul executiei permitand reancarcarea procesului oriunde in memorie.
Legat de:

2) Dispozitivul de evacuare. El este carcaterizat de doi parametrii :"capacitate" si "timp de acces" (timp de transfer).

Un spatiu de evacuare mic limiteaza numarul proceselor ce pot fi active simultan in sistemul calculator. Efecte majore asupra performantelor sistemului in ansamblu o au parametrii legati de "timpul de acces" (viteza de transfer). Deobicei "dispozitivul de evacuarea" este un disc magnetic. Unitatea de disc magnetic permite "accesul direct" la informatia stocata pe disc si asigura un debit de transfer de cca 5MB/sec (PIO MOD 1).
Exemplu:
Presupunand ca trebuie evacuat un proces a carui dimensiune este de 500KB, timpul de evacuare necesar transferului intre memorie si discul de evacuare-reancarcare este de cca 100ms iar daca trebuie sa reincarcam in memorie un proces 250KB (de ex.) timpul incarcare va fi de 50ms. La aceasta trebuie sa adaugam si timpul de executie al managerului de memorie care comanda aceste transferuri (timp de regie). Acest timp ar putea fi de exemplu de 5ms. Timpul total consumat pentru executia swaping-ului, in acest exemplu va fi : 100+50+5=155ms.
Acest timp este suficient de mare pentru a putea fi neglijat.
In aceasta situatie este necesara o marire a duratei "cuantei" de timp alocata proceselor de catre algoritmul de planificare al proceselor cu un ordin de marime mai mare. Teoretic durata acestei "cuante " ar trebui sa fie in jur de 1550ms adica cca. 1,5 sec.
Daca sunt active la un moment dat 5 procese (planificare Round Robin), atunci fiecare proces va primi resursa CPU la fiecare 5 X(1550+155) = 5x1705 =8,5sec.
Aceasta valoare este insa foarte mare.
Solutiile eliminarii acestor neajunsuri sunt indreptate catre doua directii:

pe de o parte marirea vitezei de transfer a suportului de swaping adica schimbarea perifericului de swaping cu unul mai performant (daca este posibil);

pe de alta parte marirea dimensiunii memoriei operative astfel incat "evacuarea reancarcarea" sa se faca mai rar si foarte important ar fi sa se faca in paralel cu executia altui proces. Altfel spus aducerea urmatorului proces in memorie pentru executie sa se faca in avans , in paralel cu activitatea altui proces.

Legat de optimizarea "swapingului" se mai poate lua in considerare si un aspect legat de posibilitatea de a se evacua pe disc un proces la exact dimensiunea care este necesara. Adica SO prin managerul de memorie sa "aiba cunostiinta" in fiecare moment de exact dimensiunea memoriei utile a unui proces. Acest lucru se face contorizand exact "cererile de memorie" si "eliberarile de memorie" solicitate de un proces aflat in executie.

3) legat de restrictiile evacuarii-reacarcarii.

In cazul utilizarii mecanismului de evacuare-reancare apar cateva conditii suplimentare care trebuie respectate.

Daca un proces trebuie evacuat , trebuie sa fim siguri ca procesul nu are operatii de I/O in curs. Daca un proces a lansat o operatie de iesire a carei "zone tampon" (I/O Buffers) se afla in spatiul alocat procesului, acest proces nu poate fi evacuat pana la terminarea operatiilor de I/O. Pentru a inlatura aceasta restrictie este necesar ca zonele tampon sa se afle in spatiul alocat SO. In acest fel un proces care a lansat o operatie de I/O poate fi evacuat, operatia I/O desfasurandu-se cu zone aflate in SO care este intotdeauna rezident. Transferul intre zonele tampon aflate in SO si zona de memorie a procesului se pot face atunci cand procesul este reancarcat in memorie.

5.4.Alocare cu o singura partitie (utilizator).

De departe cel mai simplu mod de management al memoriei este fara nici o schema. Altfel spus este posibil sa existe SC fara managementul de memorie. In acest caz utilizatorul are controlul asupra intregului spatiu de memorie. Utilizatorul controleaza si utilizeaza memoria asa cum doreste.
Aceasta solutie este foarte simpla si implica costuri minime SC hardware neavand nevoie de mecanism de adresare complicate. In aceasta situatie nu este nevoie de sistem de operare.
Acest sistem are insa limitari care nu-l fac utilizabil decat in sistemele dedicate unde este nevoie de simplitate si flexibilitate.
Limitarile acestui sistem sunt:

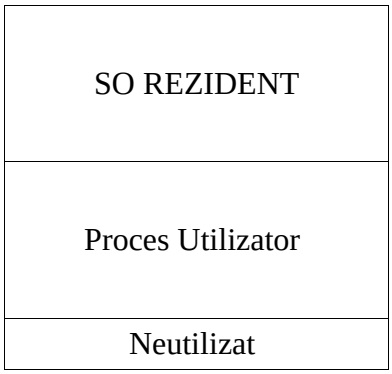
nu exista nici un serviciu oferit utilizatorului (de ex. gestiunea, intreruperilor, operatii de I/O, tratarea erorilor, etc). Utilizatorul trebuie sa-si construiasca rutine pentru toate necesitatile programelor sale.

o performanta scazuta asistemului. Incarcarea CPU este mica.

Dar lipsa sistemului de operare nu este intodeauna posibila. Vom discuta in continuare despre siteme care lucreaza sub controlul sistemului de operare dar exista o singura partitie utilizator.
Cea mai simpla schema pentru managementul memoriei (care implica existenta S.O) este de a se imparti memoria in doua, o partitie pentru procesele utilizator si alta pentru SO (partea rezidenta). In acesta caz se pune intrebarea unde va fi plasat SO.
Este posibil ca SO sa fie plasat:

la adresele mici (incepand cu adresa 0);

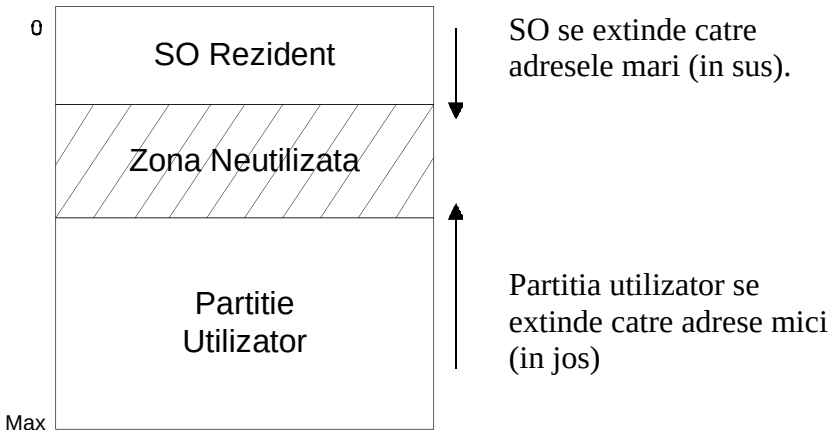
la adrese mari;



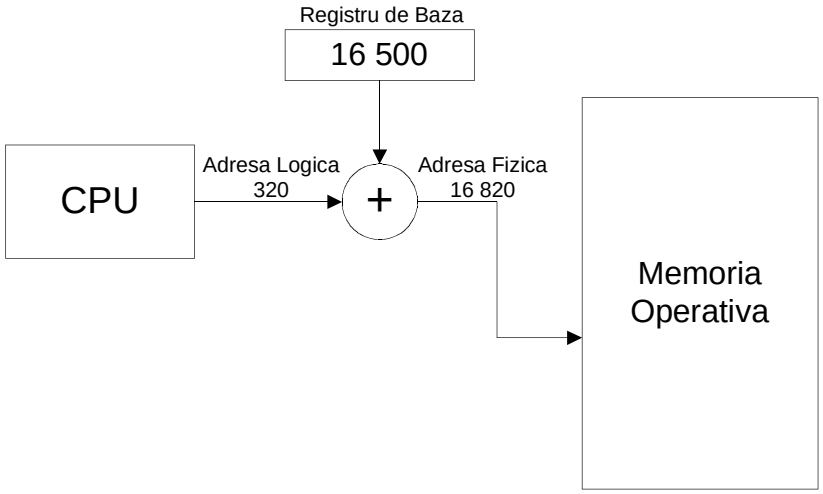
Deobicei adresele vectorilor de intrerupere se afla la adrese mici de memorie si de aceea de cele mai multe ori SO se plaseaza la adrese mici (memoria joasa) pentru a cuprinde si adresele care contin vectorii de intrerupere..
Deoarece procesul utilizator si SO se afla in acelasi timp in memorie, este necesar sa se protejeze spatiul de memorie ocupat de SO impotriva accesului neavizat sau distrugeri accidentale dinspre procesul utilizator. Aceasta protectie se realizeaza deobicei prin mecanisme hardware si se implementeaza utilizand registre de baza si registre limita. Orice adresa utilizator se verifica hard ca ea sa fie cuprinsa intre valorile din aceste doua registre.
O alta problema care trebuie luata in considerare este cum se face incarcarea in memorie a proceselor utilizator.
Spatiul de adrese (fizic) al SC incepe cu adresa 0000 si se termina cu adresa ultimei locatii de memorie fizica. Prima adresa unde se va incarca programului utilizator se numeste "adresa de baza", programul utilizator ocupand adresele de memorie fizica incepand cu aceasta adresa.
Daca adresa de baza este cunoscuta de la compilare, se poate genera un "cod absolut" care va incepe de la aceasta adresa inca din faza de compilare si se va extinde in sus de la aceasta adresa.

Daca se va schimba "adresa de baza" este necesar sa se recompileze programul. Dar compilarea poate genera si "cod relocabil" (asa se intimpla deobicei), legarea de adresele fizice reale amanandu-se pana in momentul incarcarii programului in memorie.
Daca dupa incarcare se schimba "adresa de baza" este necesara o noua "reancarcare" care sa tina cont de aceasta schimbare. Aceste cazuri presupuneau ca "adresa de baza" este fixa (statica) pe perioada de existenta a procesului. Orice modificare a "adresei de baza" in timpul executiei conduce la invalidarea adreselor deja relocate.
In practica insa, pentru o mai buna utilizare a spatiului de memorie, SO isi modifica dimensiunile (este un cod "tranzient"). Acest lucru se face dinamic prin eliberarea spatiului folosit de rutine ale SO care executa anumite "servicii" care in anumite momente nu sunt solicitate. De ex. codul si zonele tampon ale diverselor periferice (driver periferic) atunci cand perifericele respective nu sunt utilizate, elibereaza memoria ocupata. In aceasta situatie dimensiunea "SO rezident" se modifica. Pentru a valorifica spatiul eliberat, respectiv pentru a se putea extinde spatiul ocupat de SO , se pot aplica doua metode:

a) plasarea partitiilor sistem si utilizator la extremitatile spatiului de memorie fizica, spatiul de memorie neutilizat aflandu-se la mijloc intre cele doua partitii. Extinderea putandu-se face de catre ambele partitii in acest spatiu.



b) legarea instructiunilor si datelor de spatiul fizic(dinamic) in timpul executiei programului. Aceasta metoda implica insa sustinerea prin hardware adica existenta unor registre de adresare.
"Registrul de baza" numit si "registrul de relocare" contine adresa de baza.
In momentul adresarii (accesului la memorie) CPU-ul extrage din instructiune o adresa logica pe care o trimite mecanismului de adresare.



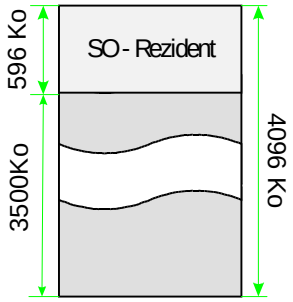
Aici adresa relativa este adunata la continutul "registrului de baza" obtinandu-se adresa fizica reala.
Programele utilizator nu au cunostiinta de adresa fizica reala , ele manipuland "adrese logice". SC hardware prin mecanismul de adresare converteste "adresele logice" in "adrese fizice". Aceasta reprezinta o modalitate de legare de adresele fizice reale in momentul executiei. Locatia finala la care se refera o adresa din program nu este determinata pana cand referinta nu este executata efectiv. Utilizand acest mecanism, schimbarea adresei de inceput a zonei unde se incarca un program implica numai schimbarea continutului "registrului de baza". Aceasta creaza posibilitatea sa se schimbe locul unde se afla un program in memorie (de exemplu ca urmare a "swapingului") in timpul executiei.
Programele utilizator genereaza (utilizeaza) numai adrese logice intr-un spatiu de adrese virtual de la **0** la o adresa: $\mathbf{ADR_{max}}$.
Prin sistemul de adresare acest spatiu este "proiectat" intr-un spatiu de "adrese fizice" in domeniul "**R + 0**" pana la "$\mathbf{R+ ADR_{max}}$", unde "R" este "adresa de baza"
O situatie speciala apare atunci cand in programele utilizator apar referiri la "adrese fizice" cunoscute *apriori*. Acesta este cazul referirilor la registrele periferic. Aceste adrese nu mai trebuie "relocatate" nicicand pe parcursul evolutiei unui program de la compilare la executie.
Rolul *managerului de memorie* este de a "lega" *spatiul de adrese logice* de *spatiul de adrese fizice*. Aceasta "legare" inseamna punerea in coresondenta a adreselor ce compun spatiul logic cu adrese din spatiul fizic.

5.5 Alocarea cu partitii multiple.

In sistemele cu multiprogramare, mai multe procese pot fi simultan in memorie si CPU-ul este comutat rapid de la un proces la altul.
Sarcina "managerului de memorie" este de a aloca memoria necesara proceselor care asteapta sa fie incarcate in memorie.
Cea mai simpla schema de alocare a memoriei (in SC multiprogramare) este de a se imparti memoria intr-un numar de partitii (zone) de dimensiune fixa, fiecare partitie putand contine un singur proces. Gradul de multiprogramare este limitat la numarul de partitii. Cand o partitie este libera se incarca in ea procesul selectat de planificator. Cand un proces se termina spatiul partitiei respective devine disponibil. Aceasta schema nu se mai foloseste la calculatoarele moderne.

5.5.1. Alocarea cu partitii multiple de dimnesiune variabila. Principii de baza.

SO utilizeaza pentru administrarea memoriei o structura de date numiata *tabela de alocare*.
Aceasta *tabela de alocare* a memoriei pastreaza datele privind zonele libere si zonele ocupate. Initial toata memoria (exceptand zona ocupata de partea rezidenta a S.O) este disponibila pentru procesele utilizator si este considerata ca un singur bloc de dimensiune mare de memorie disponibila numit " ni”ª" (Hole).
Atunci cand se lanseaza in executie un program el are nevoie de memorie pentru a fi executat. Managerul de memorie va cauta o " ni”ª" suficient de mare pentru a cuprinde programul.
Daca se gaseste o asemenea " ni”ª" (Hole) se aloca numai atata memorie cat aeste necesar, pastrand restul memoriei disponibila pentru satisfacerea cererilor urmatoare. Vom considera un Exemplu:

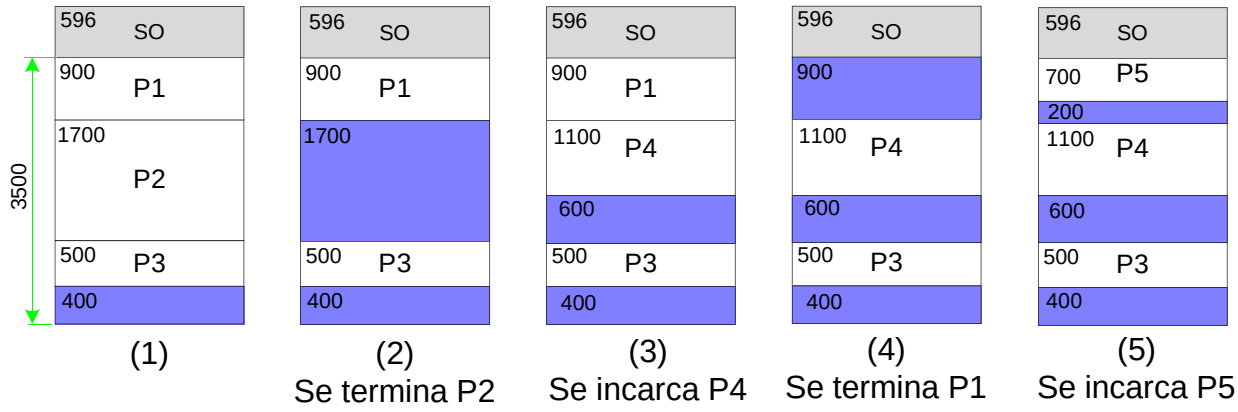


Un SC cu memorie operativa de 4096kocteti, din care este SO rezident ocupa 596KO ramanand spatiu disponibil pentru procese utilizator 3500KO. Aceasta se considera ca fiind starea initiala a sistemului.
Presupunem ca se lanseaza un numar de 5 procese P1, P2, P3, P4, P5 (in aceasta ordine) iar algoritmul de alocare este "primul sosit primul servit" (First Come - First Served-FCFS)

Proces	***Dimensiune (KO)***	***Timp rulare(nr.cuante)***
P1	900	6
P2	1700	4
P3	500	7
P4	1100	5
P5	700	6

Managemetul de memorie poate aloca memorie imediat pentru procesele P1,P2,P3, harta (imaginea)

memoriei fiind cea din primul desen, spatiul de tip "hole" fiind de 400 KO.



Procesele P4 si P5 sunt puse in asteptare neexistand nici un spatiu de tip "HOLE" de dimensiune suficienta pentru ele.
Daca algoritmul de planificare a proceselor este de tip " round-robin" dupa un numar de 11 cuante de planificare (procesor) procesor P2 se termina spatiul ocupat fiind eliberat. (desen 2), aparand inca un " HOLE" de dimensiune 1700KO. Se selecteaza din coada de intrare urmatorul proces si se cauta un spatiu " hole" de dimensiune suficienta. Procesul P4 se incarca in memorie harta de alocare fiind cea din desenul 3. In urma alocarii memoriei pentru procesul P4 apare un nou " hole" de 600KO.
In acest moment (desen 3) exista doua zone disponibile de 600 respectiv 400Ko nici una suficient de mare pentru a satisface procesul P5.
Dupa 16 Quante de timp procesor se termina procesul P1 care elibereaza spatiul ocupat (4). "Managerul de memorie" poate aloca spatiu pentru procesul P5. Harta de alocare a memoriei este cea din desenul (5).
Observatii:

In orice moment exista un numar de spatii libere (Hole) de dimensiuni diferite, dispersate in spatiul de memorie.

Atunci cand soseste un proces care are nevoie de memorie "managerul de memorie" va cauta in multimea spatiilor libere (hole) un spatiu de dimensiune suficienta pentru acest proces.

Daca "HOLE-ul" este de dimensiune mai mare decat a procesului, spatiul "hole" este impartit in doua. O parte este alocata noului proces, cealalta parte fiind inscrisa in lista de spatii libere (Hole);

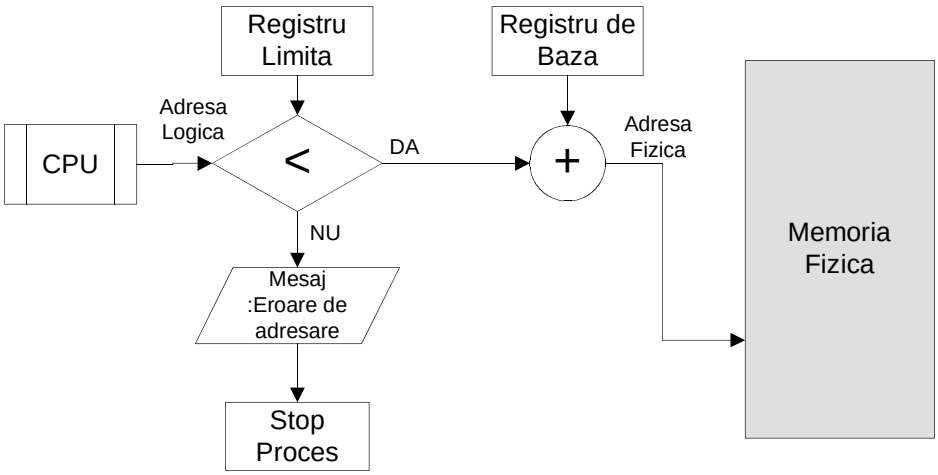
atunci cand se termina un proces se elibereaza spatiul ocupat de proces, plasandu-l in lista de spatii libere (HOLE). Daca noul spatiu liber (HOLE) este adiacent altui spatiu gol (HOLE), aceste spatii vor fi unite formand un singur " HOLE" cu dimensiunea celor doua.

ori de cate ori apare un nou spatiu liber (HOLE) este nevoie sa se verifice mai intai daca exista procese care asteapta sa-i fie alocata memorie si daca da, daca noul HOLE poate satisface aceasta solicitare de memorie.

Toate cele discutate anterior reprezinta o modalitate de alocare dinamica a spatiului de memorie.
In acest tip de aplicatie apare o problema de urmatorul tip:
Cum putem satisface o cerere de alocare pentru un proces a unui spatiu de memorie (compact) de dimensiune "x" dintr-o lista de spatii libere (HOLE).
Exista mai multe solutii pentru rezolvarea acestei probleme daca exista mai multe spatii libere (HOLE) de dimensiune d > x.

1) First-Fit
   Se aloca memorie procesului in primul spatiu liber suficient de mare. Cautarea poate incepe fie de la inceputul listei de spatii libere fie din locul unde s-a terminat cautarea anterioara.
2) Berst-Fit.
   Se aloca memorie procesului in cel mai mic spatiu liber (HOLE) suficient de mare ca sa aiba loc procesul. Acest algoritm presupune parcurgerea intregii liste de spatii libere daca aceasta lista nu este ordonata dupa dimensiune. Aceasta metoda produce cele mai mici spatii ramase in cadrul unui "HOLE".
3) Worst-fit. Se aloca memorie in cel mai mare spatiu liber (HOLE). Si aici trebuie parcursa intreaga lista daca ea nu este ordonata dupa dimensiune. Aceasta strategie produce "resturile" cele mai mari din spatiile libere. Acestea au sanse mai mari de a fi utilizate in alocarile ulterioare.

In urma simularilor s-a ajuns la concluzia ca First-Fit si Best-Fit dau rezultate mai bune in ceeace priveste timpul si gradul de utilizare al memoriei.
Intre Best-Fit si First-Fit nu se poate stabili net ca una din metode este mai buna decit alta in ceeace priveste gradul de utilizare al memoriei dar in general First-Fit este mai rapida.
Aceste metode de alocare a memoriei creaza insa asa numita *fragmentare externa*.
In urma incalcarilor si eliminarilor proceselor in si din memorie spatiul disponibil la un moment dat este fragmentata in mai multe Hole-uri. *Fragmentarea externa* este atunci cand suma tuturor dimensiunilor spatiilor libere individuale (Hole-uri) este suficienta pentru a satisface o cerere de memorie a unui proces dar nu este "continua", spatiul fiind fragmentat. Aceasta poate constitui o problema majora in exploatarea unui SC.
Statistic s-a demonstrat ca pentru metoda First-Fit de ex., chiar in situatia unor optimizari de obicei la un numar de N blocuri alocate, 0,5xN blocuri sunt pierdute (neutilizate) datorita fragmentarii.
Deoarece mai multe procese se afla simultan in memorie trebuie sa existe un mecanism pentru protectia memoriei. Protectia memoriei se refera la impiedicarea "distrugerilor accidentate" a informatiei aflate in memeorie sau impiedicarea "accesul neavizat" la informatia aflata in memorie.
Acest lucru se realizeaza ca si in cazul "alocarii monopartitie" printr-un mecanism cu "registru de baza" si "registru limita". Acest mecanism permite "realocarea dinamica" si protectia in timpul executiei. Registrul de baza , contine valoarea cea mai mica de adresa utilizata de proces, iar registrul limita contine valoarea cea mai mare a adreselor logice a unui proces.



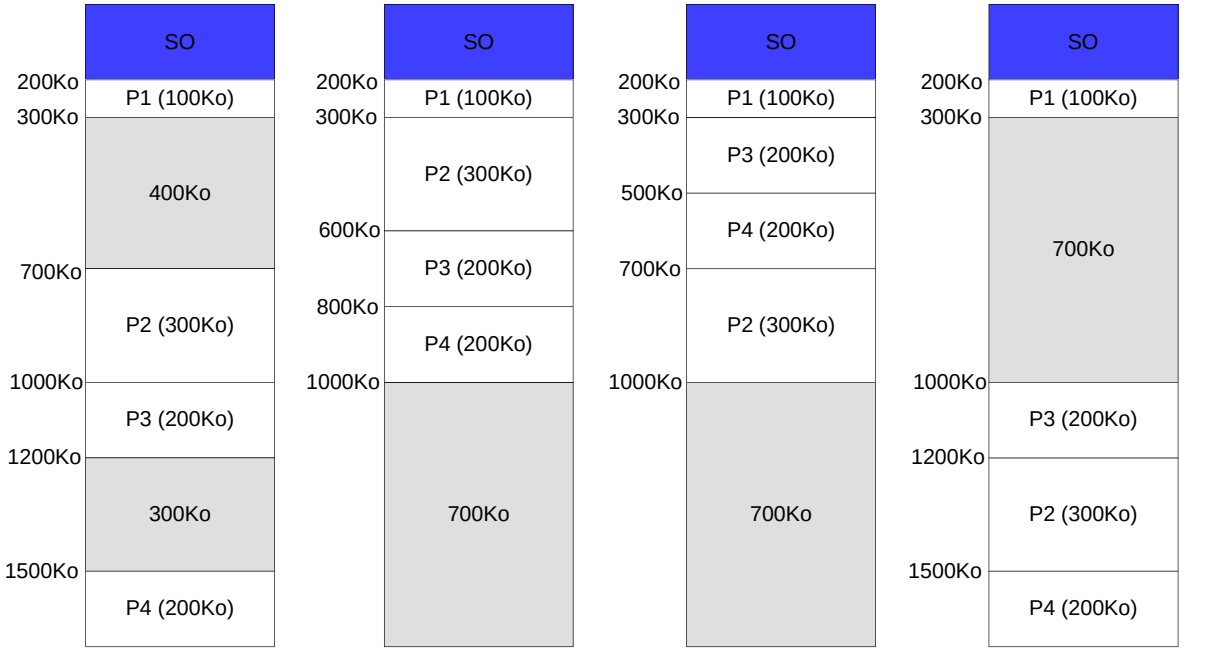
Planificatorul selecteaza urmatorul proces ce trebuie executat. Managerul de Memorie (dispecerul din MM) intretine harta de alocare a memoriei, alege Hole-ul disponibil si incarca adresa de baza in registrul de baza. Din antetul formatului executabil a programului preia lungimea programului(cea mai mare adresa logica) pe care o incarca in Registrul limita.
Pe parcursul executiei procesului fiecare adresa (logica) generata de CPU este verificata de catre mecanismul de adresare. Adresa logica nu poate fi < 0 si > val. registrului limita.
In acest fel se realizaeza o protectie eficienta in timpul executiei procesului astfel incat sa nu fie posibil ca datele si instructiunile altui program sa fie citite sau modificate de procesul in executie.
O alta problema care apare este: Cum se procedeaza in situatia in care dimensiunea procesului care se incarca in memorie este cu putin mai mica decit dimensiunea " HOLE-ului".
Aparent aici nu ar trebui sa fie nici o problema. Luam un exemplu: dimensiunea procesului este de 6452 octeti iar dimensiunea "HOLE-ului" este 6460. Daca se incarca procesul, in memorie ramine un spatiu mic de 8 octeti care formeaza un HOLE care va fi gestionat in continuare de catre "managerul de memorie". Este evident insa ca acest mic HOLE nu va putea fi alocat niciunui proces.
In aceasta situatie este mai eficient sa se aloce procesului intregului Hole (ex. 6460oct). Diferenta intre dimensiunea procesului si a HOLE-ului alocat se numeste " fragmentare interna"

5.5.2 Planificarea proceselor in cazul alocarii dinamice.

Atunci cand un program este lansat in executie el este plasat intr-o coada de asteptare. Planificatorul de procese va compara de fiecare data necesarul de memorie al fiecarui program (proces) cu cantitatea de memorie disponibila in fiecare spatiu (HOLE). Apoi daca va gasi un spatiu (HOLE) potrivit va incarca procesul in memorie alocandui-se spatiul necesar de memorie. Din acest moment procesul va "concura" pentru obtinerea resursei fizice "procesor".
Alocarea resursei procesor va fi facuta de "planificatorul de procese" in concordanta cu algoritmul utilizat. La terminarea procesului se elibereazaa memoria ocupata, ea putand fi atribuita unui alt proces care este in asteptare de memorie.
In fiecare moment planificatorul va avea lista spatiilor libere si dimensiunile lor si lista proceselor in asteptarea resursei "memorie". Lista proceselor este ordonata in concordanta cu algoritmul de planificare. Memoria este atribuita proceselor pina cand necesarul de memorie al urmatorului proces nu poate fi satisfacuta, neexistand nici un spatiu de memorie (HOLE) suficient de mare pentru a incarca procesul. In acest moment "planificatorul" poate astepta pana cand un spatiu suficient de mare de memorie va fi disponibil sau poate parcurge in jos lista proceselor in asteptarea resursei memorie pentru a vedea daca aexista un proces care poate fi incarcat in spatiul liber existent (HOLE-urile existente). Acest tip de decizie poate sa fie acceptat sau nu in algoritmul de planificare (este caracteristic S.O). Deasemeni se poate opta pentru o schema cu *fragmentare interna* sau nu.
O alta problema importanta este *fragmentarea externa*. Daca vom privi desenul (3) din paragraful 5.5.1, vom constata ca in acel moment exista 2 spatii libere (HOLE-uri) unul de 600 si altul de 400 octeti si procesul P5 de lungime 700 in asteptare. Totalul spatiului disponibil este 1000 octeti suficient pentru a rula procesul P5 dar el este fragmentat astfel incat nici unul din HOLE-uri nu poate "primi" procesul.
Problema fragmentarii este deosebit de importanta. In cel mai defavorabil caz putem avea cate un bloc de memorie libera intre fiecare doua procese.
Utilizarea unui algoritm sau altul pentru incarcare (First-Fit, Best-Fit) poate afecta gradul de fragmentare al memoriei, dar acest lucru depinde in mare masura de caracteristicile hard (dimensiunea memoriei operative) si de lungimea medie (statistic)a programelor .
Un alt element care poate fi luat in considerare in schema de alocare este: la care capat al spatiului liber (HOLE) se face alocare? In partea de sus sau in partea de jos?

5.5.3.Compactarea.

O solutie pentru eliminarea *fragmentarii externe* este compactarea. Regula este de a se "translata" continutul memoriei astfel incat sa se plaseze toata memoria libera impreuna formand un singur spatiu (HOLE).
Compactarea nu este intotdeauna posibila. Compactarea este posibila numai daca "relocarea" este dinamica in timpul executiei. Daca adresele sunt "relocate" dinamic mutatrea unui program si a datelor sale este posibila prin schimbarea continutului registrului de baza.



Starea Initiala a memoriei
fig. 1

Memoria compactata mutand P2,P3,P4 (700Ko)
fig.2

Memoria compactata mutand P3 si P4 (400Ko)
fig.3

Memoria compactata mutand numai P2 (200Ko)
fig.4

Asa cum se observa in figura este posibil ca intr-o anumita configuratie de ocupare a memoriei sa existe mai multe variante de compactare.
Cea mai simpla metoda de compactare este de a muta toate procesele catre un capat al memoriei, toate spatiile libere ramanand la celalalt capat (fig.2 si fig.3). Varianta din fig.3 este insa cea mai economica nefiind necesara mutarea decat a 200Ko.
De multe ori insa nu este necesar sa se faca o compactare a intregii memorii, fiind mai avantajos sa se faca o compactare partiala suficienta pentru a " incapea" urmatorul proces in memorie.
Evacuarea-reincarcarea (swaping) poate fi combinata cu compactarea, mutarea unui proces intr-o alta zona putandu-se face prin intermediul procedeului "roll-out roll-in" (dute-vino).
Compactarea memoriei poate fi mare consumatoare de timp, de aceea nu este intodeauna recomandata.

5.5.4.Registre de baza multiple.

Problema principala la partitiile cu dimensiune variabila este *fragmentarea externa*. O cale de a reduce *fragmentareaa externa* este prin fragmentarea memoriei necesare unui proces in mai multe parti. Acest lucru insa trebuie sustinut prin mecanisme hard corespunzatoare.
Este mult mai usor de gasit spatiul liber necesar pentru "fragmente" mai mici de memorie. De obicei se utilizeaza 2 perechi de registre de baza si registru limita. Exista doua metode:

se imparte memoria in doua parti dupa bitul cel mai semnificativ al adresei.
se impart programele in doua parti cod si date fiecare avand perechea de registre baza si limitare corespunzatoare.

In acest fel se aduce un avantaj in plus oferind posibilitatea partajarii programelor de catre mai multi utilizatori.

5.5.5.Concluzii

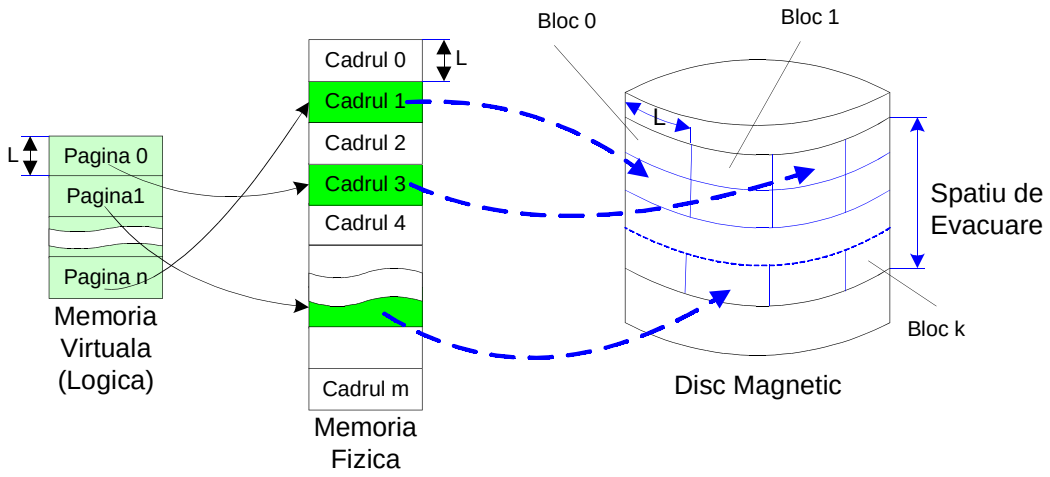
Utilizarea "partitiilor de dimensiune variabila" are dezavantajul *fragmentarii externe.* Acest lucru se intimpla atunci cand memoria disponibila nu este contigua fiind fragmentata in mai multe blocuri (HOLE) de mici dimensiuni, dispersate in spatiul memoriei operative. Deoarece memoria alocata unui proces trebuie sa fie contigua, aceste blocuri "imprastiate" nu pot fi folosite. Rezolvarea acestei probleme se poate face prin "compactare" care insa conduce la timpi suplimentari consumati pentru aceasta operatie. In plus o problema similara apare cu spatiul de evacuare la sistemele care folosesc si evacuarea reancarcarea (Swaping). Spatiul de evacuare devine in timpul activitatii S.C. o succesiune de spatii ocupate alternand cu spatiile libere. In momentul evacuarii unui proces S.O trebuie sa gaseasca un spatiu pe suportul de evacuare (zona de swaping) suficient de mare in care sa aiba loc procesul.
Spre deosebire de memoria operativa, spatiul de evacuare-reancarcare (swaping) nu se poate "compacta" decat "off line" datorita timpului mare al acestei operatii.

5.6.PAGINAREA

O alta rezolvare a *fragmentarii externe* se poate face prin eliminarea restrictiei ca unui proces trebuie sa-i fie alocate un spatiu contiguu de memorie.
Aceasta noua metoda numita *paginare* permite unui proces sa fie incarcat in zone "necontigue" de memorie. Principial este asemanatoare cu metoda "registrelor de baza multiple" cu deosebirea ca in cazul paginarii "adresele de baza" sunt mult mai multe. Aceasta metoda trebuie insa sa fie sustinuta de existenta unor mecanisme hard specifice.

5.6.1.Suportul hardware pentru paginare.

Memoria fizica este "vazuta" ca o succesiune de blocuri de dimensiune fixa numite *cadre*. Memoria logica este deasemeni impartita in blocuri de aceiasi dimensiune numite *pagini*. Si bineinteles ca si spatiu de swaping (spatiu de evacuare-reancarcare) este divizat in *blocuri* care au aceiasi dimensiune cu *cadrele* memoriei fizice, respectiv *paginile* memoriei virtuale.



L=dimensiunea *paginii* = dimensiunea *cadrului* = dimensiunea *blocului*
Atunci cand un proces trebuie executat, trebuie sa i se aloce memoria necesara, *paginile* sale putand fiind incarcate in oricare din *cadrele* de memorie fizica libere.
Atunci cand un proces trebuie "evacuat", fiecare *cadru* al procesului este transferat in cate un *bloc* liber din spatiul de *evacuare* (swaping).
Atunci cand un proces trebuie "reancarcat" in memorie, *blocurile* ocupate de proces in spatiul de evacuare vor fi mutate in oricare *cadre* libere de memorie fizica.
Pentru a se implementa acest mecanism fiecare adresa "logica" generata de CPU este "vazuta" ca fiind formata din doua parti:

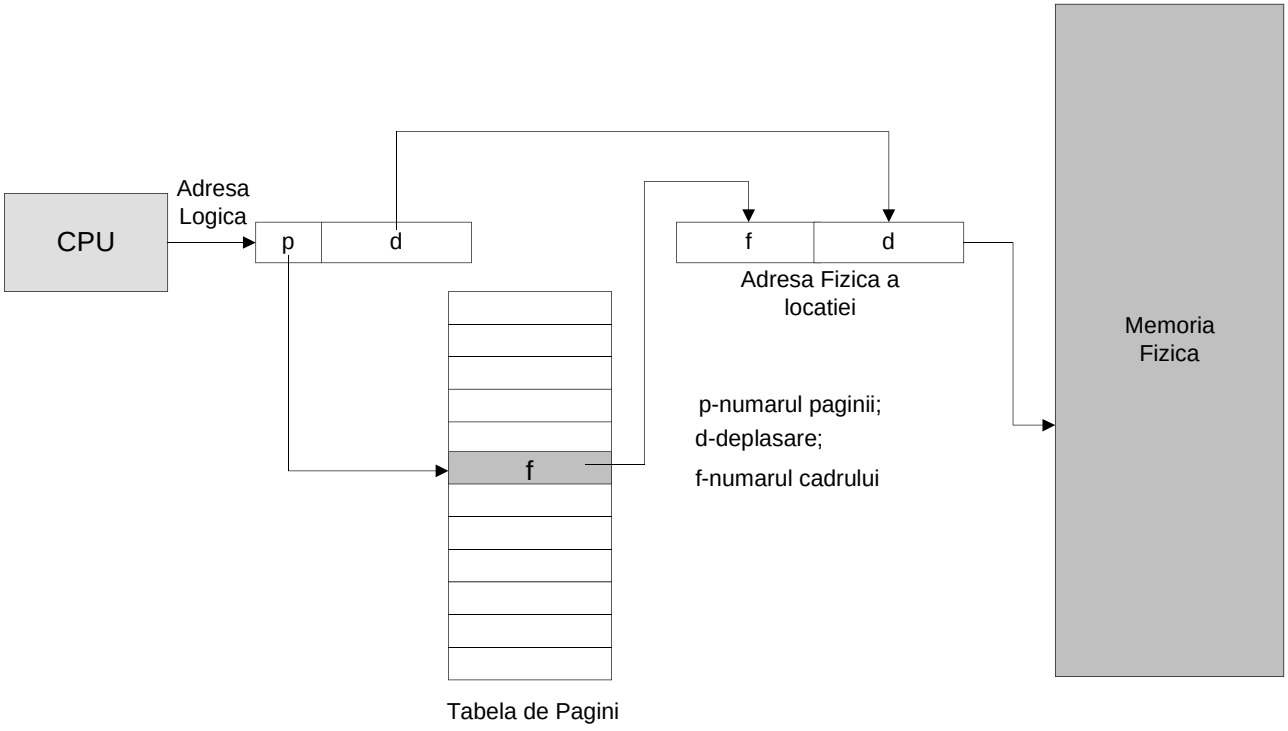
numarul paginii (p)
offset (deplasarea) in cadrul paginii (d)

Adresa logica

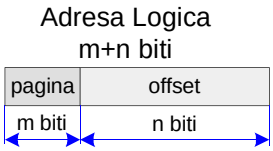
p	d

In plus SO administreaza o structura de date numita *Tabela de Pagini* cu ajutorul careia se calculeaza adresa fizica de memorie a locatiei adresate.

SO la incarcarea unui proces in memoria operativa completeaza *tabela de pagini* a procesului, inscriind in fiecare "intrare" din TP numarul *cadrului* de memorie unde se incarca pagina respectiva in memorie.
Atunci cand se face o adresare la memorie numarul paginii (p) din adresa logica este considerat ca un index in *tabela de pagini*. In intrarea corespunzatoare lui "p" in *Tabela de pagini* se va gasi *numarul cadrului* de memorie unde este plasata pagina in memorie.
De la *numarul cadrului* la *adresa fizica* a cadrului se ajunge prin concatenarea *numarului cadrului* (obtinut din *tabela de pagini)* cu *deplasarea*. Aceasta concatenare este echivalenta cu o inmultire a numarului *cadrului* cu dimensiunea sa la care se aduna *deplasarea*.
Aceasta adresa este trimisa "Unitatii de memorie".

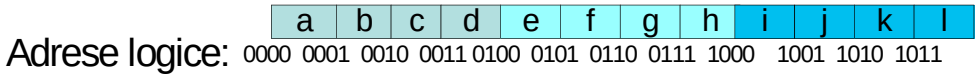


Dimensiunea *paginii* (aceiasi cu a *cadrului* si a *blocului*) este definita prin constructia SC (hardware). De obicei aceasta dimensiune este o putere a lui "2" variind intre 512 cuvinte si 2048cuvinte pe pagina.
S-a ales o putere a lui "2" pentru simplificarea mecanismului de separare a adreselor de pagina de offset. De ex. daca dimensiunea paginii este "2^n" atunci din adresa logica ultimii "n" biti sunt "offset" iar restul (primii biti) desemneaza pagina.

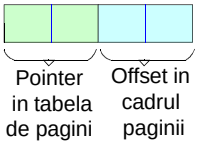


Vom exemplifica functionarea acestui mecanism pe un exemplu la scara redusa. Dimensiunea *paginii* este L = 4=$2^{2.}$ Deci cei mai "nesemnificativi" 2 biti din adresa logica vor desemna offsetul (deplasarea) in cadrul paginii. Ceilalti (mai semnificativi)biti vor "indexa" tabela de pagini unde se va gasi numarul "cadrului" fizic. Adresa fizica a cadrului va fi = "nr. cadru" x 4
Dimensiunea paginii fiind de 4 "locatii", adresele locatiilor in cadrul fiecarei pagini vor fi: 00,01,10,11.

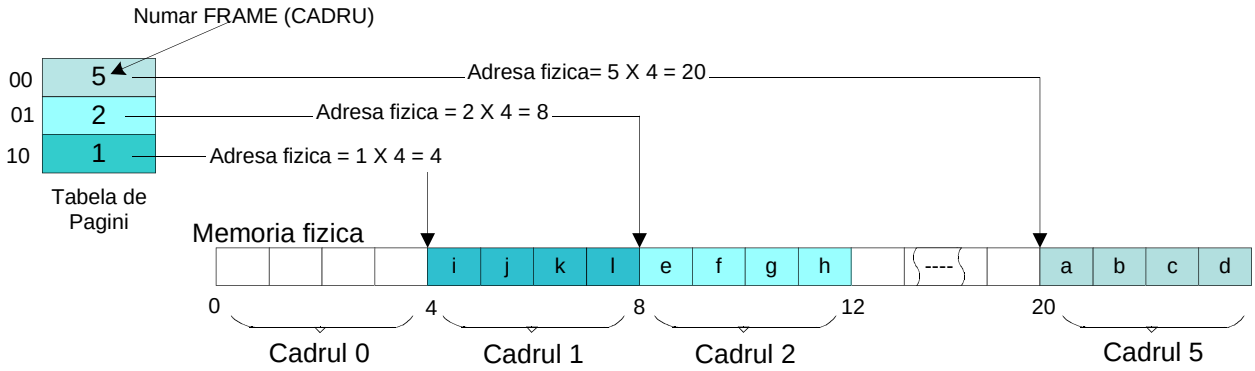
Presupunem ca imaginea programului este urmatoarea:



Deci adresele logice in exemplul nostru sunt de 4 biti, 2 pentru adresa de pagina si 2 pentru deplasare in cadrul paginii.



Spatiul virtual (logic) are maxim 4 pagini. Programul nostru acopera insa numai 3 pagini cu adresele 00, 01, 10 date de primii 2 biti ai adresei logice.



Mecanismul de adresare paginata va folosi primii 2 biti pentru accesul in *tabela de pagini*. Fiecare intrare in *tabela de pagini* va contine numarul *cadrului* in memoria fizica unde se afla incarcata pagina respectiva.
Paginarea in sine reprezinta o forma de relocare dinamica; fiecare adresa logica este "legata" prin mecanismul de paginare dinamic in timpul executiei, de o adresa fizica,.

5.6.2. Planificarea proceselor in cazul alocarii paginate.

Schema de "alocare a memorie" (managementul memoriei) influenteaza modul de planificare al proceselor. Lucrurile in cazul managementului memoriei cu paginare se petrec astfel:

se extrage din antetul programului lungimea acestuia (in nrumar de pagini);

se examineaza "lista de cadre libere a memoriei" si se verifica daca numarul "cadrelor" libere (nealocate) este mai mare sau egal cu numarul de pagini al procesului ce trebuie incarcat in memorie;

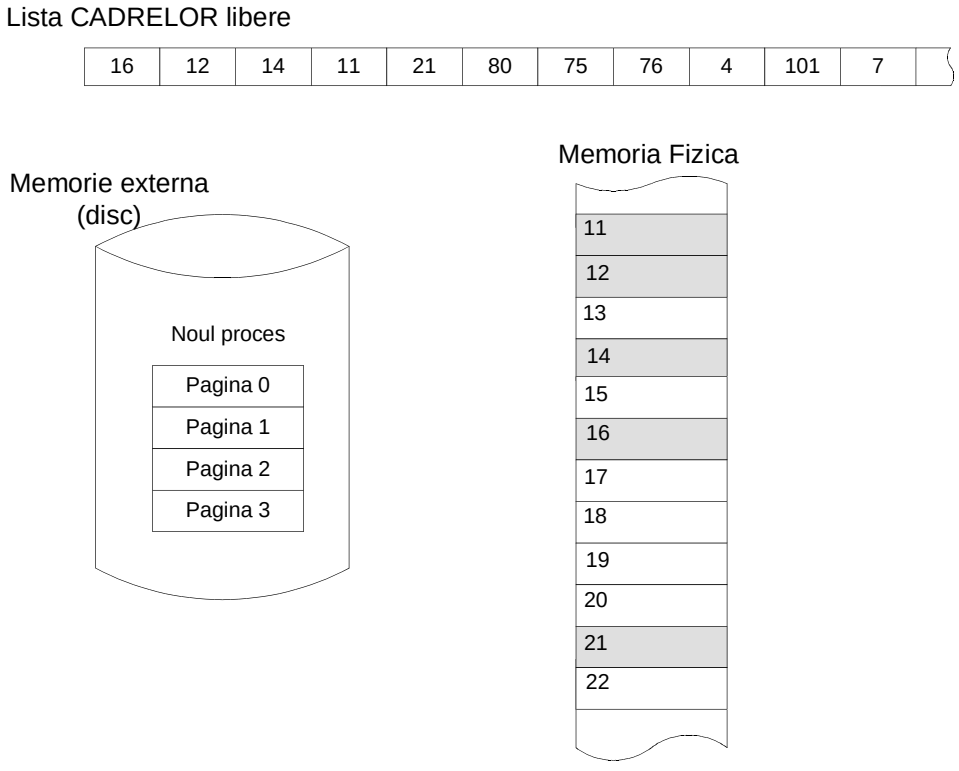
Daca se gaseste un numar de *cadre* libere suficiente atunci pentru fiecare pagina a procesului se aloca un cadru al memoriei fizice.

prima *pagina* a procesului se incarca in primul *cadru* liber din lista de cadre libere a memoriei. Numarul cadrului respectiv se marcheaza ca "ocupat" in "tabela de alocare a memoriei" si se sterge din "lista cadrelor libere". Numarul *cadrului* alocat se incarca in tabela de pagini a procesului. Urmatoarele pagini ale procesului se incarca in urmatoarele *cadre* de memorie libera, marcandu-se "ocupat" in "tabela de ocupare a memoriei" si se sterg din "lista de cadre libere", respectiv se inscriu numerele *cadrelor* alocate in TP a procesului.

Utilizarea alocarii cu *pagini* a memoriei elimina *fragmentarea externa*. Totusi este posibil sa apara *fragmentare interna* datorita faptului ca exista o probabilitate foarte mare ca dimensiunea exacta a procesului sa nu fie un multiplu a dimensiunii paginii. Alocarea memoriei se face la un numar intreg de *cadre* iar ultima pagina a procesului poate sa nu fie intotdeauna completa.
Pentru a reduce *fragmentarea interna* este de preferat ca dimensiunea paginii (cadrului) sa fie cat mai mica.
O dimensiune a *paginii* (*cadrului*) mica conduce insa la o tabela de pagini cu multe intrari ceea ce determina crestere a duratei de calcul al adresei fizice. Deci se incearca alegerea unei dimensiuni a paginii(cadrului, blocului)care asigure un echilibru intre aceste influnte contradictorii.
Fiecare SO implementeaza o anumita metoda de gestionare a *tabelelor de pagini*. De obicei in "Blocul de control" al procesului se afla un pointer catre *tabela de pagini* a procesului respectiv.
Metoda de alocare a memoriei bazata pe pagini trebuie sustinuta prin mecanisme "hardware"

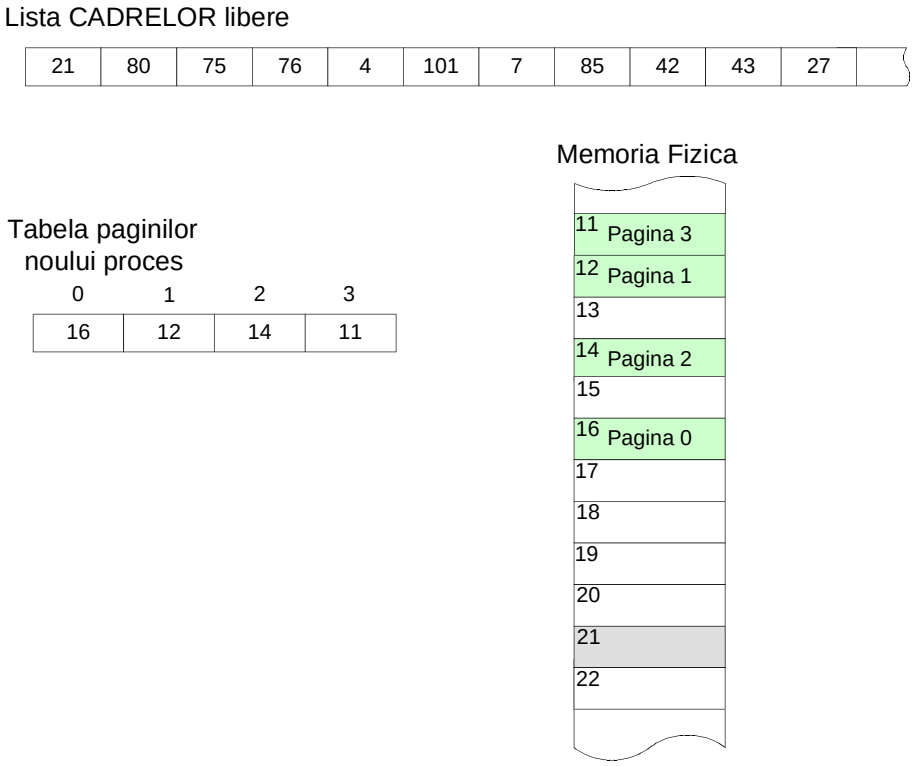
5.6.3.Implementari hardware ale *tabelei de pagini*.

Cea mai simpla metoda de implementare a tabelei de pagini este prin utilizarea unor registre speciale, de mare viteza. Deoarece fiecare acces la memorie se face prin intermediul acestor registre, viteza de lucru a acestor registre are o pondere importanta in performanta globala a sistemului.
La alocarea memoriei pentru un proces "managerul de memorie" incarca aceste registre cu numarul cadrelor (libere) alocate procesului. Instructiunile pentru incarcarea sau modificarea registrelor *tabelei de pagini* sunt "privilegiate" astfel ca numai SO sa poata modifica continutul lor.
Utilizarea registrelor pentru *tabela de pagini* este acceptabila atunci cand numarul de intrari in TP (adica dimensiuneaTP) este mica (De ex. este mai mica de 256 intrari).
Pentru administrarea cadrelor libere "managerul de memorie" utilizeaza o lista simplu inlantuita in care se gasesc in fiecare moment numerele cadrelor libere.



a) Situatia inaintea alocarii memoriei pentru un proces.

Pornind de la aceasta situatie initiala pentru noul proces se vor aloca 4 *cadre* libere in ordinea in care se gasesc in *lista de cadre libere*, respectiv:16, 12, 14, 11.
Pagina 0 va fi incarcata in cadrul 16, Pagina 1 in cadrul 12, s.a.m.d.



b) Situatia dupa alocarea memoriei pentru noul proces.

La ora actuala datorita cresterii dimensiunilor aplicatiilor (programelor) s-a ajuns la situatia de a avea un numar foarte mare de pagini (mii). In aceasta situatie nu se mai pot utiliza registre specializate pentru TP datorita cresterii costurilor echipamentului.
O alta solutie este pastrarea TP (Tabela de Pagini) in memoria operativa si utilizarea unui registru care sa "indice" locul TP in memorie.
Acest registru numit si "Page-Table Base Register" (PTBR) este incarcat de SO cu adresa de memorie a TP corespunzator atunci cand "panificatorul" trece in starea de "executie" un proces.
"Comutarea contextului" de la un proces la altul este foarte simpla utilizand registrul PTBR.Ea se rezuma la schimbarea continutului PTBR cu adresa TP a procesului trecu in starea "executie".
Utilizarea acestui mecanism in acest mod conduce insa la o crestere semnificativa a timpului de acces la locatiile memoriei fizice. Acest lucru se datoreaza faptului ca la orice acces la o locatie de memorie lucrurile se petrec astfel:

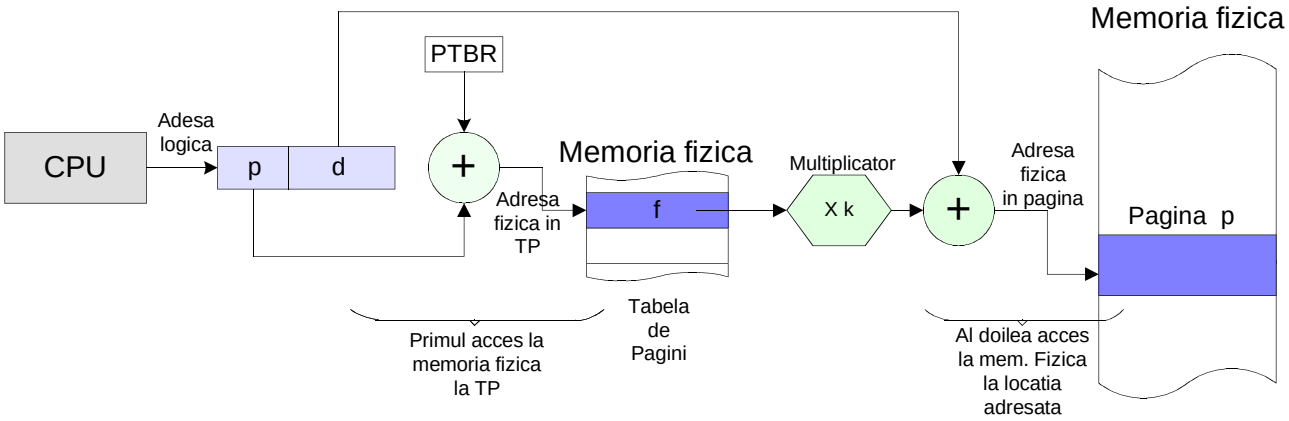
se citeste continutul locatiei de memorie care contine TP de unde se extrage numarul "cadrului" de memorie fizica.

se citeste continutul locatiei propriu-zise de la adresa de memorie fizica obtinuta prin combinarea adresei "cadrului" si "deplasamentului" (offset).

Astfel pentru orice acces la o locatie de memorie este necesar sa se execute doua accese la memorie.

primul acces la TP (care se afla in memoria fizica) de unde se extrage numarul *cadrului* in care este incarcata pagina respectiva.

al doilea acces la memorie la locatia "propriu-zisa" a carei adresa se calculeaza din numarul cadrului x dimensiunea cadrului + deplasament (sau prin concatenare).



Aceasta incetinire a accesului se poate elimina prin alte mecanisme hardware mai speciale.
Unul din mecanismele hardware utilizate pentru adresarea memoriei paginate este utilizarea memoriilor *adresabile prin continut* numite si *Registre Asociative*.
Setul de RA se folosesc pentru memorarea continutului TP.
Registrele asociative (RA) sunt un set de locatii de memorie de mare viteza adresabile prin continut.
Fiecare registru contine doua parti:

un camp "cheie";
un camp "valoare".

Atunci cand unui set de *registre asociative* i se "prezinta" o informatie (numita cheie) ea este comparata simultan cu toate campurile "cheie" din intreg setul de RA. Daca aceasta informatie este identica cu continutul unuia din campurile "cheie" ale setului de RA, atunci acesta "intoarce" continutul campului "valoare" asociat "cheii". Daca informatia "cheie" nu se regaseste in niciunul din registrele setului, atunci se executa un acces la memoria operativa la o adresa fixa "indexata" cu valoarea "cheii". Din memoria operativa se citeste "valoarea" care actualizeaza continutul RA si in acelasi timp ea este "intoarsa" ca rezultat.
Accesul la memorie utilizand RA pentru implementarea TP se deruleaza astfel:
Atunci cand se face o adresare, campul "p" din adresa logica este transmis cu rol de "cheie" la RA. Daca aceasta "cheie" se regaseste in setul de RA atunci se returneaza continutul campului "valoare" care nu este altceva decat numarul *cadrului* de memorie care apoi se concateneaza cu valoarea "d" (deplasarea) din adresa logica, obtinandu-se adresa fizica finala unde se va face accesul.
Daca numarul paginii "p" nu se regaseste in RA atunci automat se va executa un acces la memoria operativa la o adresa fixa (adresa TP) indexata cu "p" de unde se citeste numarul *cadrului* "f" care actualizeaza RA si este "intors" ca rezultat. Orce adresare ulterioara la RA cu aceiasi valoare a cheii va gasi numarul *cadrului* in setul RA nemaifiind necesar un acces la memoria operativa.
Setul de RA contine un numar limitat de locatii si deobicei mult mai mic decat numarul de intrari al TP din memorie. Deci RA vor contine imaginea "partiala" a continutului TP. Acest lucru va face ca obtinerea numarului de *cadru* sa nu se faca indodeauna exclusiv di RA.
Registrele asociative functioneaza asemanator cu memoria " CACHE".
Timpul de acces la setul de RA este foarte mic (cca 2ns) fata de timpul de acces la memoria operativa (8-10ns).
Daca "cheia" de cautare in RA care este numarul paginii "p" nu se afla in RA, se petrece situatia cea mai defavorabila si se executa un acces la TP aflata in memoria operativa. In aceasta situatie timpul de acces pentru obtinerea numarului cadrului unde se afla pagina respectiva este:

t = timp_acces_RA + timp_acces_memorie

Dar in acelasi timp este copiata in RA intrarea respectiva din TP, astfel incat urmatoarele accese la

aceiasi pagina se vor limita la accesul in setul de RA.
Deci rezumand, in situatia cea mai defavorabila accesul la o locatie utilizand RA este:

t = timp_acces_RA + timp_acces_memorie + timp_acces_memorie.

sau

t = timp_acces_RA + **2** * timp_acces_memorie

In cazul cel mai favorabil atunci cand numarul paginii se afla in setul de RA, timpul de acces la o locatie este:

t = timp_acces_RA + timp_acces_memorie

Problema care se pune este de cate ori avem situatii favorabile (acces numai in RA) si de cate ori avem situatia nefavorabila.
Bineinteles ca acest lucru depinde de numarul registrelor din setul RA.
Procentul de cazuri in care numarul paginii este gasit in setul de RA se numeste "Rata de Reusite" (Hit Ratio), iar procentul de cazuri in care numarul paginii nu este gasit in setul de RA se numeste "Rata de Nereusite" (Miss Ratio).
Cu un numar de 16 pana la 512 RA, se obtine o rata de Reusite (RR) de 80 pana la 98.
De exemplu procesorul Motorola 68030 dispune de un numar de 22 RA.
Firma Intel declara ca procesorul 80486 dispune de un set de 32 RA si obtine o rata RR de 98%.
Se poate calcula " probabilistic" timpul efectiv de acces daca se cunoaste RR si timpii de acces la setul de Ra si timpul de acces la memoria operativa.
Presupunand ca timpul de acces la memoria operativa este 10ns si timpul de acces la RA 2ns, si RR = 80% (RN = 20%);

Timpul de acces efectiv = 0,80 x 12 + 0,20 x 22 = 14ns

Daca RR = 90% (RN = 10%)

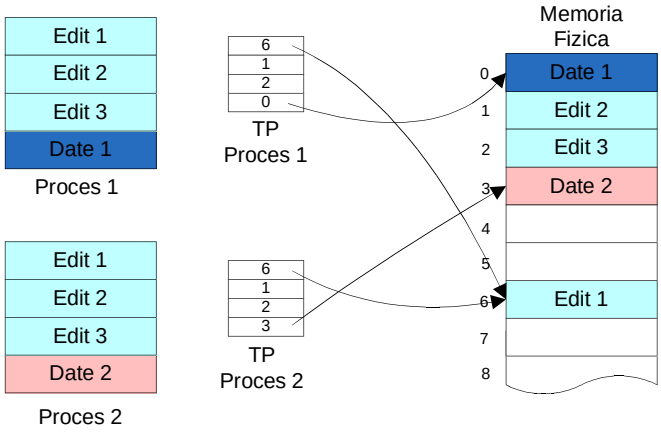
Timpul de acces efectiv = 0,90 x 12 + 0,10 x 22 = 13ns

5.6.4.Pagini partajate.

Un alt avantaj al paginarii este posibilitatea "partajarii" programelor (partea de cod). In cazul sistemelor multi-user apar deseori situatii in care mai multi utilizatori lanseaza (independent unul de altul) o aceiasi aplicatie (de ex. un editor de texte).
Daca programul "editor de texte" este format din 100K parte de cod si 50K parte de date, lansarea de catre mai multi utilizatori a acestui program inseamna ocuparea unui spatiu de memorie de:

M = n * 150K unde:"n" reprezinta numarul de utilizatori care lanseaza simultan acest program.

Daca programul este *reentrant*, partea de cod (100K) poate fi incarcata o singura data in memorie , si numai partea de date specifica fiecarui utilizator (50K) se va multiplica.



Partajarea "codului" in sistemele cu paginare.

Aceasta modalitate se numeste "partajare" a unei aplicatii (in speta a "codului" aplicatiei).
In aceasta situatie spatiul ocupat ar fi de numai:

M = n * 50K + 100K

Conditia de partajare a codului este proprietatea de r*eentranta*.
Un cod *reentrant* numit si "cod pur" trebuie sa indeplineasca urmatoarele conditii:

codul nu trebuie sa se automodifice in timpul executiei.
fiecare executie (instantiere) a codului trebuie sa isi creeze propria copie a registrelor si datelor intr-o zona de date proprie.

5.6.5.Protectia memoriei paginate.

Memoria paginata poate fi protejata in doua moduri:

1) Protejarea partiala prin stabilirea tipului de acces permis la o zona de memorie. Necesitatile de prelucrare pot impune deseori filtrarea accesului la anumite zone de memorie. Adica pot aparea cazuri in care o anumita pagina a procesului careia ii corespunde un anumit *cadru* de memorie fizica trebuie sa fie "numai citita" sau de tip "executie-numai" si "citita si scrisa".
Aceasta reprezinta si cele trei tipuri (cele mai frecvente) de acces la memorie:

read only (citire numai);
execute only (executie numai);
read-write (citire si scriere);

care pot fi impuse prin SO. Stabilirea tipului de acces permis se face la nivelul paginilor procesului.
Acest lucru se face in TP prin adaugarea la fiecare intrare in aceasta tabela a unui numar de 2 biti din combinatia carora se obtin cele trei tipuri de acces.
SO fixeaza pentru fiecare pagina tipul de acces permis in pagina respectiva. In momentul in care se calculeaza adresa fizica a unei locatii se poate verifica daca tipul de acces care urmeaza sa fie facut la locatia respectiva este in concordanta cu configuratia bitilor de protectie a paginii.

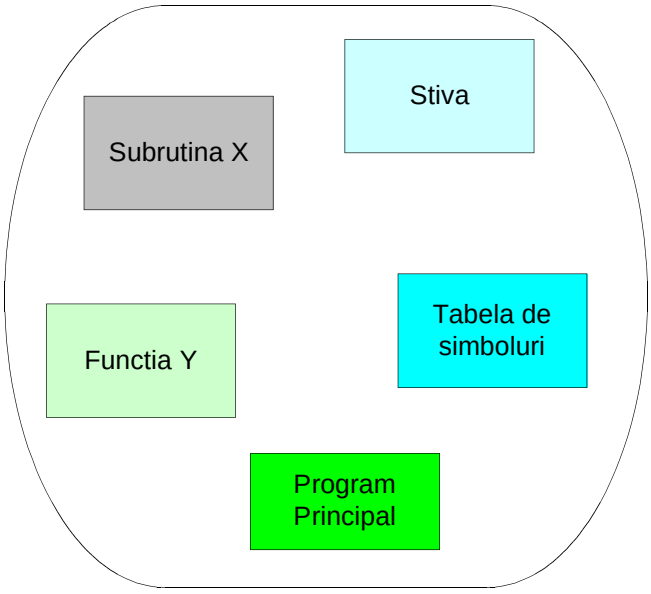
2) Al doilea mod de protectie care se are in vedere este impiedicarea accesului intr-o zona (pagina) care nu apartine procesului. Acest lucru se face prin verificrea ca nrumarul de pagina din adresa logica, sa indice o pagina apartinand procesului. Acest lucru se face verificand de fiecare data ca numarul de pagina din adresa logica este mai mic decat lungimea tabelei de pagini.
Exista mecanisme hardware care incarca intr-un registru (PTLR Page Table Length Register) dimensiunea TP. Orice tentativa de a apela o intrare care depaseste aceasta lungime declansaza o intrerupere hardware (TRAP) care trece controlul in sistemul de operare in modulele de tratare a erorilor de acces la memorie.
La fel se intimpla lucrurile si in cazul tentativei de a face un acces la o locatie de memorie nepermis prin bitii asociati intrari in TP, declansandu-se o intrerupere hardware care va trece controlul SO.
In aceste cazuri SO deobicei suspenda activitatea procesului care a produs accesul "nepermis" si emite un mesaj de eroare de tipul "Violare de memorie" sau " Violarea protectiei memoriei".

5.7. SEGMENTAREA.

Un aspect important care nu poate fi evitat in cazul alocarii paginate a memoriei este "discrepanta" intre modul in care este "vazuta" memoria de catre utilizator si memoria fizica reala. Memoria pe care o "vede" utilizatorul nu este aceeiasi cu memoria fizica reala. Mecanismul de adresare realizeaza "maparea" (proiectia) memoriei logice peste memoria fizica a SC.

5.7.1.Memoria vazuta de utilizatori.

Este in general recunoscut ca utilizatorul (sau programatorul) nu "vede" memoria ca o succesiune liniara de cuvinte (de memorie).
Mai degraba utilizatorul "vede" memoria ca o colectie de segmente de lungime variabila fara sa existe neaparat o ordonare a segmentelor.



Spatiu de adrese logice.

Programatorul atunci cand scrie un program il "vede" ca o colectie de obiecte:

- un program principal;
- un set de subrutine (proceduri si functii) module;
- structuri de date (tablouri, stive, variabile);

Fiecare dintre aceste module sau date sunt referite prin nume. De obicei spunem: "tabela de simboluri a programului" sau functia SQRT, sau "programul principal", fara ca sa ne preocupe locul din memorie ocupat de ele. Nu ne intereseaza de exemplu daca "tabela de simboluri" este inainte sau dupa "functia SQRT". Fiecare din aceste "segmente de program" sunt de lungimi diferite, in functie de informatia aflata in el.
In cardrul unui segment, orice elemente (instructiune, variabila, etc.) este identificat de pozitia sa fata de inceputul segmentului (adresa relativa, offset, deplasare). De exemplu: prima instructiune, a noua instructiune din subrutina SQRT sau a 14 intrare in tabela de simboluri, etc.
Segmentarea este o schema (metoda) de management al memoriei care exprima acest mod de reflectare a memoriei d.p.d.v. al utilizatorului. (Modul in care este perceputa memoria de catre utilizator).

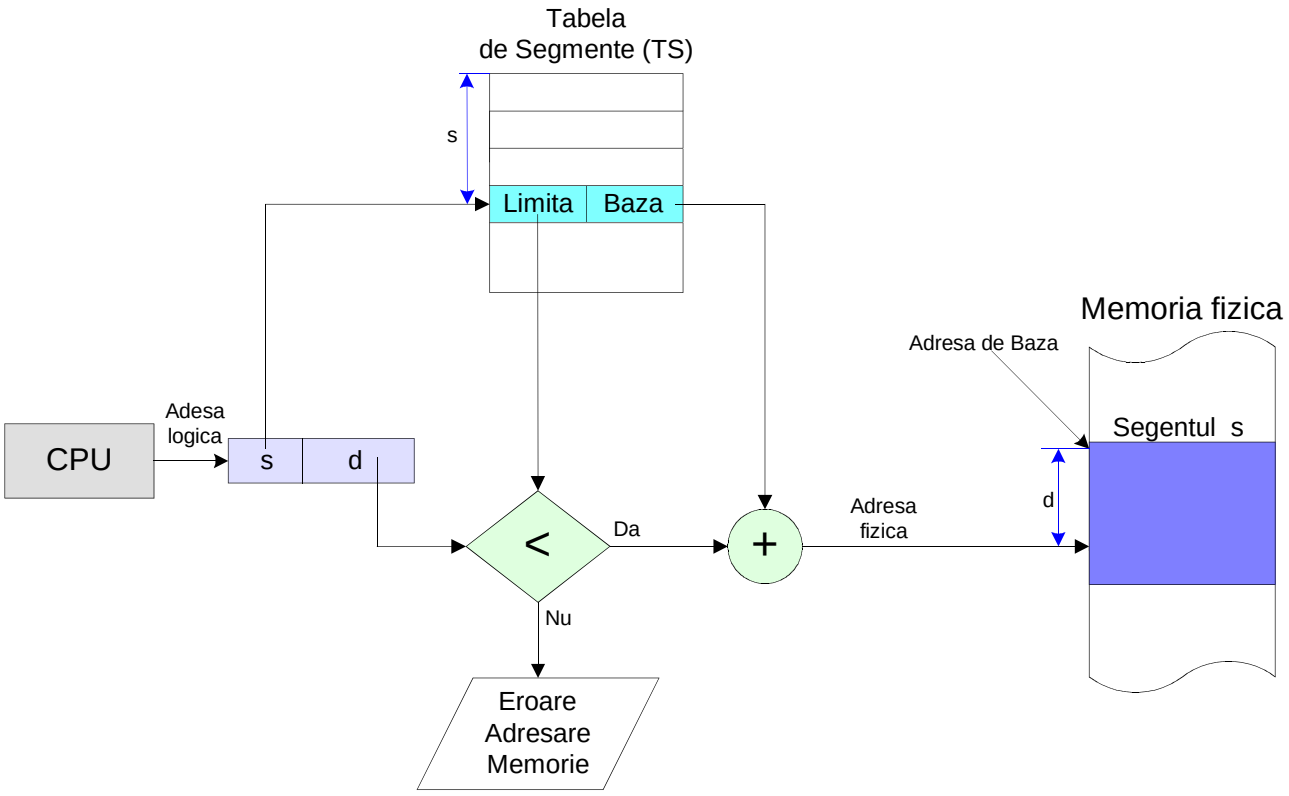
Spatiul de adrese virtuale este o colectie de segmente. Fiecare segment are un nume si o lungime. O adresa in aceasta reprezentare este perechea: nume segment , offset in cadrul segmentului.
In cazul paginarii memoriei utilizatorul specifica o singura adresa care era interpretata de catre mecanismul de adresare ca fiind numar de pagina si offset.
Pentru simplificarea mecanismului segmentele sunt numerotate, referirea la ele fiind facuta prin numarul segmentului. Deci o adresa va fi specificata prin numar segment si offset.
Segmentele rezulta in urma compilarii sau asamblarii. Modul de grupare a informatiilor unui program in segmente este caracteristic fiecarui compilator (asamblor).
De exemplu compilatorul Pascal creaza segmente separate pentru:

variabile globale;
stiva de apeluri al procedurilor(parametrii si adrese de retur al procedurilor);
codul procedurilor sau functiilor (pentru fiecare un segment);
variabilele locale ale fiecarei proceduri sau functii.

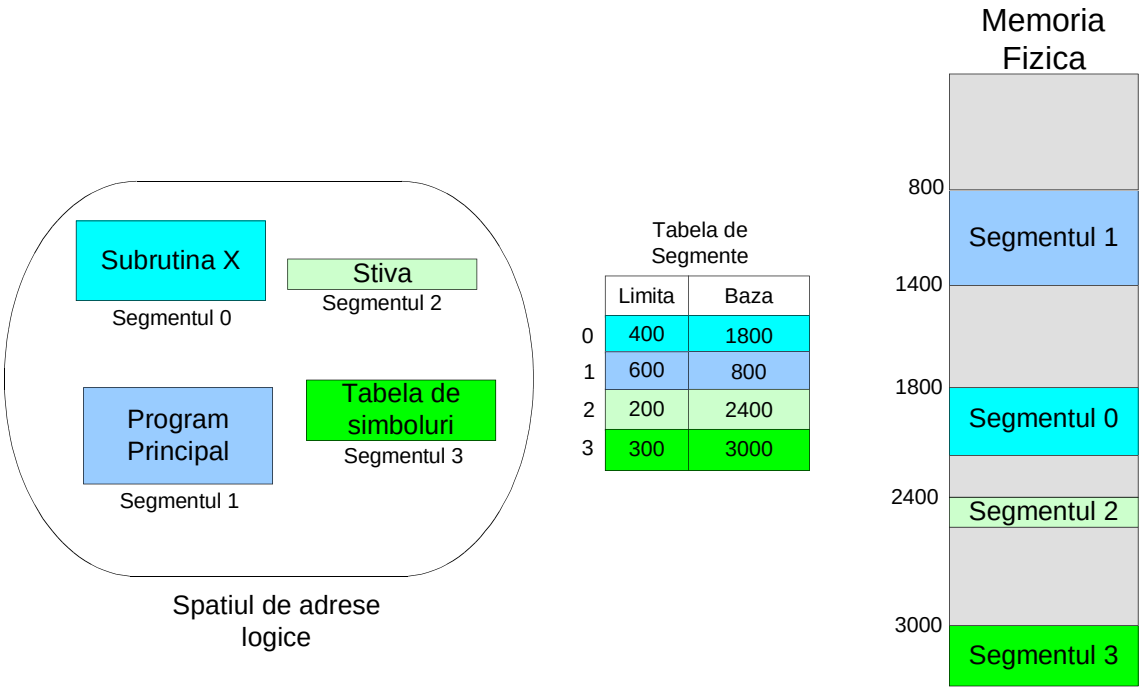
Incarcatorul de programe asigneaza fiecarui segment un numar. Aceasta schema de management al memoriei cu "segmente" trebuie sa fie suportat de hardware. De ex. procesoarele Intel X86 suporta acest mecanism de management al memoriei bazat pe segmente. De obicei programele pe sisteme calculator cu acest tip de microprocesor sunt impartite in segmente de COD, DATA si STACK (STIVA).

5.7.2.Mecanismul hardware pentru adresare cu segmente.

Utilizatorul se poate referi la obiectele sale din programe prin adrese bidimensionale (segment, offset) dar memoria fizica este bineinteles o secventa de cuvinte (un sir de adrese unidimensionale). Mecanismul de adresare trebuie sa implementeze o proiectie (transformare) a adreselor cu doua dimensiuni definite de utilizator in adrese fizice reale (cu o singura dimensiune).
Acest mecanism foloseste o structura de date numita *Tabela de segmente*.



Adresa logica furnizata mecanismului de adresare este formata din numarul de segment "s" si offset (deplasare) "d". Numarul de segment este utilizat ca index in *tabela de segmente* TS. Fiecare intrare in TS contine o pereche de valori "adresa de baza" si "limita" segmentului. Offsetul "d" din adresa logica trebuie sa aiba o valoare cuprinsa intre 0 si "limita" segmentului. Daca "d" este mai mare decat "limita" segmentului se declanseaza o intrerupere de tip "violare de memorie" executandu-se rutina de "Eroare Adresare Memorie". Daca "d" este corect, adica "d"< "limita", atunci valoarea sa se aduna cu "adresa de baza", obtinandu-se adresa fizica de memorie a locatiei adresate.
Vom considera un exemplu un program P care este structurat in 4 segmente: Subrutina X (segmentul 0), Programul Principal (segmentul 1), Stiva programului (segmentul 2) si tabela de Simboluri (segmentul 3).



5.7.3.Implementarea "Tabelelor de segmente"

Segmentarea este destul de asemanatoare cu modelul de memorie cu partitii discutate anterior. Principala diferenta este aceea ca un program consta din mai multe segmente.
Segmentarea este un concept mai complex, intr-o oarecare masura putand fi asemanata si cu paginarea. Ca si in cazul memoriei paginate, Tabela de Segmente ca si Tabela de Pagini poate fi pastrate in registre sau in memorie.
Avantajul pastrarii TS in registre rezida din rapiditatea accesului la "adresa de baza" si a operatiei de comparare cu "limita" segmentului. In plus este posibil ca aceste doua operatii sa fie facute in paralel.
In cazul programelor cu segmente multe nu mai este o solutie realizabila pastrarea TS in registre. In acest caz Tabela de Segmente se va pastra in memoria operativa. In acest caz SO utilizeaza 2 registre in care se pastreaaza adresa de memorie unde se afla Tabela de segmente (STBR) respectiv lungimea Tabelei de Segmente (STLR)

STBR Segment Table Base Register;
STLR Segment Table Length Register;

Intodeauna se verifica ca "s" sa fie mai mica decat continutul lui STLR
Daca conditia "s"< (STLR) atunci se calculeaza "s" + (STBR)

La adresa (STBR) + "s" se vor gasi "limita" si "adresa de baza" a segmentului respectiv.
Se verifica daca "d"< "limita" si se calculeaza "d" + "adresa de baza".
Ca si in cazul paginarii mecanismul de adresare executa 2 accese la memorie pentru a accesa o locatie.
Si in cazul segmentarii se pot folosi "Registre Asociative".
Utilizand un set "Registre Asociative" in care se vor pastra intrarile la segmentele cel mai recent folosite se poate reduce timpul de acces de la valoarea

t =2 * timp_acces_memorie;

la o valoare mult mai mica:

t = timp_acces_memorie + timp_acces_RA

adica

t = timp_acces_memorie + (0,1 0,15) * timp_acces_memorie

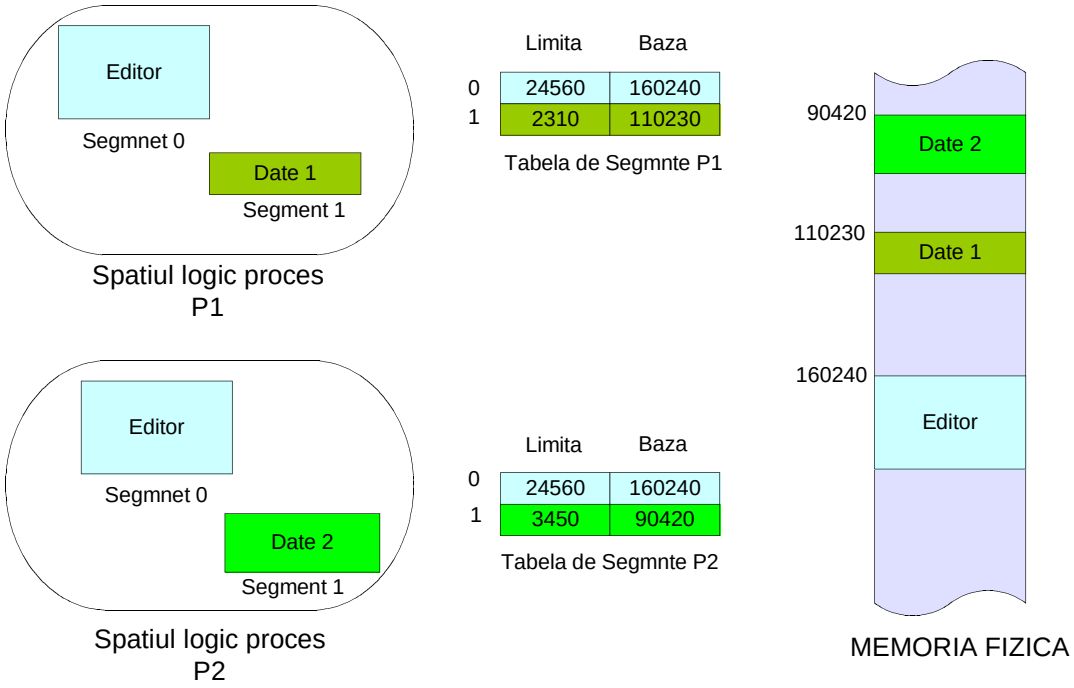
sau

t = (1,1 1,15)*timp_acces_memorie.

Aceasta insemnanad o crestere cu numai 10 - 15% a duratei accesului fata de timpul de acces la memorie.

5.7.4. Protectia si partajarea segmentelor.

Un alt avantaj al segmentarii este si din faptul ca asocierea bitilor de protectie (criteriile de permisie a accesului) se face la fiecare segment in parte. Acest lucru este mai firesc deoarece "segmentul" reprezinta o entitate semantica bine definita a programului ceeace face mai corecta stabilirea criteriilor de acces la fiecare segment in parte.
In programarea moderna instructiunile si datele se grupeaza in segmente separate ceece face mai simpla si naturala stabilirea criteriilor de acces la nivelul fiecarui segment.
De exemplu tehnicile de "partajare" a codului necesita ca segmentele de cod sa fie de tip "citeste-numai" (Read-Only) sau "executa-numai" (Execute-Only). In cazul segmentarii acest lucru este usor de implementat.
Implementarea protectiei se face asociind fiecarui segment o secventa de biti, fiecare configuratie de biti reprezentand un tip de acces permis la segmentul respectiv. Acesti biti se vor inscrie la intrarea corespunzatoare a fiecarui segment
De obicei crieriile de "filtrarea" a acesului sunt aceleasi ca in cazul paginarii:
"citire-scriere","citeste-numai", "executa-numai".
Prin mecanismul de adresare se permite sau nu accesul la o locatie de memorie care apartine unui segment dupa cum tipul accesului (citire, scriere, executie) este sau nu in concordanta cu bitii asociati segmentului unde se incearca sa se faca adresarea.
In plus se verifica ca "d" < "limita" si "s" < (STLR). Acest lucru impiedica accesul la o locatie care nu apartine segmentului sau nu apartine procesului.
Alocarea memoriei cu segmente face posibila partajarea codului. La prima vedere lucrurile par destul de simple si asemanatoare cu partajarea codului in cazul paginarii.
In cazul segmentarii partajarea codului se face prin partajarea segmentelor de cod.
Vom considera un acelasi program "editor de texte" ca si la partajarea in cazul paginarii, care este lansat de mai multi utilizatori, sau un acelasi utilizator care lanseaza de mai multe ori acest editor. Daca acest program este structurat intr-un segment de cod si unul de date, iar segmentul de cod este "partajabil", atunci se va incarca in memorie o singura data segmentul de cod si cate un segmnent de date pentru fiecare "lansare".



Partajarea segmentelor in cazul metodei de alocare cu segmentare

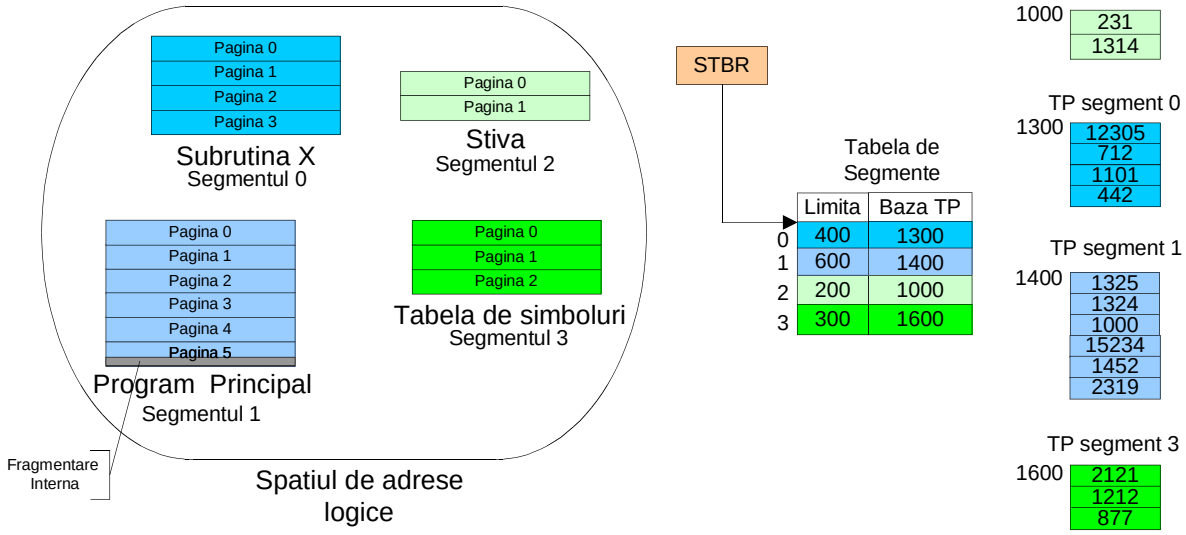
Lucrurile nu stau intodeauna asa.
Sunt si situatii in care procese diferite partajeaza un anumit segment de cod. Este cazul partajarii rutinelor care se afla in bibliotecile sistem. Aceste rutine pentru a putea sa fie partajate sunt decalarate "citeste numai" si "partajabile".Aceste rutine sunt "incorporate" in spatiul logic al fiecarui proces. Ele vor fi vazute ca segmente separate in fiecare program care le apeleaza. Apare insa o problema legata de numarul de segment. In procese(programe) diferite este posibil ca un acelasi segment de cod care este partajat de mai multe procese sa fie vazut cu un numar intr-un proces in timp ce in alt proces acelasi segment sa fie vazut cu alt numar.
Cum se face apelul in interiorul segmentului daca un program vede segmentul "partajat" ca avand numarul 4 si altul numarul 6 (de exemplu) ?
O adresa logica are structura (s,d). In procesul P1 de exemplu adresa logica poate fi (4,240) in timp ce referirea la aceiasi locatie a aceluiasi segment dar in procesul P2 poate fi (6,24). Ar trebui sa avem un acelasi numar de segment.
Aceasta problema se rezolva prin alt tip de adresare si anume prin "adresare relativa" la pozitia curenta a "contorului de program" sau indirect cu un registru care contine numarul segmentului (real).

5.7.5.Fragmentarea.

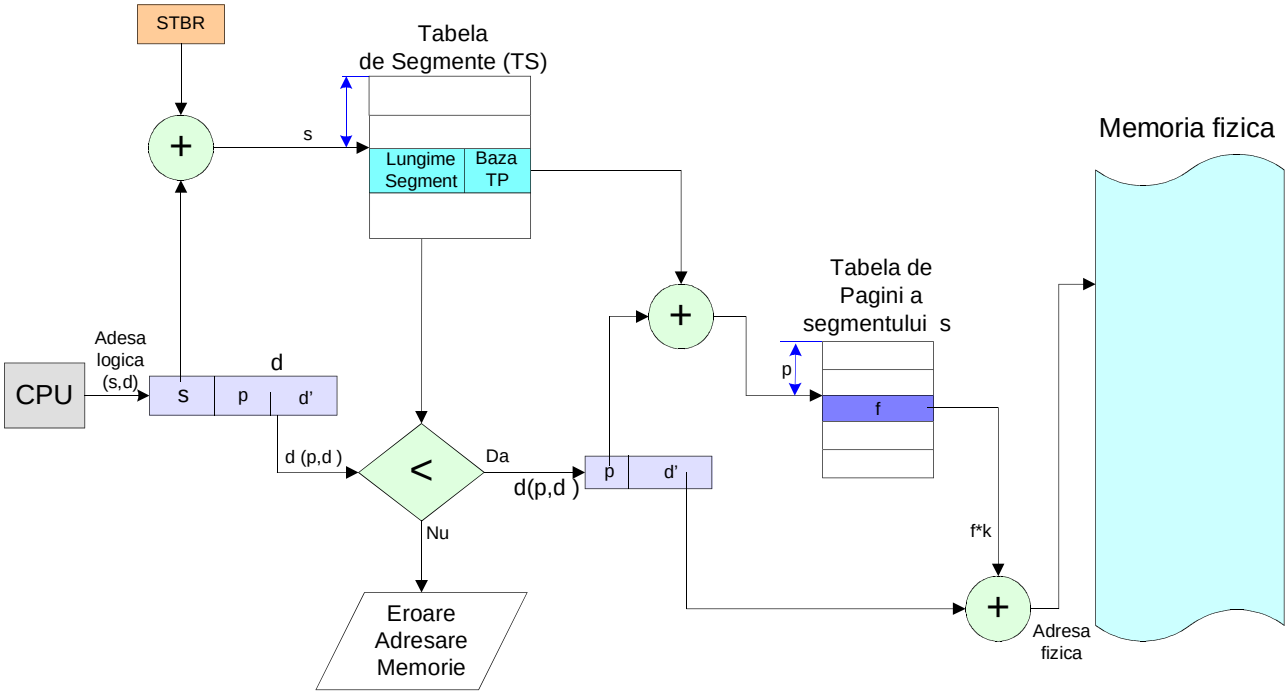
Segmentarea se aseamana atat cu alocarea cu "partitii multiple (cu dimensiune variabila)" cat si cu paginarea. Segmentarea poate produce *fragmentare externa* . In cazul aparitiei unei situatii de *fragmentare externa* fie se asteapta terminarea altor procese care prin eliminarea lor din memorie pot crea "Hol-uri" suficient de mari pentru segmentul ce trebuie incarcat in memorie fie se compacteaza memoria.
Frecventa aparitiei *fragmentarii externe* depinde de dimensiunea segmentelor si a memoriei operative.

5.8.Segmentarea paginata.

Este o metoda de management al memoriei combinat utilizand simmultan segmentarea cu paginare. Fiecare proces este "vazut" ca o multime de segmente, la randul sau fiecare segment fiind "divizat" in pagini. Memoria fizica este divizata in *cadre* ceeace conduce la eliminarea *fragmentarii externe*. Fiecare proces este format din mai multe segmente. La randul lor segmentele sunt impartite in pagini. Fiecare proces are o tabela de de segmente TS iar fiecare segment are o tabela de pagini TP.



Adresa logica este interpretata ca o pereche (s,d) unde "s" este numarul segmentului iar "d" este deplasarea in cadrul segmentului. La randul sau "d" este o structura de forma:(p,d') unde "p"este pagina in cadrul segmentului si "d' "deplasarea in cadrul paginii. In acest tip de management sunt "paginate" segmentele deci poate aparea *fragmentarea interna.*



5.9.Concluzii.

O privire generala asupra metodelor de management a memoriei ne arata ca:

exista o diversitate mare a algoritmilor,de la partitiile single-user, la segmentare-paginata.
Alegerea unui algoritm depinde de capabilitatile oferite de SC hardware.

De exemplu verificarea legalitatii fiecarei adrese generate de CPU nu se poate face eficient prin metode soft deoarece aceste verificari introduc o intarziere in calculul adresei locatiei.
Algoritmii difera prin diferite aspecte:
Suportul hardware.
Un simplu Registru de Baza (sau pereche: registru de baza si registru limita) este suficient pentru implementarea algoritmilor cu o singura partitie (utilizator) sau partitii multiple. Metodele de management ale memoriei cu paginarea sau segmentare implica insa existenta si a unor tabele de "mapare" care sa defineasca adresele segmentelor si /sau paginilor in memoria fizica.
Performanta.
Cu cat algoritmii devin mai complecsi cu atat timpul necesar pentru "maparea" adreselor logice pe adresele fizice devine mai mare. La algoritmii simpli este necesar o simpla comparatie si o adunare. La algoritmii cu paginare si segmentare apar operatii in plus care cresc timpul de acces la locatiile memoriei fizice. Depinde in acest caz foarte mult modul de implementare a tabelelor de pagini/segmente. De la implementarea lor in memoria operativa care este solutia cea mai putin performanta, la registrii asociativi care reprezinta o solutie acceptabila atat in ceeace priveste costurile cat si performanta.
Fragmentarea.
Multiprogramarea implica existenta mai multor procese in memorie. Cu cat numarul programelor exitente simultan in memorie este mai mare cu atat creste probabilitatea aparitie *fragmentarii externe* si implicit cresterea dimensiunii spatiului "irosit". *Fragmentarea externa* apare in cazul metodelor de alocare cu unitati de alocare de dimensiune variabila (alocarea cu partitii multiple si segmntarea).
Fragmentarea interna apare atunci cand se face alocarea cu unitati de alocare de dimensiune fixa (alocarea cu o singura partitie sau alocarea paginata).
Relocarea.
O solutie pentru eliminarea *fragmentarii externe* este "compactarea" periodica a memoriei operative. Compactrea memoriei inseamna "translatarea" programelor in zone compacte in memoria fizica. Acest lucru nu se poate face decat daca "relocarea" adreselor este facuta dinamic in timpul executiei proceselor. Orcum recurgerea la "compactarea" memoriei trebuie facuta cat mai rar ea fiind mare consumatoare de timp.
Evacuarea Reincarcarea (SWAPPING) .
Permite existenta mai multor proces active la un moment dat chiar daca suma memoriei "ocupata" de aceste procese este mai mare decat memoria fizica disponibila. Utilizarea *swapping*-ului complica algoritmii de planificare ceece conduce la o crestere a timpului total de executie.
Partajarea.
Reprezinta un alt mijloc prin care se poate mari gradul de utilizare al spatiului de memorie. Se pot folosi in comun de catre mai multi utilizatori (procese) o aceiasi pagina / segment. Acest lucru va crea posibilitatea ca mai multe programe sa existe simultan in memoria fizica.
Partajarea implica insa tehnici speciale suplimentare de realizarea a "codului"/"datelor" partajate.
Protectia.
SO prin rutinele de management ale memoriei trebuie sa asigure si masurile de protectie a informatiei in memoria operativa. Daca se utilizeaza paginarea sau segmentarea acestea pot fi declarate "citeste-numai", "executa-numai" sau "citeste-scrie". Orce acces la locatiile unei pagini/segment se face cu verificarea conformitatii cu aceste atribute.

5.8. MEMORIA VIRTUALA

5.8.1.Introducere

Toate schemele de alocare a memoriei discutate pana acum aveau drept scop pastrarea simultana in memorie a mai multor procese permitand astfel multiprogramarea.
In toate cazurile discuatate se impunea ca intreg procesul sa se afle in memorie pentru a putea fi executat.
Memoria virtuala este o tehnica care prmite executia proceselor fara ca sa fie necesar ca ele sa fie complet (in intregime) in memorie.
Aceasta tehnica creaza un prim avantaj: lungimea programului poate fi mai mare decat dimensiunea memoriei fizice.
Altfel spus memoria principala devine pentru programator o notiune abstracta inteleasa ca un spatiu foarte mare de stocare separand "memoria logica" vazuta de utilizator de "memoria fizica" reala.
Acest lucru elibereaza programatorul de orice restrictie datorata dimensiunii memoriei.
Memoria virtuala nu este usor de implementat, utilizarea ei putand sa conduca la scaderea performantelor. De obicei *memoria virtuala* se implementeaza cu ajutorul unui mecanism numit *"cerere de paginare"* (demand paging)
Conceptul de "*Memoria virtuala*" a rezultat din observatii asupra programelor reale. S-a contastat ca in foarte multe cazuri nu este necesar intregul program in memorie in timpul executiei sale. Cele mai evidente situatii in care se vede ca nu este necesar intregul program in memorie in timpul executiei sunt:

- Programele contin rutine (subprograme) pentru tratarea erorilor. Aceste rutine sunt apelate si executate numai daca apare o anumita eroare. Deseori acest lucru nu se intampla.
- Tablourile si listele sunt de obicei declarate cu dimensiuni acoperitoare care la cele mai multe din executie nu sunt atinse. De exemplu putem declara un tablou cu dimensiunile (100,100) elemente iar practica se foloseste de cele mai multe ori 10,10 sa zicem.
- La constructia programelor se prevad optiuni sau caracteristici care se utilizeaza foarte rar.

Realiatea ne arata ca si in cazul in care intregul program este necesar, totusi nu este necesar intregul program acelasi timp. Adica la un moment dat este necesar sa fie in memorie o anumita parte a programului iar mai tarziu este necesar sa fie in memorie o alta parte a programului s.a.m.d.
Avantajele utilizarii acestui mecanism care se pastreaza in memorie numai acea parte a procesului care este necesara in acel moment sunt urmatoarele:

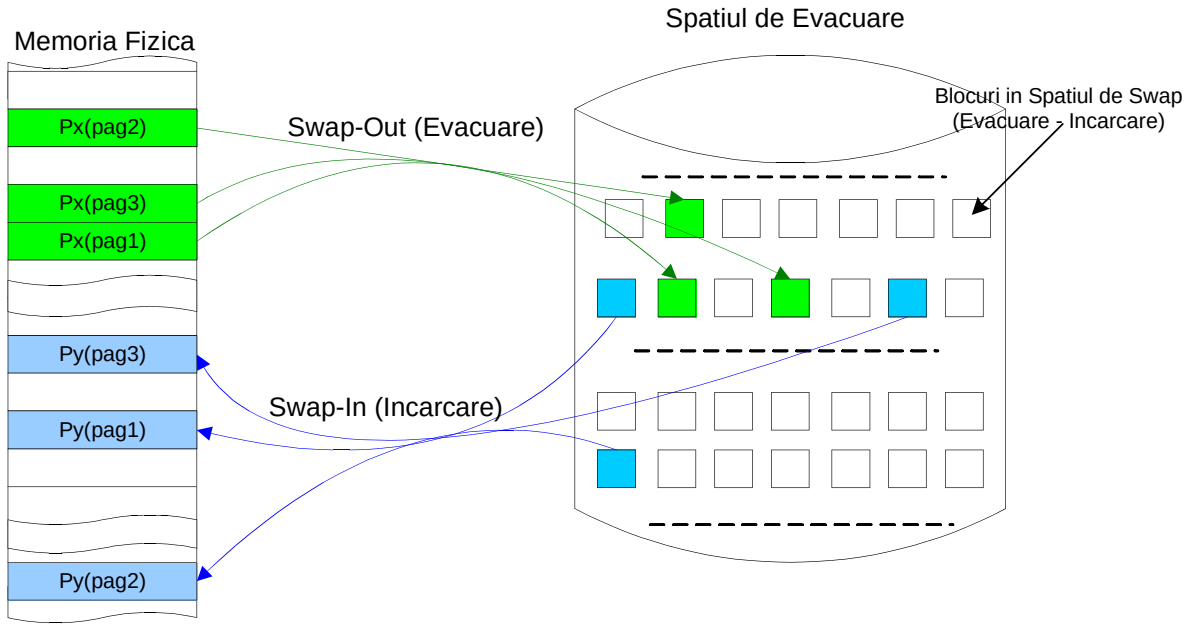
- Programul nu mai este constrans de o anumita dimensiune a *memoriei fizice*. Utilizatorii pot scrie programe pentru un spatiu foarte mare de adrese virtuale simplificand sarcina programarii.
- Deoarece un program ocupa in timpul executiei mai putina memorie, mai multe procese pot fi executate in acelasi timp. Acest lucru conduce la cresterea gradului de utilizare a CPU fara sa creasca timpul de raspuns.
- Operatiile de I/O pentru incarcarea proceselor in memorie sau pentru "swapping" manipuleaza un volum de date mai mic durata globala de executie a unui program utilizator fiind mai mica.

De acest mod de lucru in care se pot executa procese care nu sunt in intregime in memorie vor fi avantajati si utilizatorii si SO.

Memoria virtuala inseamna separarea memoriei logice "vazute" de utilizator de memoria fizica (reala). Acest lucru permite ca programatorii sa foloseasca un spatiu larg de *memorie virtuala* in timp ce *memoria fizica* este foarte mica. In acest fel programarea devine mai simpla nefiind nevoie sa ne incadram intr-o dimensiune de memorie prestabilita. *Memoria virtuala* a condus la disparitia tehnicienilor de "reacoperire" (overlays).
Memoria virtuala se implementeaza de obicei printr-un mecanism numit "cerere de paginare", ea putand fi implementata in toate sistemele care utilizeaza metoda de alocare cu paginare sau segmentare paginata.
In cazul sistemelor care utilizeaza alocarea cu segmente se poate implementa un mecanism care implementeaza memeoria virtuala folosind mecanismul "cerere de segment". Memoria virtuala in sistemele de management cu segmente (fara paginare) apar complicatii la inlocuirea segmentelor deoarece acestea au dimensiune variabila.

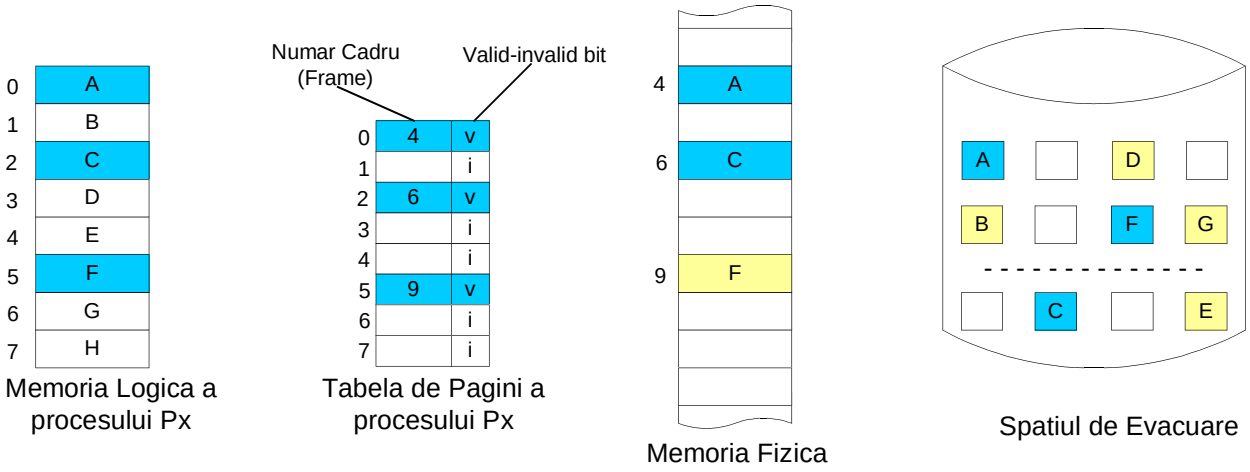
5.8.2Cererea de paginare.

Cererea de paginare (sau pagina) este un mecanism asemanator cu mecanismul utilizat de sistemele cu anagement cu paginare si evacuare-reincarcare (swapping).



In sistemele cu management cu paginare, programele (procesele) se afla initial pe disc. Atunci cand vrem sa executam un proces, acesta trebuie "incarcat" in memorie (swap-in), si el se va incarca in intregime (toate paginile sale).
De exemplu procesul Px care este format din 3 pagini, atunci cand este "evacuat" (swap-out) se vor transfera pe disc toate cele 3 pagini ale sale. Daca un alt proces Py trebuie incarcat in memorie de pe disc (swap-in) atunci toate paginile sale vor fi transferate in emorie.
Deci in sistemele cu paginare la evacuare sau reincarcare (swap-out, swap-in) se manipuleaza intregul procesul.
In cazul in care se implementeaza un mecanism de *cerere de paginare* daca un proces trebuie incarcat in memorie atunci se va incarca in memorie numai acea pagina a procesului care va fi utilizata. Altfel spus mecanismul de *cerere de paginare* incarca in memorie numai acele pagini ale procesului care sunt necesare la un moment dat. In acest fel se evita incarcarea in memorie a paginilor care nu vor fi folosite niciodata facandu-se o economie de memorie si de timp prin reducerea numarului de operatii de evacuare-reincarcare.

Cererea de paginare (demand paging) necesita insa suport hardware suplimentar.
In *tabela de pagini* se adauga un bit numit *valid-invalid bit* fiecarei intrari in tabela. Deci fiecarei pagini a procesului i se va asocia un bit care va "arata" daca pagina respectiva se afla sau nu in memorie.



Pagina aflata in memoria fizica: Valid="v"; Pagina care nu se afla in memorie: Invalid="i";
Atunci cand o pagina este incarcata in memorie, bitul "*valid-invald bit*" este setat la valoarea "v".
Daca executia procesului se face in paginile aflate in memorie (marcate "v") lucrurile se petrec ca si cum intreg procesul ar fi in memorie si acest caz se numeste "*cerere de paginare pura*".
Niciodata nu se incarca in memorie o pagina pana cand nu se face referire la ea.
Atunci cand se face referire la o pagina marcata "i" (care deci nu se afla in memorie) se declanseaza o intrerupere de tip "pagina eronata" ("page fault") care in continuare lanseaza in lucru mecanismul de incarcare a paginii "lipsa" in memorie si setarea bitului paginii respective la valoarea "v". Dupa aceasta se reia mecanismul de adresare la locatia care anterior
Teoretic este posibila si situatia extrema in care executia fiecarei instructiuni a unui program sa declanseze intreruperea "pagina-eronata". In acest caz la executia fiecarei instructiuni se va incarca o pagina in memorie, ceea ce va determina o performanta foarte scazuta a sistemului.
In realitate programele au o caracteristica care se numeste "referire locala". Aceasta inseamna ca cele mai multe instructiuni fac "referiri" la locatii care se afla in imediata vecinatate (local). Adica probabilitatea de referire la o locatie care se afla intr-o pagina absenta din memorie (marcata "i") este foarte mica.
Acest lucru face ca in realitate performanta sistemelor cu "cerere de paginare" sa fie rezonabila.

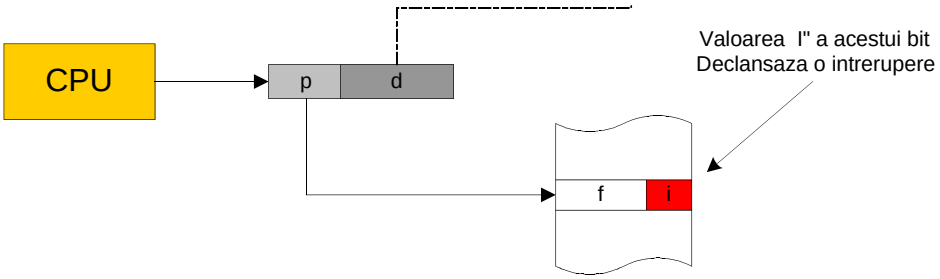
Suportul hardware pentru mecanismul "cerere de paginare" cuprinde:

Tabela de pagini cu proprietatea ca fiecarei intrari ii este asociat bitul "valid-inavalid" sau valori speciale pentru bitii de protectie care sa indice prezenta sau absenta paginii in memorie.

Memorie secundara in care se pastreaza paginile care nu se afla in memorie ale proceselor active. De obicei memoria secundara este o unitate de disc rapida, cunoscut si sub numele de "perifericul de evacuare reincarcare" (swap device) respectiv "spatiul de evacuare-reincarcare" ("swap space").

In afara suportului hardware sunt necesare mecanisme software suplimentare pentru implementarea mecanismului de *cerere de paginare*. Trebuie remarcat faptul ca daca o pagina este marcata "i" acest lucru nu are nici un efect daca nu se face acces la pagina respectiva. Atata timp cat procesul in executie acceseaza pagini rezidente in memorie (pagini cu bitul setat "v") executia se desfasoara "normal" ca si in cazul sistemelor fara *cerere de paginare*.

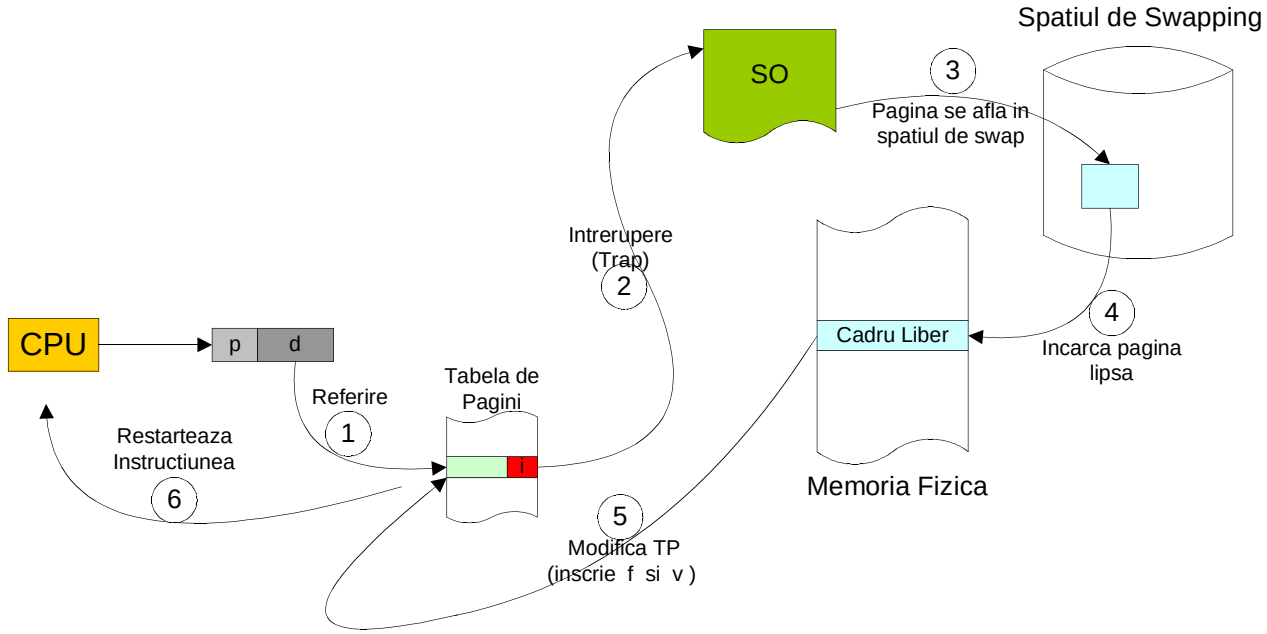
Daca insa procesul incearca sa faca un acces (citire sau scriere) la o adresa apartinand unei pagini care nu se afla in memorie se declanseaza o serie de activitati hardware si software care au ca finalitate incarcarea paginii respective in memorie.
Cum se petrec lucrurile in cazul unui acces la o adresa apartinand unui pagini "nerezidente" in memorie ?



Mecanismul de adresare va "citi" intrarea "p" corespunzatoare, din Tabela de Pagini care in acest caz are bitul "valid-invalid" marcat "i". Acest lucru va declansa o "intrerupere" de tip "adresa ilegala" numita in anumite sisteme "adresare incorecta" sau "pagina eronata".
In continuare se deruleaza un mecanism clasic de tratare a intreruperilor:

se "salveaza" contextul programului intrerupt (IP, PSW, registre, etc) in stiva;
se incarca *noua stare program* corespunzatoare rutinei din SO pentru tratarea intreruperilor de "adresare ilegala". Deoarece aceasta intrerrupere poate fi declansata si de o eroare de adresare clasica, rutina verifica cauza care a declansat aceasta intrerupere. Exista doua posibilitati:
a) Adresa este invalida dintr-o eroare de programare si in acest caz procesul respectiv este intrerupt afisandu-se un mesaj de eroare indicand utilizatorului"eroare de adresare";
b) Este o adresare intr-o pagina care nu se afla in memorie, caz in care se desfasoara schema de *cerere de paginare* propriu-zisa.
Se cauta un *cadru* (Frame) liber de memorie fizica;
Se lanseaza cererea de transfer pentru pagina dorita (aflata pe disc) in memoria fizica in *cadrul* liber gasit anterior;
Se asteapta terminarea transferului de pe disc in memoria fizica si apoi se modifica bitul valid-invalid al paginii incarcate in memorie din valoarea "i" in "v" marcand faptul ca pagina se afla in memorie;
SO reface contextul procesului "salvat" la aparitia intreruperii si se restarteaza instructiunea intrerupta de "adresare ilegala". De data aceasta pagina in care se face adresarea fiind in memorie accesul la locatia de memorie se face normal.

De remarcat faptul ca procesul este reluat exact din locul in care s-a produs "eroarea de adresare" si exact in conditiile initiale deoarece la intrerupere se salveaza registrele, IP, PSW etc iar la restartarea continutul lor este refacut de catre SO (restaurarea contextului initial)
Deci in concluzie algoritmul de alocare a memoriei bazat pe " Memoria virtuala " foloseste acest mecanism numit " cerere de paginare ".
" Cererea de paginare " asigura mecanismul prin care atunci cand un proces incearca sa acceseze locatii care nu se afla in memorie, declanseaza o intrerupere (TRAP) care deturneaza executia spre sistemul de operare care aduce pagina dorita in memorie restartand apoi procesul.



Este posibil (in cazuri extreme) ca sa lansam un proces care nu are nici o pagina in memorie. Tentativa de a lansa procesul in executie declanseaza imediat o intrerupere "Page Fault" care determina incarcarea primei pagini in memorie. Se continua executia si la urmatoarea instructiune poate sa se intample acelasi lucru (page fault). Astfel este posibil sa se incarce in memorie, pe rand, toate paginile procesului dupa care in continuare sa se petreaca lucrurile fara nici o intrerupere de tip "page fault" , aceasta situatie numindu-se "cerere de paginare pura". Niciodata nu se incarca in memorie o pagina pana ce ea nu este necesara.
In cazul *cererilor de paginare* exista cerinte specifice de natura arhitecturala pentru a ca dupa tratarea intreruperii "page fault" sa fie posibila relansarea executiei instructiunii.
Aceste restrictii se refera la sistemele calculator care accepta moduri speciale de adresare, cum ar fi modurile autodecrement sau autoincrement.

5.8.3Performanta "cererii de paginare".

Cererea de paginare poate afecta performantele in ansamblu sistemului calculator. Vom incerca sa evaluam "timpul efectiv de acces" in cazul "cererii de paginare". Timpul de acces la memoria principala (operativa) este pentru sistemele actuale in jurul valorii de 10ns.

T_{acces_mem}=10 ns =0,01 ì s

Atat timp cat nu avem "page fault" deci accesul se face in pagini prezente in memorie, timpul efectiv de acces este acelasi cu timpul de acces la memorie.
In cazul in care insa, apare un acces la o pagina care ne se afla in memorie ("page fault"), trebuie mai intai incarcata pagina respectiva in memorie si apoi se executa accesul la informatia dorita.
Notam cu "p" probabilitatea de a se intampla "page fault" cu 0Wp W1.

$T_{efectiv\_acces} = (1-p)T_{acces\_mem} + p * T_{tratare\ "page\ fault"}$

Trebuie deci sa evaluam "timpul de tratare a page fault-lui".
Daca apare o eroare de tip "page fault" se deruleaza urmatoarea secventa de activitati:

1. "Trap" la sistemul de operare (rutina de tratare intrerupere "page fault")
2. Salvarea registrelor utilizator si starea procesului (contextul procesului)
3. Determinarea (verificarea) faptului ca referinta la pagina a fost legala si determinarea locatiei unde se afla pagina pe disc.
4. Lansarea operatiei de transfer de pe disc intr-un *cadru* liber de memorie

- inscrierea in " coada de cereri " a perifericului a cererii efective de transfer;
- executia operatiei de transfer propriu zisa care inseamna urmatoarele activitati hard:
  a) pozitionarea capatelor de citire (cca 8 ms)
  b) cautarea informatiei (cca 10 ms)
  c) transferul propriu zis al paginii de pe disc in memorie (cca 1 ms).

Aceasta operatie se desfasoara pornind de la presupunerea ca exista un cadru liber in memorie.

5. Pe durata desfasurarii transferului CPU poate fi alocat unui alt proces (planificatorul de procese optional);
6. Se declansaza "intreruperea" data de unitatea de disc la sfarsitul transferului;
7. Salvarea contextului procesului activ in momentul aparitiei intreruperii (registrele si starea procesului)
8. Tratarea intreruperii disc.
9. Actualizarea *Tabelei de pagini* a procesului (numarul *cadrului* si setarea bitului *valid-invalid*).
   Actualizarea *Tabelei de procese* prin trecerea indicatorului de stare procesuluila valoarea "Gata".
   Planificatorul trece in starea "Executie" urmatorul proces din lista de procese active.
10. Asteapta ca CPU sa fie alocat din nou procesului.
    Atunci cand *planificatorul* trece din nou procesul in starea "Executie" :
11. Restaureaza contextul procesului (registre si starea procesului) si relansarea instructiunii care a declansat "page fault-trap".

Nu intotdeauna se deruleaza toti acesti pasi. De ex. pasul 5 in care CPU este alocat unui alt proces pe durata transferului, conduce la cresterea gradului de multiprogramare dar consuma un timp suplimentar pentru relansarea procesului. Daca se renunta la aceasta strategie se poate elimina acest timp de "regie" suplimentar..

In orice caz in toata aceasta secventa exista trei componente majore care apar indiferent de strategie:

tratarea intreruperii "page fault";
citirea paginii absente;
restartarea procesului;

din care prima si cea de a treia inseamna executia catorva sute de instructiuni la nivelul unitatii centrale care ca durata pot fi aproximate la circa 1 milisecunda. Deci timpul de rezolvare a unei cereri de "demand paging" poate fi aproximat astfel: (timp de tratare " page fault ")

deplasarea capetelor de citire disc: 8 milisecunde.
timpul de cautare a informatiei: 10 milisecunde.
timpul de transfer propriu zis 1 milisecunda.
timpul de executie a tratare a intreruperii si restartarea procesului:1 milisec.

Total : aprox. 20 milisecunde.

Inlocuind in formula

$$T_{acces\_efectiv}=(1-p)*T_{acces\_mem}+ p * T_{tratare\ "page\ fault"}$$

obtinem

$$T_{acces\_efectiv} =(1-p)*10nanosecunde+ p * 20milisecunde$$

si transformand totol in ns:

$$T_{acces\_efectiv}=10 \quad 10 * p + 20.000.000 * p= 10 + 19.999.990 * p \text{ (nanosecunde)}$$

deci

$$T_{acces\ efectiv} = 10 + 19.999.990 * p \text{ (ns)} = 10 + 20.000.000 * p \text{ (ns)}.$$

unde vedem ca timpul de acces efectiv ($T_{acces\_efectiv}$) este direct proportional cu probabilitatea de aparitii a "cererilor de paginare".

Daca de exemplu unul din 1000 de accese la memorie declanseaza "cererea de paginare" timpul

efectiv de acces este de cca 20 microsecunde.

p = 0,001

$T_{acces\_efectiv}$ = 10 + 19.999.990 * 0,001(nanosecunde)=

= 10 + 19.999,9 = 20.000,9ns ≅ 20ì s = 2000 * (0,01)ì s = 2000 * (10ns).

Deci timpul efectiv de acces de circa 20ì s inseamna o incetinire de cca 2000 ori fata de accese la memorie fara "cerere de paginare" la care timpul de acces este de cca 10ns.
Daca dorim sa calculam probabilitatea "p" astfel incat sa obtinem o "degradare" a performantelor cu 10% fata de accesul la memorie, vom calcula astfel:
Un timp de acces mai lent cu 10% inseamna 10ns+10%=11ns.

11 > 10 + 20.000.000 * p sau 1 > 20.000.000 * p sau p < 1 / 20.000.000

Deci **$p < 0,5 * 10^{-7}$**

Asta inseamna sa avem 1 acces la memorie care declanseaza o "cerere de paginare" din 20 milioane de accese fara "cerere de paginare".
Deci este important sa mentinem foarte scazuta rata cazurilor de acces cu "cereri de paginare" (demand paging) altfel durata prelucrarii se mareste nepermis de mult.
Un alt aspect referitor la performanta "cererii de paginare" o reprezinta modul in care este organizat spatiul disc unde se afla procesul (paginile sale).
Spatiul "swap" (evacuare reincarcare) este in general mai rapid decat utilizarea sistemului de fisiere SO , deoarece spatiul de "swap" este alocat in blocuri de dimensiuni mai mari si nu sunt utilizate metode de alocare indirecta (index de fisiere, etc).

5.8.4. Inlocuirea paginilor.

In cele discutate anterior am vazut ca rata adresarilor in care apare "page fault" nu este o problema deorece fiecare pagina este adresata cu "page fault" cel mult odata atunci cand se face o referinta la o locatie in pagina respectiva pentru prima oara. Aceasta reprezentare poate insa sa nu fie foarte realista din cauza ca intotdeauna am presupus ca exista un *cadru* liber de memorie atunci cand se face o cerere de paginare.
Consideram un exemplu:
Un proces care este format din 10 pagini dar in momentul actual foloseste numai 5 pagini.
Remarcam ca utilizarea *Cererii de paginare* creaza o economie de timp datorita faptului ca celelalte 5 pagini nu vor fi incarcate in memorie.
Presupunem de exemplu ca memoria operativa dispune de 40 *cadre* (*frame*-uri) de memorie fizica.
Teoretic, putem creste gradul de multiprogramare putand incarca si executa 8 procese (daca presupunem ca au tot dimensiunea de 10 pagini si numai 5 folosite) in loc de 4 procese fara *cerere de paginare*.
Se pune problema cat trebuie sa fie numarul maxim de procese care pot fi active simultan (gradul de multprogramare).
De exemplu daca vom stabili ca sistemul lucreaza la gradul 6 de multiprogramare atunci 6 procese (de 10 pagini dar numai 5 folosite) vor ocupa in medie 6 * 5=30 cadre memorie.
In aceasta situatie raman 10 cadre care le consideram de rezerva.
De ce sa pastram aceste cadre de rezerva ? Pentru ca atunci ca un exemplul nostru procesele aveau 10 cadre si numai 5 folosite "in cele mai multe din cazuri" dar nu intodeauna.
Este posibil ca procesele pentru seturi particulare de date sa foloseasca toate cele 10 pagini rezultand o situatie in care vor fi necesare 60 de cadre in timp ce numai 40 sunt disponibile.
In situatiile acestea (statistic foarte rare) numi putea avea 6 procese active simultan.

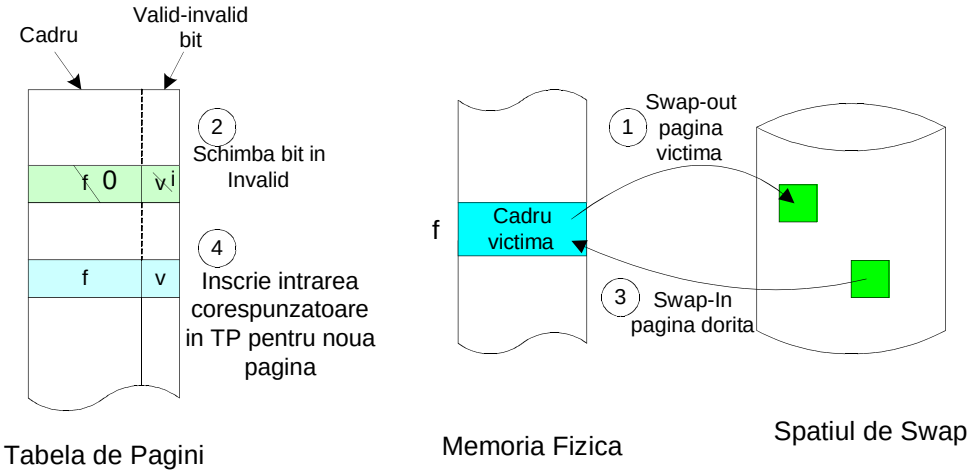
Aceasta situatie nefavorabila poate sa apara cu o probabilitate cu atat mai mare cu cat se creste gradul de multiprogramare sau altfel spus cu cat dimensiunea de memorie utilizata in medie de procesele active se apropie de dimensiunea memoriei fizice.
Din aceste motive in exemplul nostru gradul de multiprogramare se alege 6 cu toate ca el putea fi ales 7 sau 8.
Situatia in care SO in urma unei intreruperi "fault page", trebuie sa incarce de pe disc in memorie o pagina a unui proces si nu gaseste in memoria fizica nici un *cadru* (frame) liber se numeste "*Supraalocarea*" (overallocation).
In aceasta situatie SO poate actiona in mai multe feluri:

termina procesul care a solicitat *cererea de paginare*;

evacueaza procesul (swap-out) eliberand toate cadrele ocupate de el in memorie, dar reducandu-se gradul de multiprogramare;

aleage dupa un anumit criteriu (algoritm) o pagina din memorie care sa o evacueze pe disc pentru a face loc paginii care trebuie incarcate (Aceast mecanism numindu-se *inlocuire pagina*).

Bineinteles ca practic nu pot fi luate in considerare decat cea de a doua si a treia varianta.



Schema de *inlocuire pagina*

Pentru inceput o sa detaliem "mecanismul numit *inlocuirea paginii*. In aceasta varianta daca nu se gaseste un *cadru* liber SO va cauta in memorie un cadru care nu este utilizat (pagina din el nu este activa) si il va elibera. Eliberarea cadrului se poate face copiind continutul sau pe disc si schimband "indicatorii" din toate tabelele aratand ca pagina nu mai este in memorie. *Cadrul* eliberat poate fi utilizat acum pentru "primirea" paginii procesului care a declansat *cererea de pagina.*
Rutina de serviciu SO discutata anterior pentru tratarea intreruperii "page fault" se va modifica prin includerea schemei de *inlocuirea paginii.* Aceasta implica executarea urmatoarelor actiuni:

1. Gasirea locatiei paginii solicitate, pe disc in imaginea executabila a procesului;
2. Gasirea unui *cadru* liber in memoria fizica:

   a. Daca exista un cadru liber ” acesta se va utiliza (trece la punctul 3)

   b. Daca nu exista un cadru liber, aplica algoritmul de *inlocuire a paginii* pentru selectarea cadrului ce se va evacua. Acesta se numeste *cadrul victima*;

   c. Scrie (copiaza) *pagina victima* pe disc si modifica corespunzator tabelele care tin evidenta paginilor si a cadrelor;

3. Citeste (copiaza) pagina dorita in cadrul liber si modifica corespunzator tabelele de evidenta a paginilor si a cadrelor;
4. Restarteaza procesul intrerupt de data aceasta existand in memorie pagina in care se face adresarea;

Trebuie remarcat ca in cazul executarii mecanismului de *inlocuire de pagina* , va avea loc transferul a 2 pagini (una se evacueaza:swap-out, si alta incarca: swap-in), ceea ce dubleaza timpul de executie al serviciului "page fault"..

Exista posibilitatea reducerii acestui timp prin utilizarea asa numitului *bit de modificare* (sau *bit de murdarire*). Fiecare *pagina* aflata in memorie are asociat un bit care initial este "0". Daca in pagina respectiva are loc o modificare atunci *bitul de modificare* va fi setat (va capata valoarea "1"). Atunci cand o pagina trebuie evacuata (swap-out) se verifica *bitul de modificare*. Daca acesta are valoarea "0" atunci *pagina* nu mai este necesar sa fie evacuata efectiv deoarece imaginea sa este identica cu cea de pe disc.

In acest fel se reduce timpul de *inlocuire de pagina.* Daca *pagina victima* este de tip *read-only* (citeste-numai) care nu poate fi modificate in timpul executiei deasemeni nu va mai fi necesara "evacuarea" paginii pe disc deoarece ea este identica cu "imaginea" sa de pe disc.

Schema de *inlocuire de pagina* este foarte importanta in *cererea de paginare* deoarece creaza posibilitatea executiei programelor a caror dimensiune a memoriei virtuale este mai mare decat memoria fizica.

De exemplu putem executa un proces de 20 pagini intr-un spatiu de memorie fizica de 10 cadre.

Asa cum am vazut implementarea *cererii de paginare* implica rezolvarea a 2 probleme majore:

alocarea *cadrelor*;
inlocuirea *paginilor*.

Atunci cand in memorie avem mai multe procese trebuie luata urmatoarea decizie:

Cate cadre se aloca fiecarui proces?

Acest lucru este facut de "algoritmul de alocare cadre ".

In continuare atunci cand este necesara *inlocuirea paginilor* trebuie selectate *cadrele* care trebuie inlocuite, acest lucru fiind realizat de " algoritmul de inlocuire pagina ".

Impotanta acestor algoritmi este majora avand in vedere faptul ca operatiile de intrare-iesire (transferurile cu discul magnetic) sunt mari consumatoare de timp.

5.8. MEMORIA VIRTUALA

5.8.1.Introducere

Toate schemele de alocare a memoriei discutate pana acum aveau drept scop pastrarea simultana in memorie a mai multor procese permitand astfel multiprogramarea.
In toate cazurile discutate se impunea ca intreg procesul sa se afle in memorie pentru a putea fi executat. Din aceasta cauza s-au introdus concepte noi care sa permita aplicarea unor mecanisme care sa elimine aceasta "restricti". Acesta este motivului aparatiei conceptului de memorie virtuala.
Memoria virtuala este o tehnica care prmite executia proceselor fara ca sa fie necesar ca ele sa fie complet (in intregime) in memorie.
Aceasta tehnica creaza un prim avantaj: lungimea programului poate fi mai mare decat dimensiunea memoriei fizice.
Altfel spus memoria principala devine pentru programator o notiune abstracta inteleasa ca un spatiu foarte mare de stocare separand "memoria logica" vazuta de utilizator de "memoria fizica" reala.
Acest lucru elibereaza programatorul de orice restrictie datorata dimensiunii memoriei.
Memoria virtuala nu este usor de implementat, utilizarea ei putand sa conduca la scaderea performantelor. De obicei *memoria virtuala* se implementeaza cu ajutorul unui mecanism numit *"cerere de paginare"* (demand paging)
Conceptul de "*Memoria virtuala*" a rezultat din observatii asupra programelor reale. S-a contastat ca in foarte multe cazuri nu este necesar intregul program in memorie in timpul executiei sale. Cele mai evidente situatii in care se vede ca nu este necesar intregul program in memorie in timpul executiei sunt:

- Programele contin rutine (subprograme) pentru tratarea erorilor. Aceste rutine sunt apelate si executate numai daca apare o anumita eroare. Deseori acest lucru nu se intampla.
- Tablourile si listele sunt de obicei declarate cu dimensiuni acoperitoare care la cele mai multe din executie nu sunt atinse. De exemplu putem declara un tablou cu dimensiunile (100,100) elemente iar practica se foloseste de cele mai multe ori 10,10 sa zicem.
- La constructia programelor se prevad optiuni sau caracteristici care se utilizeaza foarte rar.

Realiatea ne arata ca si in cazul in care intregul program este necesar, totusi nu este necesar intregul program acelasi timp. Adica la un moment dat este necesar sa fie in memorie o anumita parte a programului iar mai tarziu este necesar sa fie in memorie o alta parte a programului s.a.m.d.
Avantajele utilizarii acestui mecanism care se pastreaza in memorie numai acea parte a procesului care este necesara in acel moment sunt urmatoarele:

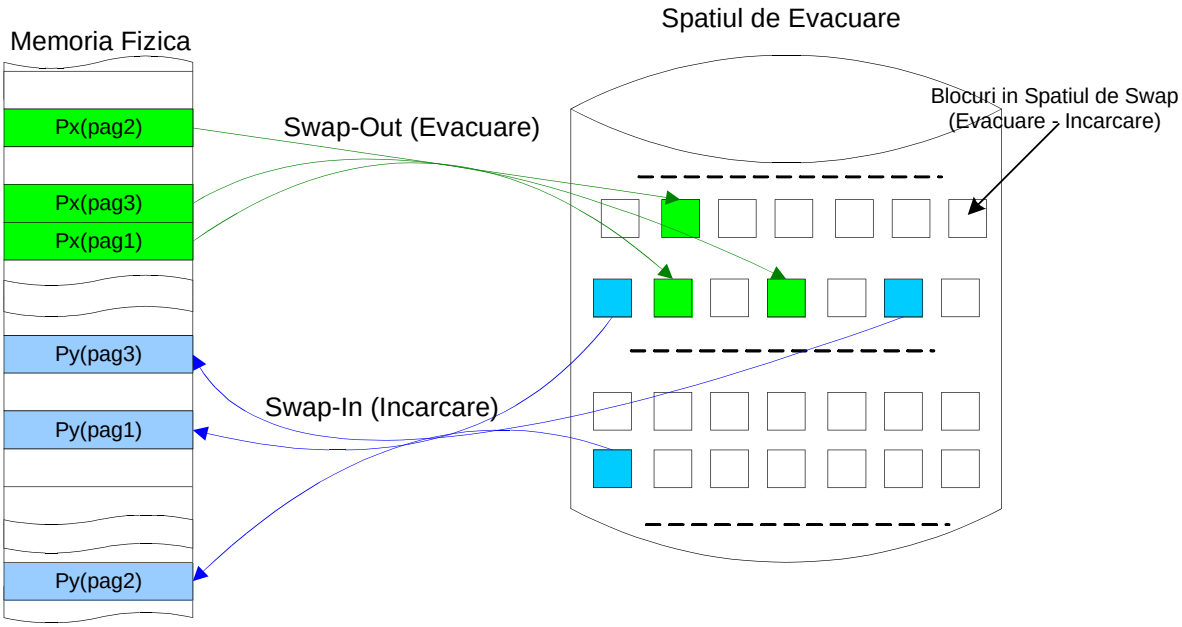
- Programul nu mai este constrans de o anumita dimensiune a *memoriei fizice*. Utilizatorii pot scrie programe pentru un spatiu foarte mare de adrese virtuale simplificand sarcina programarii.
- Deoarece un program ocupa in timpul executiei mai putina memorie, mai multe procese pot fi executate in acelasi timp. Acest lucru conduce la cresterea gradului de utilizare a CPU fara sa creasca timpul de raspuns.
- Operatiile de I/O pentru incarcarea proceselor in memorie sau pentru "swapping" manipuleaza un volum de date mai mic durata globala de executie a unui program utilizator fiind mai mica.

De acest mod de lucru in care se pot executa procese care nu sunt in intregime in memorie vor fi avantajati si utilizatorii si SO.
Memoria virtuala inseamna separarea memoriei logice "vazute" de utilizator de memoria fizica

(reala). Acest lucru permite ca programatorii sa foloseasca un spatiu larg de *memorie virtuala* in timp ce *memoria fizica* este foarte mica. In acest fel programarea devine mai simpla nefiind nevoie sa ne incadram intr-o dimensiune de memorie prestabilita. *Memoria virtuala* a condus la disparitia tehnicienilor de "reacoperire" (overlays).
Memoria virtuala se implementeaza de obicei printr-un mecanism numit "cerere de paginare", acest mecanism putand fi implementat in toate sistemele care utilizeaza metoda de alocare cu *paginare* sau *segmentare paginata*.
In cazul sistemelor care utilizeaza alocarea cu segmente se poate implementa un mecanism care implementeaza memeoria virtuala folosind mecanismul "cerere de segment". Implementarea *memoriei virtuala* in sistemele de management cu segmente fara paginare conduce la aparitia unor complicatii la inlocuirea segmentelor deoarece acestea au dimensiune variabila.

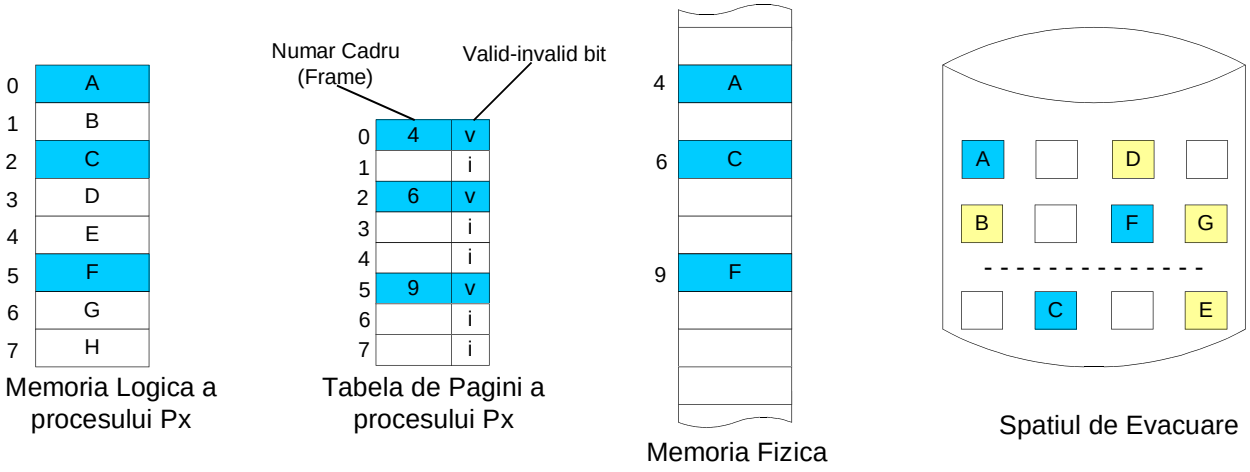
5.8.2Cererea de paginare.

Cererea de paginare (sau pagina) este un mecanism asemanator cu mecanismul utilizat de sistemele cu management cu paginare asociat cu evacuare-reincarcare (swapping).



In sistemele cu management cu paginare, programele (procesele) se afla initial pe disc. Atunci cand vrem sa executam un proces, acesta trebuie "incarcat" in memorie (swap-in), si el se va incarca in intregime (toate paginile sale).
De exemplu procesul Px care este format din 3 pagini, atunci cand este "evacuat" (swap-out) se vor transfera pe disc toate cele 3 pagini ale sale. Daca un alt proces Py trebuie incarcat in memorie de pe disc (swap-in) atunci toate paginile sale vor fi transferate in memorie.
Deci in sistemele cu paginare la evacuare sau reincarcare (swap-out, swap-in) se manipuleaza intregul proces.
In cazul in care se implementeaza un mecanism de *cerere de paginare* daca un proces trebuie incarcat in memorie atunci se va incarca in memorie numai acea pagina a procesului care va fi utilizata. Altfel spus mecanismul de *cerere de paginare* incarca in memorie numai acele pagini ale procesului care sunt necesare la un moment dat. In acest fel se evita incarcarea in memorie a paginilor care nu vor fi folosite niciodata facandu-se o economie de memorie si de timp prin reducerea numarului de operatii de evacuare-reincarcare.

Cererea de paginare (demand paging) necesita insa suport hardware suplimentar.



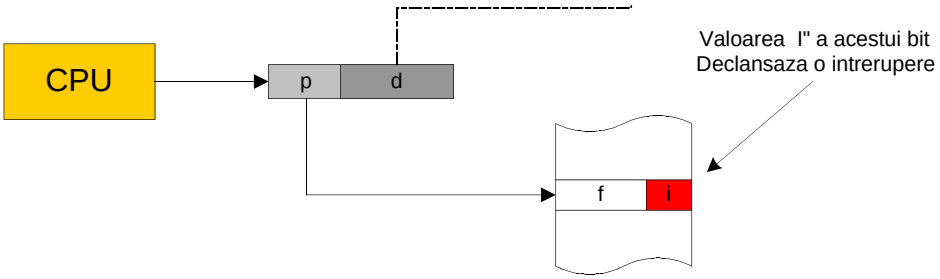
In *tabela de pagini* se adauga un bit numit *valid-invalid bit* fiecarei intrari in tabela. Deci fiecarei pagini a procesului i se va asocia un bit care va "arata" daca pagina respectiva se afla sau nu in memorie.
Pagina aflata in memoria fizica: Valid="v"; Pagina care nu se afla in memorie: Invalid="i";
Atunci cand o pagina este incarcata in memorie, bitul "*valid-invald bit*" este setat la valoarea "v".
Daca executia procesului se face in paginile aflate in memorie (marcate "v") lucrurile se petrec ca si cum intreg procesul ar fi in memorie si acest caz se numeste "*cerere de paginare pura*".
Niciodata nu se incarca in memorie o pagina pana cand nu se face referire la ea.
Atunci cand se face referire la o pagina marcata "i" (care deci nu se afla in memorie) se declanseaza o intrerupere de tip "pagina eronata" ("page-fault") care in continuare lanseaza in lucru mecanismul de incarcare a paginii "lipsa" in memorie si setarea bitului paginii respective la valoarea "v". Dupa aceasta se reia mecanismul de adresare la locatia care anterior
Teoretic este posibila si situatia extrema in care executia fiecarei instructiuni a unui program sa declanseze intreruperea "pagina-eronata". In acest caz la executia fiecarei instructiuni se va incarca o pagina in memorie, ceea ce va determina o performanta foarte scazuta a sistemului.
In realitate programele au o caracteristica care se numeste "referire locala". Aceasta inseamna ca cele mai multe instructiuni fac "referiri" la locatii care se afla in imediata vecinatate (local). Adica probabilitatea de referire la o locatie care se afla intr-o pagina absenta din memorie (marcata "i") este foarte mica.
Acest lucru face ca in realitate performanta sistemelor cu "cerere de paginare" sa fie rezonabila.
Suportul hardware pentru mecanismul "cerere de paginare" cuprinde:

Tabela de pagini cu proprietatea ca fiecarei intrari ii este asociat bitul "valid-inavalid" sau valori speciale pentru bitii de protectie care sa indice prezenta sau absenta paginii in memorie.

Memorie secundara in care se pastreaza paginile care nu se afla in memorie ale proceselor active. De obicei memoria secundara este o unitate de disc rapida, cunoscut si sub numele de "perifericul de evacuare reincarcare" (swap device) respectiv "spatiul de evacuare-reincarcare" ("swap space").

In afara suportului hardware sunt necesare mecanisme software suplimentare pentru implementarea mecanismului de *cerere de paginare*. Trebuie remarcat faptul ca daca o pagina este marcata "i" acest lucru nu are nici un efect daca nu se face acces la pagina respectiva. Atata timp cat procesul in executie acceseaza pagini rezidente in memorie (pagini cu bitul setat "v") executia se desfasoara "normal" ca si in cazul sistemelor fara *cerere de paginare*.

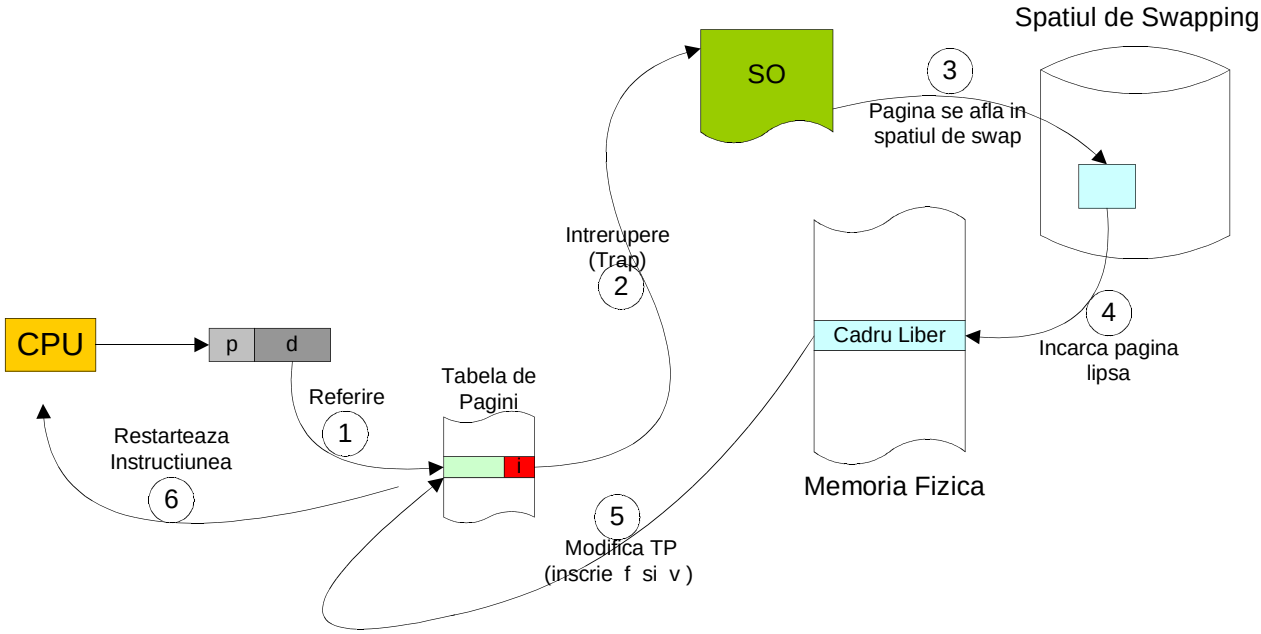
Daca insa procesul incearca sa faca un acces (citire sau scriere) la o adresa apartinand unei pagini care nu se afla in memorie se declanseaza o serie de activitati hardware si software care au ca finalitate incarcarea paginii respective in memorie.
Cum se petrec lucrurile in cazul unui acces la o adresa apartinand unui pagini "nerezidente" in memorie ?



Mecanismul de adresare va "citi" intrarea "p" corespunzatoare, din Tabela de Pagini care in acest caz are bitul "valid-invalid" marcat "i". Acest lucru va declansa o "intrerupere" de tip "adresa ilegala" numita in anumite sisteme "adresare incorecta" sau "pagina eronata".
In continuare se deruleaza un mecanism clasic de tratare a intreruperilor:

se "salveaza" contextul programului intrerupt (IP, PSW, registre, etc) in stiva;
se incarca *noua stare program* corespunzatoare rutinei din SO pentru tratarea intreruperilor de "adresare ilegala". Deoarece aceasta intrerrupere poate fi declansata si de o eroare de adresare clasica, rutina verifica cauza care a declansat aceasta intrerupere. Exista doua posibilitati:
a) Adresa este invalida dintr-o eroare de programare si in acest caz procesul respectiv este intrerupt afisandu-se un mesaj de eroare indicand utilizatorului"eroare de adresare";
b) Este o adresare intr-o pagina care nu se afla in memorie, caz in care se desfasoara schema de *cerere de paginare* propriu-zisa.
Se cauta un *cadru* (Frame) liber de memorie fizica;
Se lanseaza cererea de transfer pentru pagina dorita (aflata pe disc) in memoria fizica in *cadrul* liber gasit anterior;
Se asteapta terminarea transferului de pe disc in memoria fizica si apoi se modifica bitul valid-invalid al paginii incarcate in memorie din valoarea "i" in "v" marcand faptul ca pagina se afla in memorie;
SO reface contextul procesului "salvat" la aparitia intreruperii si se restarteaza instructiunea intrerupta de "adresare ilegala". De data aceasta pagina in care se face adresarea fiind in memorie accesul la locatia de memorie se face normal.

De remarcat faptul ca procesul este reluat exact din locul in care s-a produs "eroarea de adresare" si exact in conditiile initiale deoarece la intrerupere se salveaza registrele, IP, PSW etc iar la restartarea continutul lor este refacut de catre SO (restaurarea contextului initial)
Deci in concluzie algoritmul de alocare a memoriei bazat pe *Memoria Virtuala* foloseste acest mecanism numit *cerere de paginare.*
Cererea de paginare asigura mecanismul prin care atunci cand un proces incearca sa acceseze locatii care nu se afla in memorie, declanseaza o intrerupere (TRAP) care deturneaza executia spre sistemul de operare care aduce pagina dorita in memorie restartand apoi procesul.



Este posibil (in cazuri extreme) ca sa lansam un proces care nu are nici o pagina in memorie. Tentativa de a lansa procesul in executie declanseaza imediat o intrerupere "page-fault" care determina incarcarea primei pagini in memorie. Se continua executia si la urmatoarea instructiune poate sa se intample acelasi lucru (page-fault). Astfel este posibil sa se incarce in memorie, pe rand, toate paginile procesului dupa care in continuare sa se petreaca lucrurile fara nici o intrerupere de tip "page-fault" , aceasta situatie numindu-se *cerere de paginare pura*. Niciodata nu se incarca in memorie o pagina pana ce ea nu este necesara.
In cazul *cererilor de paginare* exista cerinte specifice de natura arhitecturala pentru a ca dupa tratarea intreruperii "page-fault" sa fie posibila relansarea executiei instructiunii.
Aceste restrictii se refera la sistemele calculator care accepta moduri speciale de adresare, cum ar fi modurile autodecrement sau autoincrement.

5.8.3 Performanta *cererii de paginare.*

Cererea de paginare poate afecta performantele in ansamblu sistemului calculator. Vom incerca sa evaluam "timpul efectiv de acces" in cazul "cererii de paginare". Timpul de acces la memoria principala (operativa) este pentru sistemele actuale in jurul valorii de 10ns.

T_{acces_mem}=10 ns =0,01 ì s

Atat timp cat nu avem "page-fault" deci accesul se face in pagini prezente in memorie, timpul efectiv de acces este acelasi cu timpul de acces la memorie.
In cazul in care insa, apare un acces la o pagina care ne se afla in memorie ("page-fault"), trebuie mai intai incarcata pagina respectiva in memorie si apoi se executa accesul la informatia dorita.
Notam cu "p" probabilitatea de a se intampla "page-fault" cu 0W pW1.

$T_{efectiv\_acces} = (1-p)T_{acces\_mem} + p * T_{tratare\ "page-fault"}$

Trebuie deci sa evaluam "timpul de tratare a page-fault-lui".
Daca apare o eroare de tip "page-fault" se deruleaza urmatoarea secventa de activitati:
1. "Trap" la sistemul de operare (rutina de tratare intrerupere "page-fault")
2. Salvarea registrelor utilizator si starea procesului (contextul procesului)
3. Determinarea (verificarea) faptului ca referinta la pagina a fost legala si determinarea locatiei

unde se afla pagina pe disc.

4. Lansarea operatiei de transfer de pe disc intr-un *cadru* liber de memorie
   - inscrierea in "coada de cereri" a perifericului a cererii efective de transfer;
   - executia operatiei de transfer propriu zisa care inseamna urmatoarele activitati hard:
     a) pozitionarea capatelor de citire (cca 8 ms)
     b) cautarea informatiei (cca 10 ms)
     c) transferul propriu zis al paginii de pe disc in memorie (cca 1 ms).

Aceasta operatie se desfasoara pornind de la presupunerea ca exista un cadru liber in memorie.

5. Pe durata desfasurarii transferului CPU poate fi alocat unui alt proces (planificatorul de procese optional);
6. Se declansaza "intreruperea" data de unitatea de disc la sfarsitul transferului;
7. Salvarea contextului procesului activ in momentul aparitiei intreruperii (registrele si starea procesului)
8. Tratarea intreruperii disc.
9. Actualizarea *Tabelei de pagini* a procesului (numarul *cadrului* si setarea bitului *valid-invalid*).
   Actualizarea *Tabelei de procese* prin trecerea indicatorului de stare procesuluila valoarea "Gata".
   Planificatorul trece in starea "Executie" urmatorul proces din lista de procese active.
10. Asteapta ca CPU sa fie alocat din nou procesului.
    Atunci cand *planificatorul* trece din nou procesul in starea "Executie" :
11. Restaureaza contextul procesului (registre si starea procesului) si relansarea instructiunii care a declansat "page-fault-trap".

Nu intotdeauna se deruleaza toti acesti pasi. De ex. pasul 5 in care CPU este alocat unui alt proces pe durata transferului, conduce la cresterea gradului de multiprogramare dar consuma un timp suplimentar pentru relansarea procesului. Daca se renunta la aceasta strategie se poate elimina acest timp de "regie" suplimentar..

In orice caz in toata aceasta secventa exista trei componente majore care apar indiferent de strategie:

tratarea intreruperii "page-fault";
citirea paginii absente;
restartarea procesului;

din care prima si cea de a treia inseamna executia catorva sute de instructiuni la nivelul unitatii centrale care ca durata pot fi aproximate la circa 1 milisecunda. Deci timpul de rezolvare a unei cereri de "demand paging" poate fi aproximat astfel: (timp de tratare " page-fault ")

deplasarea capetelor de citire disc: 8 milisecunde.
timpul de cautare a informatiei: 10 milisecunde.
timpul de transfer propriu zis 1 milisecunda.
timpul de executie a tratare a intreruperii si restartarea procesului:1 milisec.

Total : aprox. 20 milisecunde.

Inlocuind in formula

$$T_{acces\_efectiv}=(1-p)*T_{acces\_mem}+ p * T_{tratare\ "page-fault"}$$

obtinem

$$T_{acces\_efectiv} =(1-p)*10nanosecunde+ p * 20milisecunde$$

si transformand totol in ns:

$$T_{acces\_efectiv}=10 \quad 10 * p + 20.000.000 * p= 10 + 19.999.990 * p \text{ (nanosecunde)}$$

deci

$$T_{acces\ efectiv} = 10 + 19.999.990 * p \text{ (ns)} = 10 + 20.000.000 * p \text{ (ns)}.$$

unde vedem ca timpul de acces efectiv ($T_{acces\_efectiv}$) este direct proportional cu probabilitatea de aparitii a "cererilor de paginare".

Daca de exemplu unul din 1000 de accese la memorie declanseaza "cererea de paginare" timpul efectiv de acces este de cca 20 microsecunde.

p = 0,001

$T_{acces\_efectiv}$ = 10 + 19.999.990 * 0,001(nanosecunde)=

= 10 + 19.999,9 = 20.000,9ns ≅ 20ì s = 2000 * (0,01)ì s = 2000 * (10ns).

Deci timpul efectiv de acces de circa 20ì s inseamna o incetinire de cca 2000 ori fata de accese la memorie fara "cerere de paginare" la care timpul de acces este de cca 10ns.
Daca dorim sa calculam probabilitatea "p" astfel incat sa obtinem o "degradare" a performantelor cu 10% fata de accesul la memorie, vom calcula astfel:
Un timp de acces mai lent cu 10% inseamna 10ns+10%=11ns.

11 > 10 + 20.000.000 * p sau 1 > 20.000.000 * p sau p < 1 / 20.000.000

Deci **$p < 0,5 * 10^{-7}$**

Asta inseamna sa avem 1 acces la memorie care declanseaza o "cerere de paginare" din 20 milioane de accese fara "cerere de paginare".
Deci este important sa mentinem foarte scazuta rata cazurilor de acces cu "cereri de paginare" (demand paging) altfel durata prelucrarii se mareste nepermis de mult.
Un alt aspect referitor la performanta "cererii de paginare" o reprezinta modul in care este organizat spatiul disc unde se afla procesul (paginile sale).
Spatiul "swap" (evacuare reincarcare) este in general mai rapid decat utilizarea sistemului de fisiere SO , deoarece spatiul de "swap" este alocat in blocuri de dimensiuni mai mari si nu sunt utilizate metode de alocare indirecta (index de fisiere, etc).

5.8.4. Inlocuirea paginilor.

In cele discutate anterior am vazut ca rata adresarilor in care apare "page fault" nu este o problema deorece fiecare pagina este adresata cu "page fault" cel mult odata atunci cand se face o referinta la o locatie in pagina respectiva pentru prima oara. Aceasta reprezentare poate insa sa nu fie foarte realista din cauza ca intotdeauna am presupus ca exista un *cadru* liber de memorie atunci cand se face o cerere de paginare.
Consideram un exemplu:
Un proces care este format din 10 pagini dar in momentul actual foloseste numai 5 pagini.
Remarcam ca utilizarea *Cererrii de paginare* creaza o economie de timp datorita faptului ca celelalte 5 pagini nu vor fi incarcate in memorie.
Presupunem de exemplu ca memoria operativa dispune de 40 *cadre* (*frame*-uri) de memorie fizica.
Teoretic, putem creste gradul de multiprogramare putand incarca si executa 8 procese (daca presupunem ca au tot dimensiunea de 10 pagini si numai 5 folosite) in loc de 4 procese fara *cerere de paginare*.
Se pune problema cat trebuie sa fie numarul maxim de procese care pot fi active simultan (gradul de multprogramare).
De exemplu daca vom stabili ca sistemul lucreaza la gradul 6 de multiprogramare atunci 6 procese (de 10 pagini dar numai 5 folosite) vor ocupa in medie 6 * 5=30 cadre memorie.
In aceasta situatie raman 10 cadre care le consideram de rezerva.
De ce sa pastram aceste cadre de rezerva ? Pentru ca atunci ca un exemplul nostru procesele aveau 10 cadre si numai 5 folosite "in cele mai multe din cazuri" dar nu intodeauna.
Este posibil ca procesele pentru seturi particulare de date sa foloseasca toate cele 10 pagini rezultand o situatie in care vor fi necesare 60 de cadre in timp ce numai 40 sunt disponibile.
In situatiile acestea (statistic foarte rare) numi putea avea 6 procese active simultan.
Aceasta situatie nefavorabila poate sa apara cu o probabilitate cu atat mai mare cu cat se creste

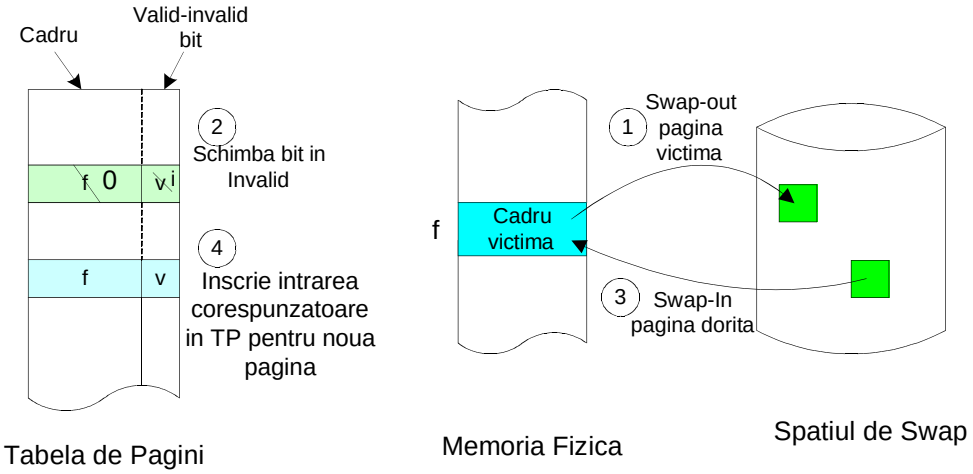
gradul de multiprogramare sau altfel spus cu cat dimensiunea de memorie utilizata in medie de procesele active se apropie de dimensiunea memoriei fizice.
Din aceste motive in exemplul nostru gradul de multiprogramare se alege 6 cu toate ca el putea fi ales 7 sau 8.
Situatia in care SO in urma unei intreruperi "fault page", trebuie sa incarce de pe disc in memorie o pagina a unui proces si nu gaseste in memoria fizica nici un *cadru* (frame) liber se numeste "*Supraalocarea*" (overallocation).
In aceasta situatie SO poate actiona in mai multe feluri:

termina procesul care a solicitat *cererea de paginare*;

evacueaza procesul (swap-out) eliberand toate cadrele ocupate de el in memorie, dar reducandu-se gradul de multiprogramare;

aleage dupa un anumit criteriu (algoritm) o pagina din memorie care sa o evacueze pe disc pentru a face loc paginii care trebuie incarcate (Aceast mecanism numindu-se *inlocuire pagina*).

Bineinteles ca practic nu pot fi luate in considerare decat cea de a doua si a treia varianta.
Pentru inceput o sa detaliem "mecanismul numit *inlocuirea paginii*. In aceasta varianta daca nu se gaseste un *cadru* liber SO va cauta in memorie un cadru care nu este utilizat (pagina din el nu este activa) si il va elibera. Eliberarea cadrului se poate face copiind continutul sau pe disc si schimband "indicatorii" din toate tabelele aratand ca pagina nu mai este in memorie. *Cadrul* eliberat poate fi utilizat acum pentru "primirea" paginii procesului care a declansat *cererea de pagina.*



Schema de *inlocuire pagina*

Rutina de serviciu SO discutata anterior pentru tratarea intreruperii "page-fault" se va modifica prin includerea schemei de *inlocuirea paginii.* Aceasta implica executarea urmatoarelor actiuni:

1. Gasirea locatiei paginii solicitate, pe disc in imaginea executabila a procesului;
2. Gasirea unui *cadru* liber in memoria fizica:
   a. Daca exista un cadru liber ” acesta se va utiliza (trece la punctul 3)
   b. Daca nu exista un cadru liber, aplica algoritmul de *inlocuire a paginii* pentru selectarea cadrului ce se va evacua. Acesta se numeste *cadrul victima*;
   c. Scrie (copiaza) *pagina victima* pe disc si modifica corespunzator tabelele care tin evidenta paginilor si a cadrelor;
3. Citeste (copiaza) pagina dorita in cadrul liber si modifica corespunzator tabelele de evidenta a paginilor si a cadrelor;

4. Restarteaza procesul intrerupt de data aceasta existand in memorie pagina in care se face adresarea;

Trebuie remarcat ca in cazul executarii mecanismului de *inlocuire de pagina* , va avea loc transferul a 2 pagini (una se evacueaza:swap-out, si alta incarca: swap-in), ceea ce dubleaza timpul de executie al serviciului "page fault"..

Exista posibilitatea reducerii acestui timp prin utilizarea asa numitului *bit de modificare* (sau *bit de murdarire*). Fiecare *pagina* aflata in memorie are asociat un bit care initial este "0". Daca in pagina respectiva are loc o modificare atunci *bitul de modificare* va fi setat (va capata valoarea "1"). Atunci cand o pagina trebuie evacuata (swap-out) se verifica *bitul de modificare*. Daca acesta are valoarea "0" atunci *pagina* nu mai este necesar sa fie evacuata efectiv deoarece imaginea sa este identica cu cea de pe disc.

In acest fel se reduce timpul de *inlocuire de pagina.* Daca *pagina victima* este de tip *read-only* (citeste-numai) care nu poate fi modificate in timpul executiei deasemeni nu va mai fi necesara "evacuarea" paginii pe disc deoarece ea este identica cu "imaginea" sa de pe disc.

Schema de *inlocuire de pagina* este foarte importanta in *cererea de paginare* deoarece creaza posibilitatea executiei programelor a caror dimensiune a memoriei virtuale este mai mare decat memoria fizica.

De exemplu putem executa un proces de 20 pagini intr-un spatiu de memorie fizica de 10 cadre.

Asa cum am vazut implementarea *cererii de paginare* implica rezolvarea a 2 probleme majore:

alocarea *cadrelor*;
inlocuirea *paginilor*.

Atunci cand in memorie avem mai multe procese trebuie luata urmatoarea decizie:

Cate cadre se aloca fiecarui proces?

Acest lucru este facut de "algoritmul de alocare cadre ".

In continuare atunci cand este necesara *inlocuirea paginilor* trebuie selectate *cadrele* care trebuie inlocuite, acest lucru fiind realizat de " algoritmul de inlocuire pagina ".

Impotanta acestor algoritmi este majora avand in vedere faptul ca operatiile de intrare-iesire (transferurile cu discul magnetic) sunt mari consumatoare de timp.

5.8.5. Algoritmi de inlocuire pagina

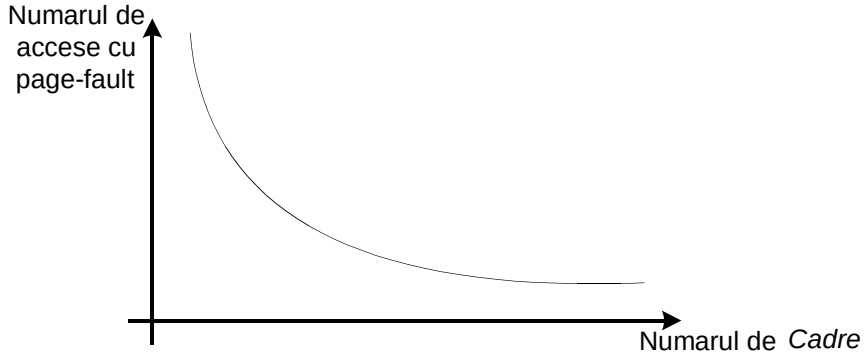
Exista mai multi algoritmi de *inlocuire pagina*. Fiecare SO foloseste un singur algoritm de inlocuire. Alegerea unui anumit algoritm de inlocuire se face astfel incat sa avem cea mai mica rata de "page-fault" (*page-fault rate*). In acest caz devine foarte importanta procedura de evaluare a performantelor unui algoritm de *inlocuire pagina.*
Algoritmii se evalueaza printr-o simulare utilizand diverse secvente de adrese (refeinte la memorie) numite *siruri de referinta*, urmarind numarul de cazuri in care apare "page-fault".
Cum se construieste acest *sir de referinta*? De obicei *sirul de referinta* se genereaza:

fie aleator;
fie se inregistreaza pe un sisitem real sirul (secventele) de "referinte" la memorie.

In cazul in care se inregistreaza referintele la memorie pe un sistem real se produce un numar foarte mare de date (de ordinul milioanelor de adrese pe secunda).
Pentru reducerea lor se tine cont de doua lucruri:

pentru o dimensiune data a *paginii* (data prin constructia SC hard) nu trebuie inregistrate decat numerele pagina "p" nefiind interesanta intreaga adresa;
daca se face referinta la pagina "p" niciodata nu se poate ca urmatoarea referinta daca se face tot la pagina "p" sa produca "page-fault".

De exemplu daca vom inregistra pentru un anume proces urmatoarea secventa de referinte la memorie (adrese referinta): 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105.
Daca dimnensiune paginii este 100Bytes, atunci ultimii 2 biti for reprezenta adresa (deplasarea) in pagina iar primii 2 biti adresa paginii.
In aceasta situarie tinand cont de observatiile anterioare *sirul de referinta* va fi redus la urmatorul sir de adrese de pagina: 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1.
Pentru determinarea numarului de accese care produc "page-fault" pentru un sir particuar de referinte si un anumit algoritm de *inlocuire pagina*, va fi nevoie sa cunoastem si numarul de *cadre* disponibile (libere). Ne vom astepta ca daca numarul de *cadre* disponibile creste, numarul de accese care produc "page-fault" sa descreasca.
Pentru *sirul de referinta* considerat anterior, daca numarul *cadrelor* de memorie disponibile este 3 sau mai mare, vom avea numai 3 accese cu "page-fault", cate unul pentru fiecare intaia referinta in pagina. Cazul extrem este atunci cand avem un singur cadru disponibil, caz in care vom avea cate o *inlocuire de pagina* la fiecare referinta rezultand 11 accese cu "page-fault". Intuitiv ne asteaptam la urmatoarea dependenta intre numarul de accese care produc "page-fault" si numarul de cadre disponibile.

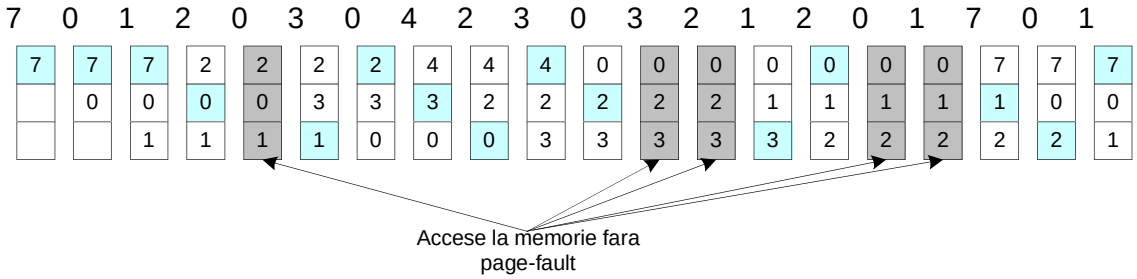


Graficul dependentei acceselor cu page-fault
de numarul cadrelor

Cu cat creste numarul de *cadre* cu atat numarul de accese cu "page-fault" scade catre un anumit nivel minim.
In continuare pentru a ilustra si evalua algoritmii de *inlocuire pagina* vom utiliza *sirul de referinta*:
7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
pentru o memorie fizica cu 3 *cadre* disponibile.
Daca observam acest *sir de referinta* vom vedea ca el se refera la un proces cu 8 pagini (0-7) din care se fac referinte numai la 6 dintre ele (0,1,2,3,4,7).

5.8.5.1 Algoritm FIFO.

Un algoritm simplu este FIFO (first-in, first-out). Acest algoritm tine cont de "vechimea" paginii in memerie. Atunci cand o pagina trebuie inlocuita este aleasa ca pagina "victima" cea mai veche pagina aflata in memorie. Nu este neaparat necesar a se inregistra timpul cand pagina a fost adusa in memorie ci se poate rezolva aceasta cerinta printr-o lista de tip FIFO care pastreaza numarul paginii aflate in memorie in ordinea cronologica a incarcarii lor in memorie. Se va inlocui intotdeauna pagina aflata la capatul listei FIFO, iar pagina incarcata se va adauga la "coada" listei FIFO (celalalt capat).
Vom exemplifica acest algoritm acest algoritm pe *sirul de referinta* de test, considerand ca avem 3 cadre disponibile.



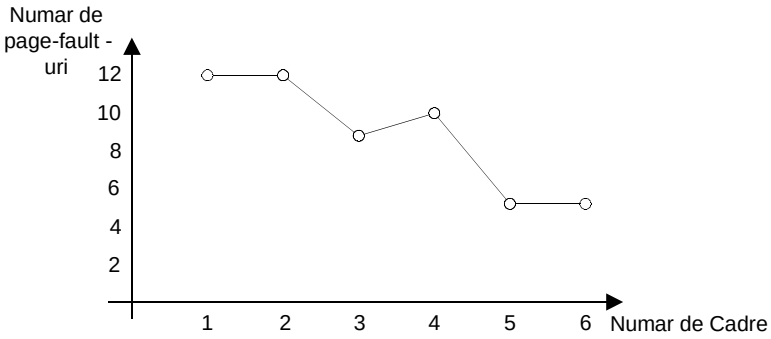
Algoritmul FIFO de inlocuire pagina

In acest exemplu la un numar de 20 de accese la memorie vor aparea 15 cazuri de adresare la care se produce "page-fault" care determina utilizarea mecanismului de *inlocuire de pagina*. Numai 5 accese se vor face la *pagini* care se afla deja in memorie.
Primele trei accese in mod automat vor determina aparitia adresarii cu "page-fault" deoarece nici-o pagina nu se afla in memorie. Aplicarea algoritmului FIFO determina ca in continuare atunci cand pagina adresata nu se afla in memorie se va inlocui pagina care se afla de cea mai multa vreme in memorie.
Algoritmul FIFO de *inlocuire pagina* este usor de implementat. Totusi acest algoritm nu furnizeaza intotdeauna rezultate bune. De exemplu pagina "victima" (cea mai veche din memorie) poate contine un modul al programului care este des utilizat. Aceasta inseamna cu nu intodeauna "vechimea" paginii in memorie este cel mai bun "indicator" al paginii ce trebuie inlocuite.
Pentru a evidentia toate problemele care apar la utilizarea algoritmului FIFO vom alege un alt *sir de referinta*:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

Utilizand acest sir de referinta si vom calcula numarul de adresari care produc "page-fault" pentru situatiile cand avem 1, 2, 3, 4, 5 si 6 *cadre* disponibile.
Pentru a fi mai sugestiv vom reprezenta grafic relatia intre numarul de adresari care produc "page-fault" in functie de numarul de cadre disponibile.

Vom observa urmatoarea situatie:



Corelatia intre numarul de accese care produc "page-fault" si numarul cadrelor disponibile.

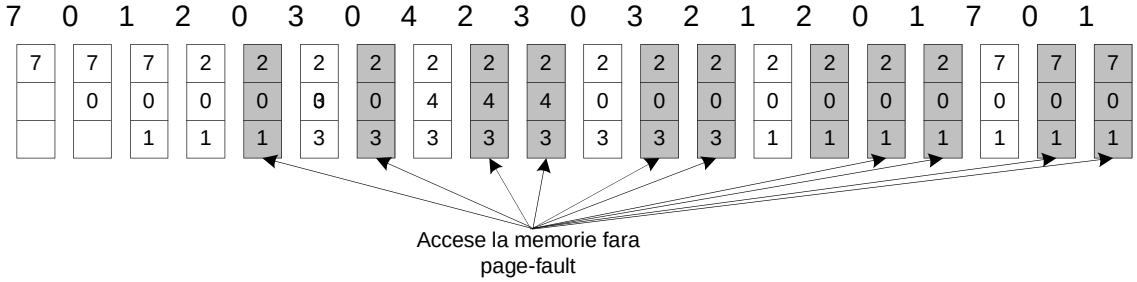
Remarcam un paradox. Numarul de adresari in care apare "page-fault" pentru un numar de 4 cadre disponibile este "10" mai mare decat in cazul a 3 cadre (9 "page-fault"-uri). Acest rezultat neasteptat este cunoscut sub numele de "anomalia lui Belady".
Anomalia lui Belady reflecta faptul ca pentru anumiti algoritmi de *inlocuire de pagina* numarul de adresari cu "page-fault" sa creasca chiar daca creste numarul de cadre alocate.
Ne-am astepta ca, alocand mai multa memorie unui proces (mai multe *cadre*) sa apara mai putine adresari la memorie in care se intampla "page-fault" deci sa creasca si nu sa scada performanta sistemului. In urma cercetarilor recente s-a demonstrat ca aceasta ipoteza nu este adevarata, anomalia Belady demonstrand acest lucru.

5.8.5.2 Algoritmul "Optimal"

Un algoritm optimal nu trebuie sa sufere de "anomalia Belady". Un algoritm optimal de inlocuire de pagina exista si el a fost numit OPT sau MIN. Acest algoritm este deasemenea simplu:

Se inlocuieste pagina care nu va fi utilizata cea mai lunga perioada de timp.

Acest algoritm garanteaza cea mai scazuta rata de adresari "page-fault" pentru un numar fix de cadre. Acest algoritm este insa dificil de implementat deoarece presupune sa cunoastem "apriori" *sirul de referinta*. Deci trebuie sa cunoastem inainte de a executa procesul ce referinte se vor face la memorie. Lucru care este foarte dificil sau chiar imposibil pentru procesele care se executa in sistem.
Totusi acest algoritm este foarte util pentru ca ne permite sa evaluam alti algoritmi raportat la algoritmul OPT.
Algoritmul OPT are la baza ideea ca atunci cand trebuie sa eliminam o pagina din memorie (sa o inlocuim) pagina "victima" trebuie sa fie aceea care va fi folosita (referita) cel mai tarziu.

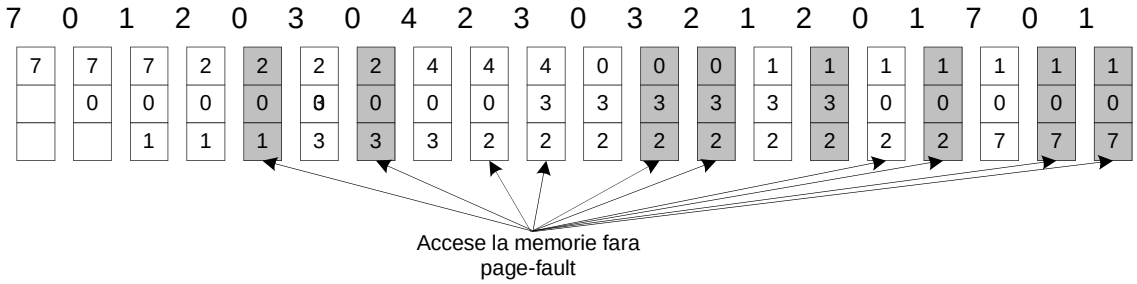


Algoritmul Optimal de inlocuire pagina

Algoritmul optimal de inlocuire aplicat sirului nostru de referinte pentru 3 cadre disponibile va produce 9 "page-fault"-uri. Primele 3 referinte determina "page-fault" pentru incarcarea primelor 3 pagini ale procesului. Referinta a 4-a se face la pagina 2 care nu se afla in memorie declansand o "inlocuire de pagina". Care va fi in acest caz pagina inlocuita din memorie?
Va fi evacuata pagina 7 care din cele 3 pagini prezente in memorie (7,0,1) va fi referita cel mai tarziu (in referinta numarul 18). S.a.m.d.
Deosebirea majora intre algoritmul FIFO si algoritmul OPT este faptul ca la algoritmul FIFO se "priveste" in urma iar OPT se "priveste" inainte. Algotitmul FIFO ia in considerare timpul in trecut cand pagina a fost incarcata in memorie iar algoritmul OPT timpul cand pagina va fi referita din nou (in viitor).

5.8.5.3. Algoritmul LRU (Least Recently Used)

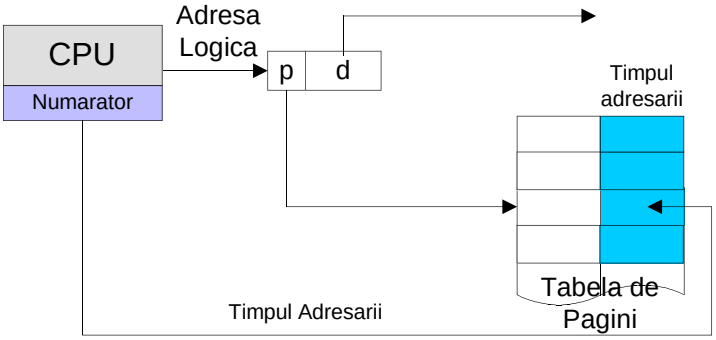
Acest alogoritm ia in considerare timpul de cand o pagina nu a mai fost referita. Acest algoritm inlocuieste pagina care nu a fost utilizata de cel mai mult timp (cea mai putin recent utilizata). Spre deosebire de algoritmul FIFO, algoritmul LRU are in vedere timpul de la ultima referire si nu timpul de cand pagina se afla in memorie.



Algoritmul LRU de inlocuire pagina

Algoritmul LRU aplicat aceluiasi *sir de referinta* produce 12 adresari "page-fault" din totalul de 20 referinte la memorie. Pentru referintele 1-7 inlocuirile de pagina sunt identice cu cele de la algoritmul OPT, dar acest lucru este numai intamplator. Pagina "victima" in cazul algoritmului LRU se alege pagina care nu a fost referita de cel mai de mult timp. De exemplu la cea de a 8-a referinta la memorie se face o cerere de acces in pagina "4" care nu se afla in memorie. Dintre paginile existente in memorie se va gasi ca cea care nu a fost referita de cel mai mult timp este pagina "2". Deci pagina "2" va fi inlocuita de pagina "4".
Algoritmul LRU pentru inlocuire pagina este considerat suficient de bun insa el necesita un mecanism hardware suplimentar.
Acest mecanism hardware suplimentar inseamna trebuie sa ofere o cale simpla si rapida pentru a determina care este pagina care nu a fost utilizata (referita) de cel mai mult timp. Aceasta inseamna de a avea posibilitatea sa "ordonam" refeririel la paginile de memorie dupa momentul (timpul) cand s-au produs.
Mecanismul hard pentru "ordonarea" in timp a utilizarii paginilor foloseste fie "Numaratoare" (Contoare) asociate fiecarei intrari in tabela de pagini fie "liste" in care se introduc paginile utilizate.
Metoda "contoarelor".
Necesita pe de o parte ca fiecarei intrari in TP sa i se adauge un camp care va contine "timpul utilizarii (adresarii)" iar pe de alta adaugarea CPU-ului a unui registru (numarator) care sa fie incrementat la fiecare referinta la memorie reprezentand momentul adresarii.
"Timpul adresarii" nu trebuie inteles ca un "timp" absolut. El este un numar care prin valoare lui

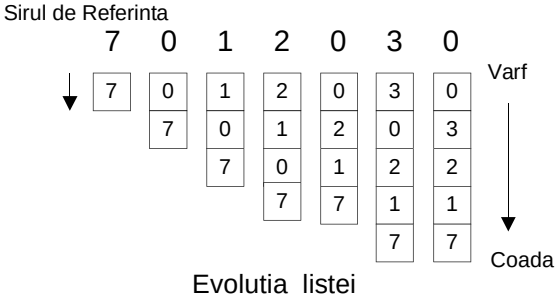
indica a cata adresare la memorie a fost facuta pentru procesul respectiv oferind o informatie asupra "ordinei" in care s-au facut referintele la memorie.



Algoritmul de *inlocuire de pagina prin* compararea valori "timpului adresarii" pentru toate paginile afalate in memorie determina care este pagina la care s-a facut cea mai "putin recenta" referire. Pagina cu cel mai mic "timp al adresari" este pagina la care nu s-a facut referire de cel mai mult timp (cea mai putin recenta referire).Acest "timp al adresarii" este pastrat si pe perioada cand procesul nu este planificat pentru executie (se afla in starea "gata" si starea "blocat"). Deasemeni se tine cont de situatiile cand apare o depasire (overflow) a valorii "numaratorului".

Metoda "listei" (stivei).

Foloseste o *lista* in care la fiecare adresare se extrage din *lista* numarul paginii referite (oriunde s-ar afla in cadrul *listei*) si se introduce in "varful" (Top) ei. In acest fel in "varful" *listei* se vor afla intodeauna cele mai recent utilizate pagini iar la "coada" *listei* cele mai putin recent utilizate pagini. Algoritmul de *inlocuire pagina* va alege intodeauna pagina ce trebuie inlocuita din "coada" *listei*.



Avantajul metodei rezida din faptul ca nu necesita nici-o comparatie. Pagina ce se va inlocui va fi intodeauna cea afla in "coada" listei. Aceasta metoda presupune insa un mecanism mai complicat de "gestionare" a listei. Eliminarea unei pagini din lista (ea putandu-se afla oriunde in lista) si plasarea ei in "varful" listei presupune utilizarea listelor de tip "dublu inlantuit" cu pointeri pentru "varful" si "coada" listei. Metoda "listei" se preteaza foarte bine la o implementare software sau in "microcod".
Algoritmii OPT si LRU nu sufera de "anomalia lui Belady".
Trebuie insa remarcat ca nici-o implementare a algoritmului LRU care nu este "sustinuta" de suport hardware nu este posibila. Orcare din metodele de determinare a paginii ce trebuie inlociuita (fie cu "numaratoare" fie cu "lista") presupune utilizarea mecanismului de "intrerupere" pentru a putea actualiza structurile de date corespunzatoare. Acest lucru conduce la o marire considerabila a duratei fiecarui acces cu "page-fault".

5.8.5.4 Algoritmi LRU cu aproximare.

Foarte putine SC ofera suportul hardware suficient pentru implementarea algoritmului LRU de *inlocuire pagina*. Dificultatea utilizarii algoritului LRU survine din faptul ca SC hardware trebuie sa furnizeze mijloacele pentru ordonarea in timp ale referintelor de memorie. Acest lucru creste complexitatea SC hardware deci si costurile.
Din aceasta cauza de multe ori se prefera utilizarea unor mecanisme mai simple de mecanisme hardware care ofera insa o "ordonare aproximativa" a referintelor la memorie ceeace face ca in aceste cazuri algortmii sa se numeasca LRU cu aproximare. Un asemenea mecanism hardware foloseste asa numitul "bit de referire" (numit si "bit R") asociat fiecarei intrari a TP. Acest bit este "setat" de catre SC hardware oridecate ori se face o referire la o locatie (adresa) a paginii respective, indiferent daca este citire sau scriere in memorie.
Acest mecanism functioneaza astfel. Initial odata cu lansarea procesului in executie SO va "sterge" toti "bitii de referire" din TP. Pe masura ce se executa procesul, SC hardware va "seta" acei "biti de referire" ai paginilor in care se face acces. In acest fel in orce moment putem determina foarte simplu care sunt paginile la care s-a facut referire si paginile la care nu s-a facut nici-o referire.
Este evident ca utilizand "bitul de referire" nu vom avea o "ordonare" completa a utilizarii paginilor. Vom avea o ordonare partiala (aproximativa) in pagini care au fost folosite si pagini care nu au fost folosite. Aceasta ordonare partiala este folosita de multi algoritmi de *inlocuire pagina* care aproximeaza algoritmul LRU de inlocuire.
Algoritmi aditionali la metoda cu "bit de referire".
Putem imbunatatii aproximarea in ordonarea facuta cu ajutorul "bitului de referire" daca vom "inregistra" periodic, la anumite intervale de timp valorile "bitului de referire". Acest lucru se poate face cu ajutorul unui registru de deplasare asociat suplimentar pentru fiecare pagina in TP. Acest "registru de istoric" al referirilor are de obicei are 8 biti.
La intervale regulate de timp, de exemplu 50 milisecunde, cu ajutorul unui numarator care da o intrerupere la acest interval, SO preia controlul si introduce prin deplasare, in bitul cel mai semnificativ al fiecarui "registru istoric" valoarea "bitilor de referire" cititi in acel moment. Astfel acest "registrul istoric" va contine "istoricul" utilizarii fiecarei pagini a procesului pe o perioada de 8 X 50 milisececunde (daca registrul are 8 biti). Daca "registrul istoric" contine valoarea binara "00000000" inseammna ca pagina respectiva nu a fost utilizata nici-o data in ultimele 8 perioade de timp. Daca el contine valoarea "11111111" asta inseamna ca pagina respectiva a fost utilizata cel putin o data in fiecare din ultimile 8 perioade.
O pagina a carui valoare a "registrului istoric" este "10101100" a fost ma i recent utilizata decat o pagina a carui valoare a "registrului istoric" este de exemplu "01110101". Altfel spus intodeauna pagina LRU este acea pagina a carui valoare a "registrului istoric" este cea mai mica considerata ca valoare "intreg fara semn".
Valorile acestor "registre istoric" nu sunt garanta diferite. Deci uneori pot exista mai multe pagini al carui valoare a "regsitrului istoric" este identica si cea mai mica (mai multe pagini LRU). In acest caz se poate alege pagina "victima" orcare dintre acestea sau se poate aplica intre ele un algoritm FIFO.
Numarul bitilor "registrului istoric" poate varia de la sistem la sistem. El se alege astfel incat timpul de actualizare al acestora sa fie cat mai mic.
Algoritmul "”ansei a doua".
Este o combinatie a algoritmului FIFO de inlocuire cu utilizarea "bitului de referire", adica combina criteriul de "vechime" a paginii in memeorie dat de lista FIFO cu criteriul "vechimii" utilizarii paginii dat de "bitul R". Algoritmul "sansei a doua" functioneaza astfel:

Atunci cand o pagina a fost selectata pentru a fi inlocuita pe baza pozitiei sale in lista FIFO, se inspecteaza si "bitul R". Daca valoarea "bitului R" este "0" atunci pagina este inlocuita. Daca "bitul r" este "1" atunci pagina nu este inlocuita ci, i se acorda o a doua sansa si se selecteaza urmatoarea pagina din lista FIFO. In acelasi timp daca unei pagini i-a fost acordata o "a doua sansa" bitul sau de referire este sters ("0"). Acest lucru face ca la urmatoarea selectie a ei ca pagina "victima" daca pagina nu a fost referita intre timp ea sa fie inlocuita.

Algoritmul LFU- (cel mai putin frecvent utilizata Least Frequently Used).

Acest algoritm de *inlocuire pagina* alege pagina "victima" dupa frecventa utilizarii paginilor.

Pentru aceasta este nevoie de un mecanism harware ("numarator") care sa numere cate referiri s-au facut in fiecare pagina. Pagina al carui "numarator" are valoarea cea mai mica va fi inlocuita. Aceast algoritm poate da rezultate nestisfacatoare atunci cand o pagina este utilizata intens la inceput iar apoi ea nu va mai fi utilizata deloc. Utilizarea intensa de la inceput va face ca valoarea "numaratorului" acestei pagini sa fie suficient de mare pentru ca ea sa nu fie inlocuita foarte mult timp cu toate ca ea nu mai este necesar sa fie in memorie. O inbunatatire a acestui algoritm se poate face prin introducerea unei operatii de deplasare spre stanga a acestui "numarator" cu un bit la intervale regulate de timp.

Algoritmul MFU (cel mai frecvent utilizata Most Frequently Used).

Un alt algoritm de *inlocuire pagina* este cel bazat pe compararea frecventei de utilizare a paginilor este MFU. Spre deosebire de algoritmul LFU pagina "victima" este pagina al carui "numarator" al referirilor este del mai mare.

Atat algoritmul LFU cat si MFU necesita mecanisme hardware suplimentare iar rezultatele sunt destul de indepartate fata de algoritmul OPT.

5.8.5.4. Algoritmi Ad Hoc

Deseori se folosesc si alte proceduri in completarea unor algoritmi de *inlocuire pagina* specifici.

1) Asa cum am vazut, de obicei, SO pastreaza un numar de *cadre* de rezerva care nu se aloca decat in situatii speciale. Atunci cand are loc un "page-fault", este aleasa pagina "victima" conform algoritmului de *inlocuire pagina*. Totusi, pagina dorita se incarca intr-un *cadru* de "rezerva" nu peste *pagina* "victima". In acest fel procesul se restarteaza imediat nemai fiind necesar sa se astepte copierea *paginii* "victima" pe disc. Ulterior pagina "victima" se va copia de pe disc (swap-out), daca este posibil in paralel cu alte activitati si *cadrul* este adaugat in lista de cadre libere.
2) In extensia procedurii anterioare este o alta procedura care va actualiza paginile de pe disc care au suferit modificari. Sistemul administreaza o lista a paginilor modificate. Atunci cand perifericul de "swap" (evacuare-reincarcare) este neutilizat, paginile modificate vor fi copiate pe disc apoi vor fi scoase din lista "paginilor modificate". Aceasta schema face ca sa creasca probabilitatea ca o pagina care trebuie inlocuita, deci copiata pe disc, sa nu mai fie necesar transferul ei pe disc deoarece imaginea sa este identica cu cea de pe disc.
3) O alta procedura adaugata la algoritmul de *inlocuire pagina* care poate conduce la cresterea performantei sistemului este cea prin care se memoreaza in fiecare *cadru* liber ce *pagina* s-a aflat in acel *cadru*. Daca continutul *cadrului* nu a fost alterat ultima *pagina* se mai afla acolo si poate fi reutilizata nemaifiind necesara incarcarea ei pe disc la urmatoare ei utilizare.
   Atunci cand se produce un "page-fault" se verifica mai intai daca nu cumva *pagina* ce trebuie incarcata se afla intr-unul din *cadrele* din "lista cadrelor libere" dintr-o pseudo-evacuare anterioara. Daca nu, pagina se incarca de pe disc. Si aceasta procedura este in completarea procedurii ajutatoare de la punctul 1, ea nefiind posibil de implementat fara aceasta.

5.8.6. Alocarea cadrelor.

SO trebuie sa aloce fiecarui proces un spatiu de memorie.
Cel mai simplu caz este cel in care activ la un moment dat este un singur proces utilizator. In acest caz toata lista de cadre libere este alocata pe masura ce este necesar procesului. Daca lista de cadre libere este epuizata va fi utilizat algoritmul de *inlocuire pagina*.
In plus se poate elibera o zona de memorie pe care o foloseste SO pentru "buffere" (zone tampon) care poate fi utilizata de procesele utilizator daca in acel moment nu sunt utilizate de SO.
Lucrurile se complica in momentul in care sunt mai multe procese active (in sisteme cu multiprogramare) fiind necesara incarcarea in memorie a mai multor procese.
Strategia de alocare a *cadrelor* pentru mai multe procese active simultan, este influentata si de diverse constrangeri legate de arhitectura sistemului calculator.
Intodeauna exista un numar minim de cadre care poate fi alocat fiecarui proces. Pe langa faptul ca performantele scad in cazul alocarii unui numar mic de cadre unui proces, exista si o limita inferioara a numarului de cadre dictata de arhitectura sistemului calculator.
Acest numar minim de cadre este diferit de la sisitem la siste,m si depinde de arhitectura setului de instructiuni.
Restrictia provine de la conditia obligatorie de a avea suficiente pagini simultan in memorie cate pot fi referite de o singura instructiune. Acest lucru este legat de cate nivele de adresare indirecta sunt permise in SC.
Daca numarul minim de *cadre* ce trebuie alocat unui proces trebuie sa fie cel putin egal cu numarul maxim de adresari indirecte acceptat de SC. Daca aceasta conditie nu este indeplinita atunci executia unei singure instructiuni intra intr-o "bucla" de *inlocuire de pagini* fara sfarsit deoarece la fiecare "page-fault" se restarteaza instructiunea.
De exemplu daca este permis un numar de 16 nivele de adresare indirecta atunci numarul minim de cadre alocate unui proces este 16.
Deci numarul minim de cardre alocat fiecarui proces este definit de arhitectura SC in timp ce numarul maxim este definit de cantitatea de memorie fizica disponibila.

5.8.6.1. Algoritmi de alocare.

Algoritmii de alocare trebuie sa rezolve urmatoarea problema: cum se distribuie "m" cadre de memorie la un numar de "n" procese ?
Cea mai simpla cale este de a atribui fiecarui din cele "n" procese o cantitate egala de cadre = "m/n".

daca pi ” procesele active
unde i=1,2 ... n
si "n" este numarul de procese;

m ” numar de cadre de memorie fizica disponibile

ai ” numar de cadre de alocat procesului "i"

ai = m/n ;

In cazul in care raportul "m/n" produce rest, acest rest reprezinta numarul de cadre de rezerva. Acest mod de alocare se numeste " alocare egala ".
Acest algoritm sufera de faptul ca numarul de cadre alocate unui proces nu tine cont de numarul de cadre necesare procesului.
De exemplu daca SC are disponibile un numar de 93 de cadre (m=93) si presupunem ca numarul proceselor active este 5 (n=5) atunci fiecarui proces i se va aloca un numar de:

ai = 93/4=18 rest 3 ,

Deci fiecarui proces i se vor aloca 18 *cadre* ramanand un numarde 3 *cadre* de rezerva.
Se remarca usor ca, de exemplu, daca necesare fiecarui proces le sunt, sa zicem:
s1 = 10, s2 = 42 , s3 = 16, s4 = 24, s5 = 28
atunci:
procesele "p1" si "p3" au alocat mai mult decat le este necesar pe cand proceselor "p2", "p4" si "p5" mai putin.
Pentru a rezolva o asemenea situatie putem utiliza algoritmul de "alocare proportionala" care aloca cadrele proportional cu numarul de cadre necesare fiecarui proces. Daca

S = xsi reprezinta totalul cadrelor necesare
unde "si" necesarul de cadre pentru procesul "pi";

Daca "m" reprezinta numarul de cadre disponibile in memoria fizica, atunci numarul de cadre alocat fiecarui proces este:

ai = si/S * m

Aplicand aceast algoritm pentru exemplul considerat anterior, totalul necesar de cadre este:

S=10+32+16+24+28=110

iar numarul de cadre alocat fiecarui proces (rotunjit la valoare intreaga in minus) va fi:

a1 = 10/110 * 93 = 8

a2 = 32/110 * 93 = 27

a3 = 16/110 * 93 = 13

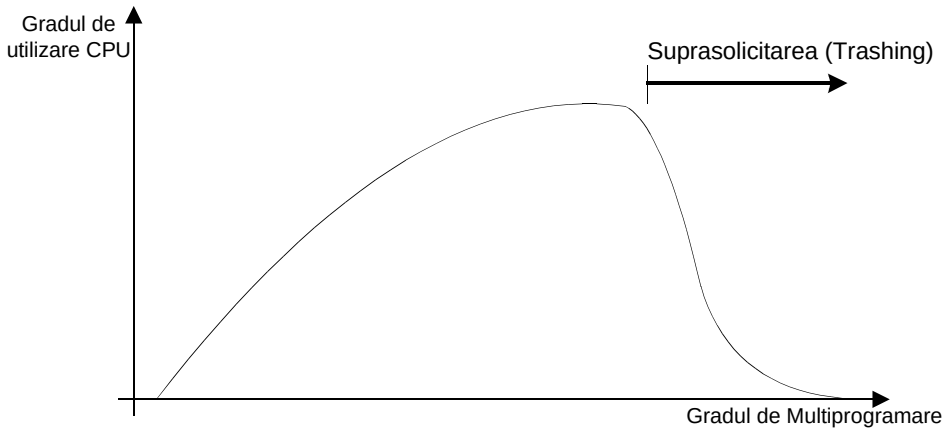
a4 = 24/110 * 93 = 20

a5 = 28/110 * 93 = 23

Aceasta alocare a cadrelor este evident mai buna decat cea in cazul alocarii egale.
Cele 5 procese primesc in mod proportional *cadre*, iar doua cadre raman in "lista de cadre de rezerva".
In ambele metode de alocare (proportionala si egala), numarul de cadre alocat unui proces depinde de gradul de multiprogramare. In cazul in care gradul de multiprogramare creste fiecare proces cedeaza cateva cadre pentru a furniza memoria necesara noilor procese. In schimb daca gradul de multiprogramare scade cadrele ramase disponibile se distribuie intre procesele ramase.
Remarcam insa ca acesti algoritmi de alocare trateaza in mod egal procesele cu prioritate mare sau mica. Prin definitie proceselor cu prioritate mai mare trebuie sa le asiguram memoria pentru a creste viteza de executie. Din aceste motive au aparut si algoritmi care tin cont si de prioritatea proceselor la alocarea cadrelor.
Astfel acesti algoritmi realizeaza o alocare proportionala cu prioritatea procesului sau o combinatie intre prioritatea procesului si dimensiunea procesului (numarul de pagini).
Acest algoritm avantajeaza procesele cu prioritate mare in detrimentul celor cu prioritate mica.

5.8.7.Suprasolicitarea (Trashing)

Daca numarul de cadre alocat proceselor cu prioritate mica coboara sub numarul minim de cadre datorat arhitecturii SC executia procesului trebuie suspendata.
Paginile aflate deja in memorie, ale procesului care se suspenda vor fi evacuate (swap-out), eliberand cadrele ocupate de acestea. Aceasta prevedere introduce un nivel suplimentar de evacuare-reincarcare (swap out swap in) pentru planificator.
Desi tehnic este posibil sa fie redus numarul de cadre alocate unui proces la minim, exista un numar limita de cadre sub care numarul de accese "page-fault" creste foarte mult. Acest numar limita este de obicei egal cu numarul paginilor "in uz" la un moment dat. Daca procesul nu are alocat cel putin acest numar de cadre, foarte frecvent apar accese "page-fault". Aceasta activitate foarte intensa de *inlocuire de pagina* este numita suprasolicitare (TRASHING).
Suprasolicitarea determina o scadere puternica a performantelor sistemului. Timpul consumat de mecanismul de *inlocuire pagina* devine mai mare daecat timpul de executie efectiva aproceselor.
Dependenta gradului de utilizare a CPU de gradul de multiprogramare este reprezenrtata in diagrama:



Suprasolicitarea (Trashing)

Se remarca ca odata cu cresterea gradului de multiprogramare creste si gradul de utilizare al CPU, dar acest lucru se intampla numai pana la un anumit grad de multiprogramare. Din acest punct apare *Suprasolicitarea* (TRASHING) datorita faptului ca numarul cadrelor se reduce pentru fiecare proces sub limita numarului de cadre corespunzatoare numarului de pagini in uz , cu reducerea drastica a performantei. In acest caz SO trebuie sa reduca numarul proceselor active din sistem (gradul de multiprogramare).

5.8.9.Cererea de segmentare

Cu toate ca *cererea de paginare* este considerata in general ca fiind cea mai eficienta schema de *memorie virtuala*, ea implica existenta unor dispozitive hardware suplimentare.
Atunci cand aceste dispozitive hardware lipsesc se aleg metode de implementare a *memoriei virtuale* mai putin eficiente. Acesta este cazul *cererii de segmentare* care este un mecanism de alocare a memoriei destul de asemanator cu *cerere de paginare* care insa utilizeaza ca unitate de manipulare *segmentul* de program. Procesoarele Intel nu ofera mecanisme de paginare ci mecanisme de segmentare.

Sistemele de operare OS/2 (IBM) care ruleaza pe aceste procesoare utilizeaza mecanismele hardware de segmentare pentru a implementa *cererea de segmentare* ca o aproximare a *cererii de paginare*.
Sistemul de operare OS/2 aloca memoria in *segmente* nu in *pagini*. Pentru administrarea memoriei SO intretine niste *tabele de descriere segment (Tabela de Segmente)* care contin informatii legate de fiecare *segment* in parte: dimensiune, protectie, localizare, etc.
Procesul in executie nu are nevoie de toate segmentele sale in memorie. In Tabela de Segmente exista bitul "v-i" (valid-invalid) care indica daca segmentul este sau nu in memorie.
Atunci cand in timpul executiei se face referire la o anumita adresa mecanismul hardware de adresare verifica bitul "v- i" este valoarea lui este "v" atunci segmentul se afla in memorie si accesul se executa imediat. Daca bitul este "i" atunci se declanseaza o intrerupere (TRAP) si SO incarca segmentul respectiv in memorie, dupa care se restarteaza instructiunea.
Aceasta intrerupere se numeste "eroare adresare segment" ("segment fault").
Daca in momentul in care SO trebuie sa incarce in memorie segmentul referit, nu este spatiu in memoria fizica si declanseaza un mecanism de *inlocuire segment*. Pentru a determina care segment trebuie inlocuit (segment "victima") SO utilizeaza un alt bit din TS numit "bit de accesare" prin care se alege unul din segmentele cele mai vechi pentru a fi evacuat.
Deci mecanismul de *cerere de segmentare* este similar cu cel de *cerere de paginare* numai ca se lucreaza la nivel de segment. Deoarece dimensiunea segmentelor este variabila alocarea memoriei pentru segmente este complicata.

CAP.VI ADMINISTRAREA MEMORIEI SECUNDARE

Scopul acestui capitol este descrierea principalelor tipuri de memorii secundare ale SC si a metodelor utilizate de SO pentru administrarea acestora.

In sistemele calculator numim memorii secundare sau dispozitive de stocare secundare dispozitivele periferice cu suport magnetic (banda magnetica, discul flexibil (flopy), discul dur (rigid) sau dispozitive periferice optice (Compact discul). Dintre acestea exceptand banda magnetica, toate celelalte periferice (sau suporti de informatie) au denumirea generica de discuri.
Le numim "secundare" ca o extensie la memoria primara care o mai numim si memorie principala.

6.1.Introducere

Principalul scop al sistemelor calculator este executia programelor. Aceste program impreuna cu datele pe care le prelucreaza sau le produc trebuie sa se afle in *memoria principala* pe durata executiei programului. Ideal ar fi ca sa putem avea programele si datele lor permanent in memoria principala. Acest lucru nu este posibil din doua motive:

- memoria principala este prea mica pentru a stoca toate programele si datele necesare care se prelucreaza intr-un sistem calculator;
- memoria principala este volatila adica informatia stocata in ea se pierde odata cu scoaterea de sub tensiune a SC.

Principalul scop al *memoriei secundare* este acela de a pastra un numar foarte mare de informatii (programe, date, etc) pe o durata nedeterminata de timp (permanent).
Perifericul *banda magnetica* este utilizat din ce in ce mai putin. *Banda magnetica* este un mediu de stocare permanent dar are dezavantajul ca este lenta si nu permite decat accesul secvential la informatie. Banda magnetica se mai foloseste acum pentru:

- salvare de date pentru pastrarea unor copii de rezerva ale datelor (suport de "backup");
- stocarea datelor utilizate foarte rar;
- pentru transferul de informatie de volum mare intre SC si altul.

Exista doua diferente majore intre exploatarea *benzii magnetice* si *a discului:*

- durata unei operatii de citire sau scriere in cazul *benzii magnetice* depinde de pozitia (distanta fata de capatul benzii) unde se citeste/scrie informatia, in timp ce la *dsic* aceasta durata este identica indiferent de locul pe suport unde se afla (la citire) respectiv unde se scrie informatia;
- actualizarea (modificarea) informatiei este posibila la *banda magnetica* numai prin rescrierea continutului intregii benzi, in timp ce la *disc* actualizarea se poate face direct inlocuind numai informatia ce se modifica.

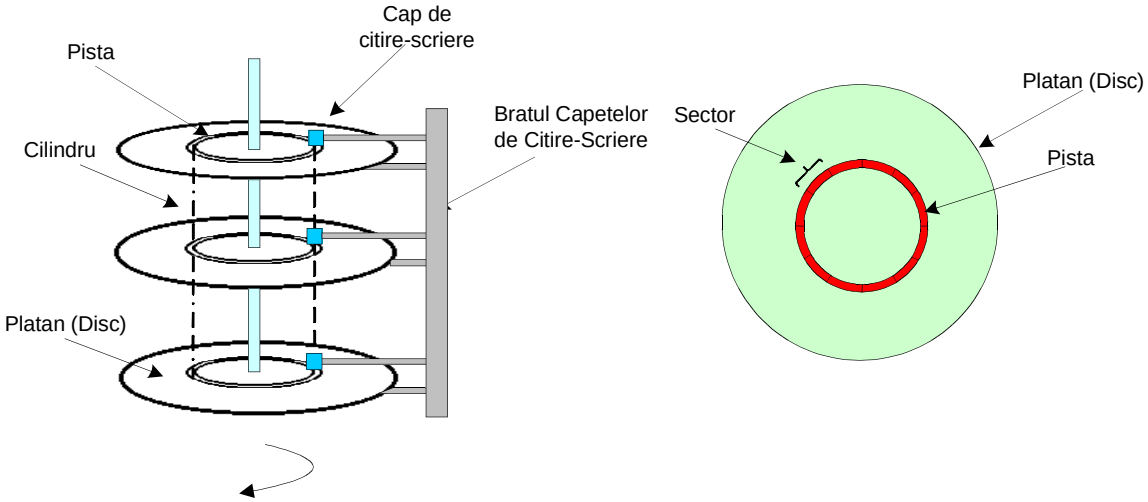
La ora actuala *discul* (magnetic sau optic) are o raspandire foarte mare. Este folosit aprope in exclusivitate ca dispozitiv secundar de stocare. Chiar daca *discul* poate fi *disc magnetic* sau *disc optic* d.p.d.v functional (logic) ele sunt de cele mai multe ori echicvalente. De aceea de cele mai multe ori le vom trata identic fara sa mai specificam faptul ca este magnetic sau optic.
Diferenta majora a memoriei secundare-disc fata de memoria principala este timpul de acces la informatie care este mult mai mic la *memoria principala.*

6.2.Structura discului

Fizic, atat dispozitivul periferic propriuzis cat si suportul de informatie disc sunt deosebite la *discurile magnetice* fata de *discurile optice*. Vom discuta in continuare principial functionarea discului magnetic.
Perifericul *disc magnetic* foloseste pentru "memorarea" informatiei proprietatile magnetice ale suportului iar *discurile optice* folosesc modificarile fizice in structura materialului (folia de aluminiu).

6.2.1.Structura fizica a discului magnetic

Atunci cand discul este in functiune un motor roteste suprafetele circulare numite *platane* sau *placi* ale discului. Deasupra fiecarui platan "pluteste" cate un *cap de citire-scriere*. Suprafata discului este impartita logic in *piste* .
Informatia se memoreaza, prin magnetizarea materialului aflat pe *pista* care se afla sub capul de citire-scriere. In acest fel putem avea sute sau mii de *piste* pe un disc ele fiind succesiunea cercurilor concentrice (imaginare) descrise de toate pozitiile succesive ale capetelor de citire. Capacitatea discului depinde de densitatea de inregistrare care reprezinta numarul de biti inregistrati pe unitatea de lungime (o pista).
Toate *pistele* de acelasi diametru formeaza un cilindru. Fiecare pista este divizata in *sectoare* de dimensiune fixa (de ex.emplu 512 Ko). Pe fiecare suprafata a discurilor se afla un cap de scriere-citire. Deci intodeauna *suprafata* identifica si *capul de citire-scriere* pe fiecare "placa" a discului existand doua suprafete deci doua capete de citire-scriere. De obicei un disc magnetic are mai multe placi (platane).



In acest fel discul poate fi vazut ca un tablou tridimensional de sectoare. Unitatea de adresare la o informatie aflata pe disc este sectorul. O adresa particulara de sector este data de:

pista (cilindru);
suprafata; (sau numar cap)
sector.

Adica tripleta (NumarSector, NumarSupr., NumarCil) reprezinta adresa fizica a unui sector.
Transferul de informatie intre memoria principala si disc se face in grupe ("unitati de transfer") de unul sau mai multe sectoare numite si blocuri.

Sistemul de operare trateaza orice adresa de sector ca un vector de adrese de blocuri de disc. Un bloc putand avea unul sau mai multe sectoare, adresa unui bloc este adresa primului sector al blocului.
Putem converti o adresa fizica data sub forma (NumarSector, NumarSupr., NumarCil.) intr-o valoare care reprezinta adresa de bloc:

Adresa.Bloc = NumarSector+ Sect /Pista * (NumarSupr + NumarCil * Piste/Cil);

unde Sect /Pista numarul de sectoare pe pista;
Piste/Cil numarul de piste (suprafete) pe cilindru;
NumarSector numarul sectorului adresat in cadrul pistei;
NumarCil numarul cilindrului unde se afla sectorul adresat;
NumarSupr suprafata (sau numarul capului de citire scriere) unde se afla sectorul adresat.

Aceasta adresa reprezinta numarul de ordine al sectorului fata de inceputul discului adica fata de primul sector al primei suprafete (primul cap) de pe primul cilindru.
Discul magnetic este caracterizat de faptul ca:

permite accesul direct atat in scriere cat si in citire la orice bloc de pe disc indiferent de pozitia sa in cadrul discului;
Indiferent daca blocul este scris sau citit si indifernt de pozitia blocului pe disc timpul de acces este aproximativ acelasi;

6.2.2.Directorul discului

Orice disc contine o structura de informatie numita "directorul discului" (sau ROOT sau Radacina). In acest director se gaseste o lista cu numele fisierelor care se afla pe disc cu alte cuvinte continutul discului. In aceasta lista cu continutul discului se afla pe langa numele fisierelor si informatii legate de aceste fisiere: locul unde se afla, lungimea, tipul fisierulu, proprietarul, data creerii fisierului, data ultimei utilizari a fisierului, protectie, etc.
Directorul discului este memorat chiar pe disc, de obicei la o adresa fixa, de exemplu 00001. Adresa 00000 este in general incarcatorul SO.

6.3.Administrarea spatiului liber pe disc

Deoarece spatiul disponibil al oricarui disc este limitat, acest spatiu este reutilizat in sensul ca fisierele care sunt sterse elibereaza spatiul ocupat. Pentru a tine evidenta spatiului liber pe un disc SO foloseste o lista numita "lista spatiului liber". In aceasta lista se gasesc toate blocurile libere adica lista blocurilor care nu sunt alocate fisierelor. La crearea unui fisier SO cauta in "lista spatiului liber" un numar de blocuri necesare fisierului si le aloca acestuia. Blocurile alocate unui fisier sunt apoi sterse din "lista spatiului liber". Atunci cand un fisier este sters, blocurile sale vor fi adaugate in "lista spatiului liber". Lista spatiului liber poate fi implementata in diverse moduri.

1. Vector de biti (harta de biti)

O metoda de implementare a listei spatiului liber este printr-un sir de biti care formeaza o structura de date numita *vector de biti* sau *harta de biti*. In aceasta structura fiecare bloc este reprezentat printr-un bit. Daca blocul este liber bitul corespunzator este "0", daca este alocat este "1".
De exemplu pentru un disc care are libere blocurile 2,3,4,5,8,10,13,... *harta de biti* arata astfel:

11000011010110.....

Avantajul acestei implementari este ca este usor de prelucrat. De obicei exista instructiuni specializate care cerceteaza siruri de biti si furnizeaza pozitia primului bit "1" . In acest fel devine simplu si eficient de a gasi "n" blocuri libere consecutive. Pentru a avea o manipulare rapida a *hartii*

de biti devine necesara pastrarea lui in memoria principala si numai periodic copiata pe disc. Dezavantajul metodei este ca pentru discuri de capacitate mare sirul de biti este foarte mare si devine dificila pastrarea si manipularea lui.

2. Liste inlantuite

O alta implementare a listei de blocuri libere este prin *liste inlantuite* implementate chiar in blocurile discului. SO pastreaza adresa primului bloc liber iar acesta la randul sau contine adresa urmatorului bloc liber, etc. In exemplul nostru SO pastreaza in poinetrul de blocuri libere adresa blocului 2. In blocul 2 se va gasi adresa urmatorului bloc liber adica 3 s.a.m.d. Aceasta schema nu este rapida fiind necesara executia mai multor operatii de I/O pentru parcurgerea acesei liste.

3. Gruparea

O alta metoda este de a se construi o lista de adrese ale primelor n blocuri libere in primul bloc liber. In ultimul bloc liber al primei liste se afla adresa altor n blocuri libere s.a.m.d.
Avantajul acestei metode este posibilitatea gasirii rapide a unui numar mare (n) de blocuri libere foarte rapid.

4. Numararea

Acesta metoda de administrare a spatiului liber pe disc se bazeaza pe informatia ca de cele mai multe ori blocurile libere sunt grupate. In loc sa se pastreze o lista cu "n" adrese de blocuri libere se construieste o lista in care fiecare intrare contine adresa primului bloc liber si numarul blocurilor libere contigue care urmeaza blocul respectiv.
In exemplul anterior adresa primului bloc liber va fi 2 urmata de valoarea 3 care reprezinta numarul de blocuri libere care urmeaza dupa blocul liber 2, s.a.m.d.
Aceasta metoda este foarte buna atunci cand blocurile libere sunt grupate.

6.4.Metode de alocare disc

Informatiile memorate pe disc trebuie sa fie organizate sub forma de ***fisiere.*** Administrarea fisierelor si a spatiului pe disc este realizata de SO. Atunci cand ne referim la alocarea pe disc ne vom referi la modul in care se aloca unui fisier spatiul necesar memeorarii sale pe disc.
Accesul direct caracterizeaza memoria secundara de tip disc. Acest lucru creaza flexibilitate in implementarea metodelor de alocare. In cele mai multe cazuri pe acelasi disc pot fi memorate mai multe fisiere. Principala problema care apare este cum se aloca spatiul la aceste fisiere in asa fel incat acest spatiu sa fie utilizat eficient si accesul la fisier sa se faca cat mai rapid.
Exista trei metode mai importante pentru alocarea spatiului disc:

- alocare contigua;
- alocare legata (inlantuita);
- alocare indexata.

In discutiile ulterioare un fisier va fi considerat ca o secventa de blocuri. Unitatea de transfer intre memoria principala si periferic este blocul, functiile de I/O operand cu blocuri. Conversia din inregistrari logice cu care opereaza aplicatiile, in blocuri fizice care sunt manipulate de operatiile de I/O este relativ simpla si se realizeaza de catre sistemul de operare. Unitatea de alocare de spatiu pentru un fisier este ***blocul*** sau ***clusterul***. Dimensiunea *clusterului* este foarte importanta si este caracteristica fiecarui sistem de fisiere. O dimensiune mare a acestuia conduce la un timp de citire sau scriere mai scurt al uni fisier insa va produce o crestere a spatiului disc neutilizat datorita fragmentarii interne.
Fiecare SO utilizeaza de obicei o singura metoda de alocare.
Utilizarea blocurilor care de obicei sunt formate din mai multe sectoare ca unitate de transfer si alocare creaza fragmentare interna, ultimul bloc alocat unui fisier putand fi incomplet ocupat.

6.4.1.Alocarea contigua

Acest tip de alocare este caracterizat de faptul ca fiecare fisier ocupa un set de adrese succesive pe disc adica zone contigue de disc. Adresele disc ale blocurilor unui fisier definesc o ordine liniara de adrese disc. Aceasta ordonare a adreselor disc face ca accesul la blocul "b+1" dupa accesul la blocul "b" sa nu determine deplasarea capetelor (in mod normal). Daca la blocurile unui fisier se face un acces secvential nu va avea loc o deplasarea a capetelor decat foarte rar atunci cand se intampla trecerea de la ultimul bloc al cilindrului la blocul urmator care este primul bloc pe urmatorul cilindru. In aceast caz aceasta deplasare este numai cu o pista. Deci se consta ca in cazul alocarii contigue acesul secvential se face foarte rapid.
Alocarea "contigua" a unui fisier este definita de doua elemente:

adresa disc a primului bloc alocat fisierului;
lungimea (nr. de blocuri alocate);

Daca un fisier are lungimea de "n" blocuri si adresa de inceput "b", atunci fisierul va ocupa blocurile disc:

b, b+1, ..., b+(n-1).

Accesarea unui fisier in aceasta situatie se poate face simplu atat secvential cat si direct. Accesul secvential presupune citirea intotdeauna a urmatorului bloc in secventa.
Accesul direct al unui bloc de rang "i" al unui fisier se face imediat stiind adresa de inceput "b" a fisierului. Sistemul calculeaza imediat adresa "b+i" unde se face accesul.
Problema care apare in cazul " alocarii contigue " este, unde se face alocarea unui numar de "n" blocuri cand avem o lista de spatii libere (nise sau hole-uri).
Cele mai cunoscute strategii in aceasta situatie sunt aceleasi ca si in cazul alocarii dinamice a memoriei principale pentru procese:

best -fit : cea mai buna umplere a unui spatiu (nise);
first-fit : primul loc liber in care incape fisierul;
worst-fit :cea care lasa cel mai mare spatiu liber in nisa in care se face alocarea.

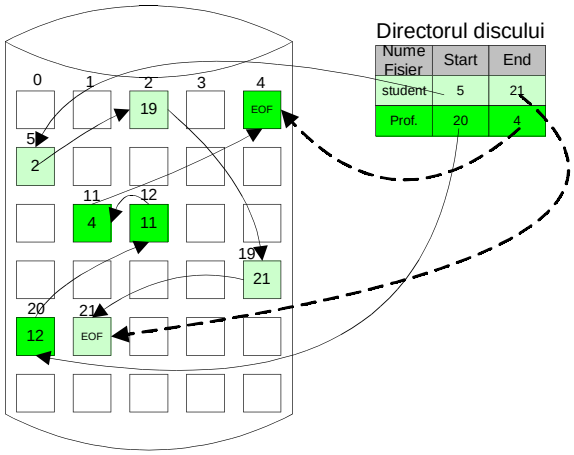
In urma simularilor s-a constatat ca cele mai bune strategii sunt *first-fit* si *best-fit* din punct de vedere al utilizarii spatiului, nefiind o deosebire neta intre cele doua. Dar din punct de vedere al timpului, *first-fit* se dovedeste mai rapida. Apar doua dezavantaje ale *alocarii contigue*:

1. *Alocarea contigua* produce *fragmentarea externa.* care se poate elimina prin programe de compactare care insa dureaza destul de mult ele ne putand fi executate din acest motiv foarte des;
2. Nu se poate determina "apriorii" spatiul necesar unui fisier. Aplicatiile in general atunci cand creaza fisiere, isi extind dinamic numarul de inregistrari fara sa se cunoasca de la inceput cate inregistrari logice va avea fisierul deci cate blocuri fizice vor fi necesare. Aceast lucru creaza mari probleme deoarece initial se aloca un anumit spatiu unui fisier care ulterior nu mai este suficient si este necesar sa fie extins.

Pentru rezolvarea acestei probleme exista doua solutii:
a) Se cere utilizatorului o estimare a numarului de inregistrari total pe care-l va avea un fisier. Aceasta metoda creaza fragmentare interna pentru ca de obicei utilizatorul estimeaza acoperitor numarul de inregistrari al unui fisier.
b) Se aloca un spatiu initial acoperitor care ocupa o intreaga nise. In momentul cand aceasta nise devine insuficiente se recopiaza fisierul in alta nise mai mare si se continua extinderea fisierului. Aceasta metoda insa este mare consumatoare de timp datorita faptului ca este neceesara uneori recopierea intregului fisier.

6.4.2.Alocarea inlantuita

In cazul acestei metode unui fisier i se aloca blocurile necesare acolo unde se gasesc ele disponibile pe disc. Blocurile alocate unui fisier sunt liste inlantuite de blocuri disc. In acest fel blocurile unui fisier pot fi dispersate oriunde pe disc. Directorul fisierului va contine un pointer catre primul bloc alocat fisierului, acesta la randul lui punctand urmatorul bloc alocat fisierului s.a.m.d. Aceasta metoda nu creaza *fragmentare externa.*



Alocarea inlantuita a spatiului disc

Avantajele *alocarii inlntuite*:

crearea fisierelor este simpla;
nu creaza fragmentare externa;
nu este necesara declararea la crearea fisierului a dimensiunii fisierului;
nu este necesara compactarea periodica a discului fisierului (este o consecinta a lipsei fragmentarii externe).

Metoda aceasta creaza insa si cateva dezavantaje:

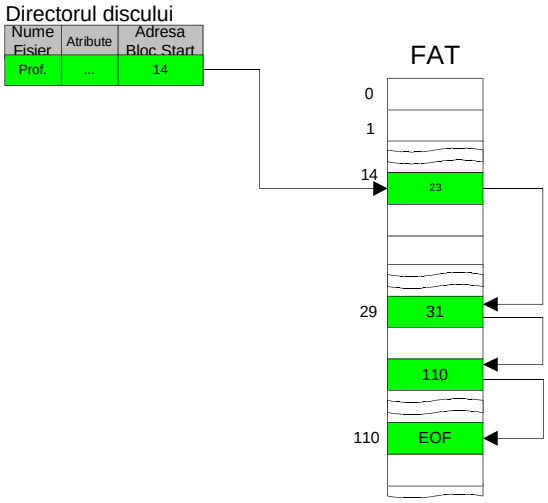
nu poate fi accesat fisierul decat secvential;
necesita un spatiu suplimentar pentru pointeri;
fiabilitate redusa. In cazul unui defect care face imposibila citirea unui bloc se pierde "inlantuirea" ceea ce face ca urmatoarele blocuri in secventa sa nu se mai poate adresa.

O solutie partiala la aceasta problema este dublarea listei (redundanta) sau memorarea in fiecare bloc al fisierului a unor informatii suplimentare: numele fisierului si adresa relativa a blocului. Aceste metode introduc o marire a spatiului ocupat de un fisier si cresterea duratei accesului.

O varianta perfectionata a *alocarii inlantuite* este utilizarea unei structuri de date asociate. Aceasta structura poarta numele de *Tabela de alocare Fisiere* (**FAT** File allocation Table) si este memorat pe disc in zona de informatii a SO.
Aceasta tabela indexata de numarul blocului contine cate o intrare pentru fiecare bloc de pe disc. Fiecare intrare (corespunzatoare la cate un bloc) contine:

valoarea "0" daca blocul nu este alocat nici unui fisier;
numarul blocului urmator al fisierului (pointer);
o valoare speciala marcand "end-of-file (**EOF**) sfarsitul fisierului".

In directorul discului alaturi de numele fisierului se va gasi si numarul primului bloc al fisierului.

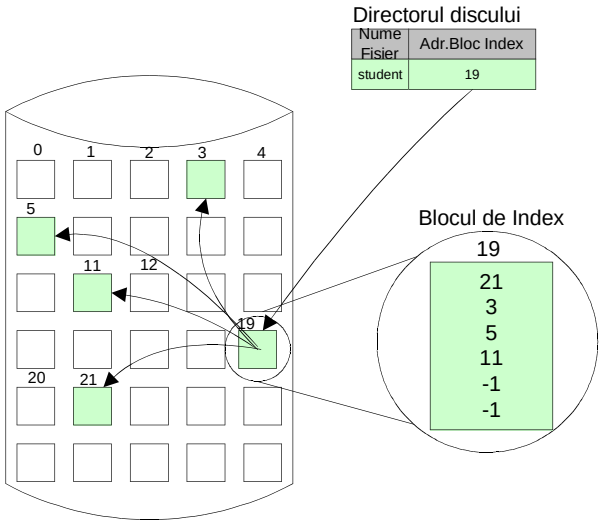


Alocarea utilizand tabela FAT

Adaugarea unui nou bloc la un fisier (extinderea fisierului) se face foarte simplu cautand prima intrare cu valoarea zero din FAT si apoi se inlocuieste intrarea in FAT asociata acestui fisier care are valoarea EOF cu adresa acestui bloc.
Aceasta metoda este utilizata de SO, MS-DOS, OS/2, el permitand accesul secvential la fisiere.

6.4.3.Alocarea indexata

Aceasta metoda elimina fragmentarea externa si necesitatea declararii de catre programator la crearea unui fisier a dimensiunii estimate a acestuia. In plus creaza posibilitatea accesului direct la blocurile fisierului.
Fiecare fisier are asociat cate un *bloc index* care este o tabela de blocuri logice. Fiecare intrare in acest *bloc index* contine adresa fizica a unui bloc disc.
Blocul index este memorat pe disc in primul bloc alocat fisierului. Citirea blocului logic "i" al fisierului presupune accesarea intrari "i" a *blocului index* . In aceasta intrare se va gasi adresa blocului fizic unde se afla blocul logic "i". Aceasta schema este asemanatoare cu schema de paginare utilizata la alocarea memoriei principale.



Alocarea Indexata a spatiului disc

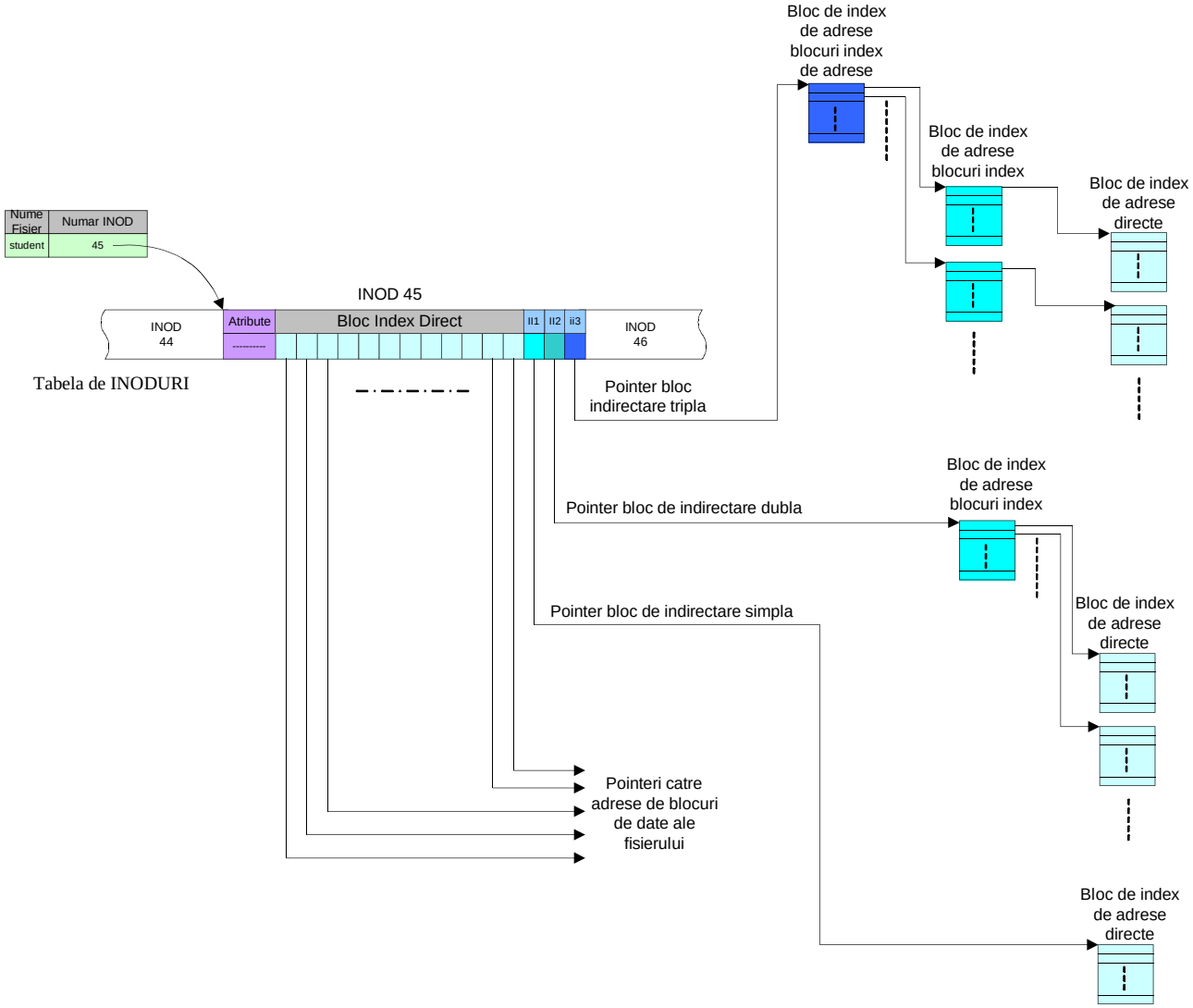
Atunci cand fisierul este creat toti pointerii in *blocul index* sunt "-1" (NIL). Atunci cand se scrie prima oara blocul "i" se cauta un bloc liber in lista de blocuri libere a discului si adresa sa este inscrisa in intrarea "i" a *blocului index.* Accesul secvential la blocurile fissierului se face parcurgand secvential lista aflata in *blocul de index.*
Problema care se pune in cazul alocarii indexate este cat alegem de mare *blocul de index.* Tendinta este de a avea un bloc de index cat mai mic dar asta inseamna si vor putea fi pastrati mai putini pointeri deci insuficienti pentru fisiere de dimensiuni mai mari. Pentru a solutiona aceasta problema se pot utiliza mecanisme suplimentare.
Blocuri index inlantuite In mod normal unui fisier un singur *bloc de index* care atunci cand numarul de pointeri catre blocurile disc ale fisierului devin insuficienti (fisierul devine mai mare) se inlantuie cu un alt *bloc de index.* In acest caz *blocul de index* va contine pe ultima sa pozitie un pointer catre urmatorul *bloc de index.*
Index pe mai multe nivele
Aceasta schema foloseste mai multe nivele de *bloc index*.In cazul fisierelor de mari dimensiuni la care nu sunt suficiente intrarile care pot fi memorate in *blocul de index* atunci in primul bloc de index nu se vor gasi indexul blocurilor de date ale fisierului ci pointerii catre alte blocuri care contin fie pointeri la blocurile de date ale fisierului fie pointeri la blocuri care sunt la randul lor blocuri de index, s.a.m.d. Toretic pot fi 3 sau 4 nivele de blocuri de index dar de obicei doua nivele de blocuri index sunt suficiente pentru fisiere e mari dimensiuni.
De exemplu daca o schema de blocuri index pe doua nivele si putem avea sa zicem 256 pointeri intrun bloc de index atunci avem la dispozitie un numar de 256 * 256 pointeri adica un numr de 65.536 blocuri de date pentru fisier. Daca tinem cont ca un bloc de date poate contine mai multe inregistrari logice atunci putem spune ca acesasta schema este acoperitoare pentru majoritatea fisierelor.
Alocarea indexata suporta accesul direct si secvential si nu produce fragmentare externa.
Alocarea indexata este asemanatoare cu " alocarea inlantuita " cu FAT.
Alocarea indexata utilizeaza insa spatiu suplimentar pentru memorarea " blocului index ".

Schema de alocare combinata.

Sistemul de operare BSD UNIX foloseste o schema combinata de inlantuire si *blocuri index* pe mai multe nivele.

SO BSD UNIX pastreaza in directorul perifericului un prim nivel de 15 pointeri. Primii 12 pointeri din acesti pointeri contin adrese de blocuri de date (blocuri directe) iar urmatorii 3 pointeri contin respectiv:

adresa unui "bloc index" - indirect bloc
adresa unui bloc de adrese de blocuri index - indirect simplu bloc
adresa unui bloc de adrese de adrese de blocuri index - dublu indirect bloc



Schema de alocare combinata (BSD UNIX)

Aceasta schema este eficienta atat in cazul fisierelor de dimensiuni mici (cu mai putin de 12 blocuri) cat si in cazul fsierelor de diensiuni mari adaptandu-se dinamic prin utilizarea adreselor indirecte de blocuri index.

6.5.Planificarea accesului la disc

Timpul de executie al programelor depinde in mare masura de viteza te transfer a discului.
In afara de faptul ca insasi procesele utilizatorului solicita transferuri de date cu discul si SO isi bazeaza in mare parte mecanismele de adimnistrare ale SC pe utilizarea discului. Din aceste motive este important sa se creasca performantele discului.
Timpul de acces la disc este compus din trei componente:

timpul de deplasare a capetelor de citire-scriere pe cilindrul care contine adresa unde se afla informatia. Acest timp se mai numeste *timp de cauatare* (seek).

timpul de asteptare pana cand blocul ce se doreste a fi citit ajunge, prin rotirea discului, in dreptul capetelor de citire-scriere. Aceasta timp se numeste *timp de latenta.*

timpul in in care sunt efectiv transferate datele intre memorie si disc numit *timp de transfer*.

Timpul total de servire a unei cereri de transfer este suma accestor trei timpi.
SO poate imbunatati timpul de acces la disc printr-o planificare optima a servirii cererilor de acces la disc.
In cazul sistemelor cu multiprogramare mai multe procese solicita cereri de intrare iesire I/O care vor fi onorate pe rand de catre SO. Dupa ce o cerere este rezolvata SO alege o noua cerere din coada de cereri pentru a fi onorata (servita). Din activitatile desfasurate pentru efectuarea unui acces trebuie sa spunem *timpul de cautare(seek)* are ponderea cea mai mare in rezolvarea unei cereri. SO prin ordonarea intr-un anumit mod a cererilor de I/O se pot minimiza *timpul de cautare* ceea ce conduce la o reducere a timpului total de prelucrare.
Atunci cand un proces al unui utilizator sau al SO lanseaza o cerere de I/O este posibil ca unitatea de disc (controlerul sau unitatea propriuzisa) sa nu fie disponibila deoarece executa o cerere soliciata anterior. Aceste cereri care nu pot fi servite pentru ca unitatea nu este disponibila se introduc intr-o coada de cereri de I/O. Onorarea acestor cereri se va face dupa ce unitatea devine disponibila. Servirea cererilor trebuie sa urmeze o ordine cronologica numai pentru cererile provenite de la un acelasi proces. Cererile din coada de cereri de I/O care provin de la procese diferite dar de prioritae egala pot fi onorate in orice ordine cu conditia de a se respecta ordinea cronologica pentru cererile provenind de la acelasi proces. In acest fel SO poate interveni in onorarea cererilor astfel incat sa obtina o reducere a timpului total de acces la disc.

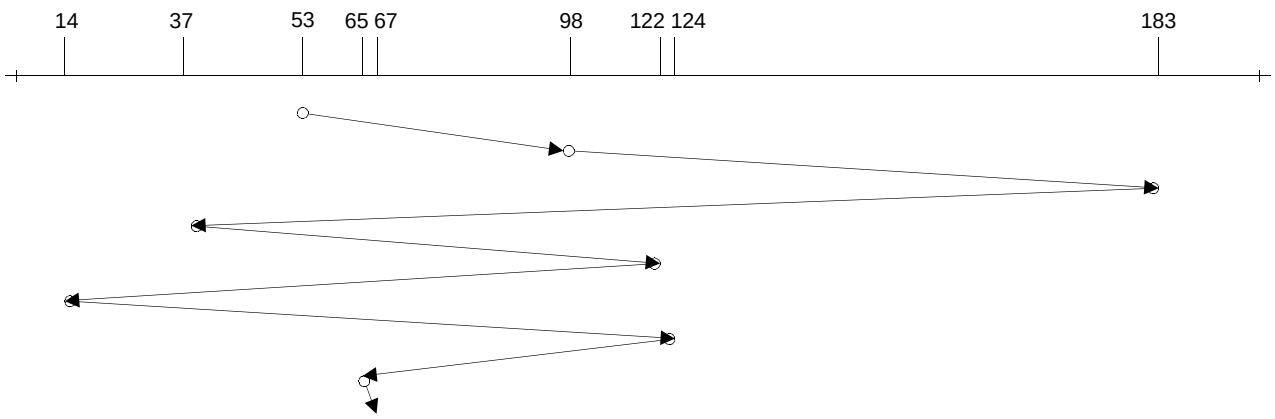
6.51.Planificarea FCFS (First Com First Send) (Primul Sosit - Primul Servit)

Acest algoritm este foarte simplu dar nu realizeaza o optimizare a timpului de acces (timp de deplasare a capetelor).
Vom considera de exemplu o coada de cereri de I/O care solicita accesul pe urmatoarele piste:

98, 183, 37, 122, 14, 124, 65,67

Presupunand ca in momentul actual capetele de citire-scriere se afla pozitionate pe pista 53, SO va onora cererile in ordinea sosirii lor ceea ce va insemna o deplasare a capetelor de citire-scriere de la 53 la 98, apoi la 183, 37,122,14,124,65 si 67.
Daca contorizam numarul de piste peste care trec capetele in aceasta secventa de cereri obtinem: 640 piste (98-53)+(183-98)+(98-137)+.....
Numarul de piste traversate este o masura a timpului consumat pentru deplasarea capetelor pentru a rezolva cererile din lista.
Numarul de piste traversate poate fi un criteriu de evaluare a performantei diversilor algoritmi utilizati pentru planificarea deplasarii capeteleor de citire-scriere la disc.

Daca reprezentam grafic deplasarile capetelor de citire-scriere in cazul algoritmului FIFO obtinem:

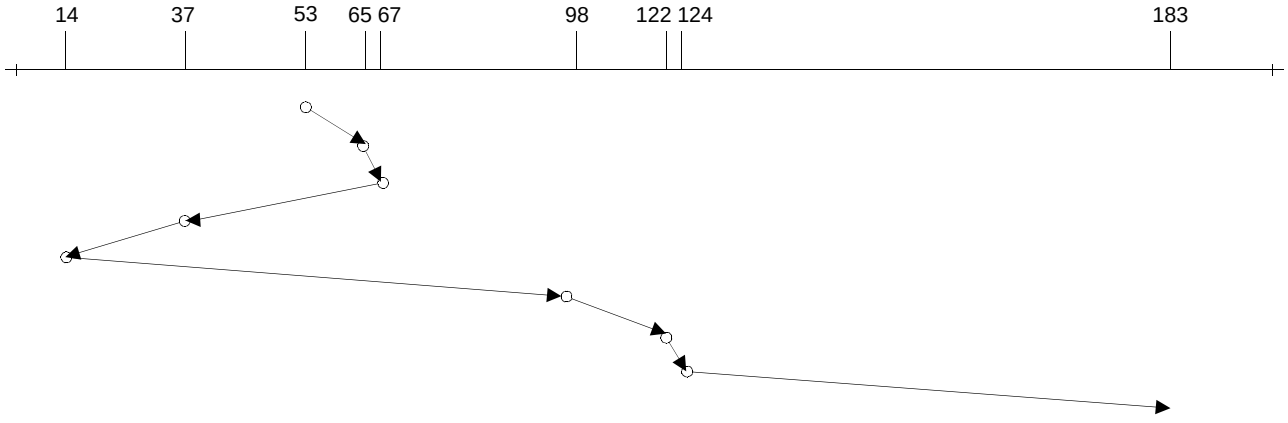


Planificarea discului de tip FCFS

Din aceasta diagrama se observa ca printr-o alta ordine a comenzilor de deplasare a capetelor de citire-scriere este posibila o imbunatatire a *timpului de cautare* (seek time)

6.5.2.Planificarea SSTF (Shortest Seek-Time First)
(cel mai scurt timp de cautare -primul)

Acest algoritm selecteaza din coada de cereri, pe cea care implica cel mai scurt "seek" din pozitia curenta. Timpul de pozitionare (seek time) fiind proportional cu diferenta intre piste se va obtine un optim momentan din punct de vedere al timpului.
Algoritmul reduce substantial *timpul de cautare*, numarul total de piste "traversate" pentru aceiasi secventa fiind 208 fata de 640 in cazul algoritmului FCFS.
Reprezentat grafic:

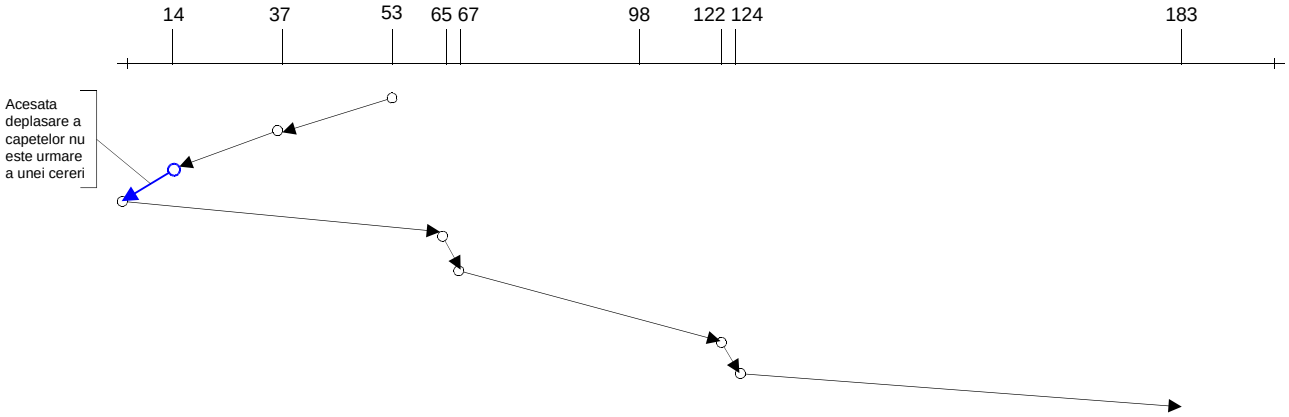


Planificarea discului de tip SSTF

Algoritmul SSTF optimizeaza in buna masura timpul de pozitionare dar sufera de un inconvenient. Exista posibilitatea ca anumite cereri sa astepte foarte mult pana cand vor fi servite daca acestea sunt cereri pentru pozitionarea pe un cilindru indepartat de cilindrul unde sunt pozitionate momentan capetele de citire-scriere.

6.5.3.Algoritmul SCAN

Algoritmul SCAN incepe onorarea cererilor de la un capat catre celalalt capat al discului si inapoi.
SO stie in fiecare moment pozitia capetelor si directia de deplasare a capetelor (sensul).
In exemplul nostru daca pozitia initiala este 53 si capetele se deplaseaza spre inceputul discului (catre adrese mici) pentru aceiasi coada de cereri se obtine urmatoare diagrama:



Planificarea discului de tip SCAN

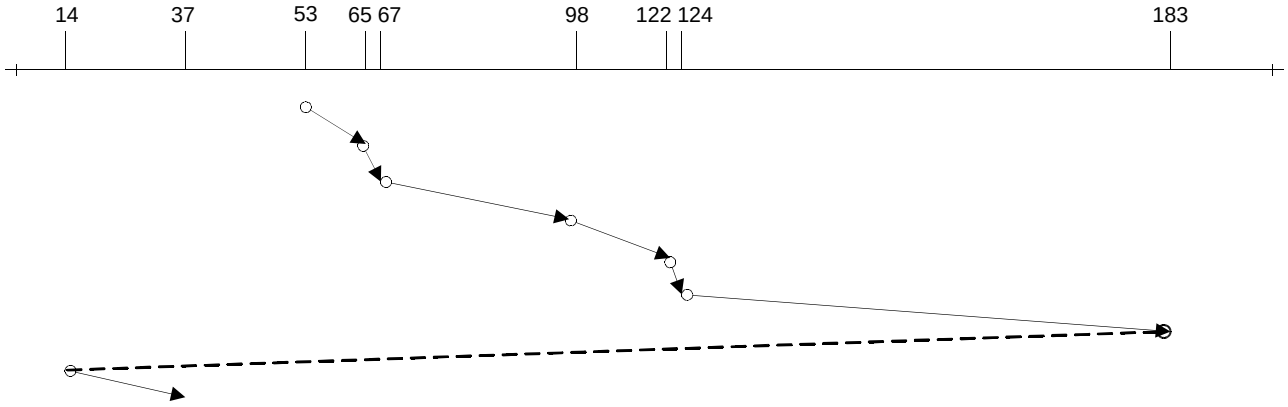
Se observa ca dupa satisfacerea cererii de pozitionare pe pista (cilindrul) 14 capetele se deplaseaza pana la inceputul discului chiar daca nu este o cerere de pozitionare in aceasta zona, dupa care incepe deplasarea pana la celalalt capat al discului.
Acest algoritm realizeaza o servire achitabila a tuturor cererilor dar are si o particularitate.
Particularitatea acestui algoritm este aceea ca daca soseste o cerere de pozitionare pe un cilindru aflat "inaite" pe directia de deplasare, atunci acea cerere va fi onorata imediat. In timp ce o cerere de pozitionare "in urma" fata de pozitia curenta a capetelor pe directia de deplasare, atunci acea cerere va fi onorata abea la "intoarcerea" capetelor. Acest algoritm functioneaza cu un "lift" automat care se deplaseaza in sus si in jos de la parter la ultimul etaj si invers, care opreste la fiecare etaj. De aceea acest algoritm se mai numeste si *algoritmul ascensorului de planificare a discului*.
Acest algoritm nu asigura intodeauna o astepatare proprtionala a cererilor. Orce cerere care soasete "in urma" pozitiei capetelor asteapta mult mai mult decat orcare cerere de pozitionare "in fata".

6.5.4.Planificarea C-SCAN

Este o varianta a algoritmului SCAN modificata. Ea asigura o parcurgere circulara (Circular SCAN) al cilindrilor discului.
Acest algoritm determina o deplasare a capetelor de la cilindrul "0" la cilindrul "maxim" si invers dar servirea cererilor nu se face decat pe un sens al deplasarii. In acest fel dupa ce capetele ating cilindrul "maxim" al discului ele se deplaseaza imediat la cilindrul "0" fara ca in acest parcurs sa onoreze vre-o cerere. Lucrurile se petrec ca si cum cilindrii discului ar reprezenta o lista circulara, adica dupa cilindrul "maxim" ar urma cilindrul "0", "1",....."maxim", "0","1", ... Retragerea (pozitionarea) capetelor pe cilindrul zero se face foarte rapid ea ne fiind echivalenta ca timp cu o pozitionare pe cilindrul "0". Timpul de pozitionare in acest caz este foarte mic si se poate neglija.
Acest algoritm asigura un timp mai uniform de asteptare a servirii cererilor.
Specific ambilor algoritmi SCAN cat si C-SCAN este faptul ca intodeauna are loc o deplasare a capetelor intre cele doua extreme ale discului. Adica capetele se deplaseaza de la primul cilindru ("0") la ultimul ("max") si invers chiar daca nu sunt cereri de pozitionare pe acesti cilindrii.

6.5.5.Planificarea LOOK si C-LOOK a discului

Algoritmul LOOK "priveste" dupa o cerere inainte de a deplasa capetele in acea directie. Astfel in lista de cereri din exemplul nostru daca capetele se afla initial pe cilindrul 53 atunci cele mai multe cereri sunt catre numerele de cilindru mari deci capetele se vor deplasa pe cilindrul 65.
Si acest algoritm are o varianta C-LOOK adica atunci cand capetele ajung pe pozitia celui mai mare numar de cilindru din lista cerut, imediat capetele se deplaseaza la celalalt capat, adica la cilindrul cu numarul cel mai mic din lista de cereri.



Planificarea C-LOOK a discului

6.5.6.Selectarea algoritmului de planificare a discului

Existand mai multi algoritmi de planificare a "servirii" cererilor de acces la disc cum vom alege un anumit algoritm?
Algoritmul SSTF este destul de utilizat si usor de implemantat iar algoritmul SCAN este mult mai potrivit pentru sistemele cu activitate intensa a discului.
In general este posibil sa se defineasca un algoritm optim dar efortul de calcul poate conduce la cresterea timpului nejusticat. Indiferent de algoritmul folosit efectul sau depinde de numarul cererilor aflate in coada. Daca in mod frecvent in coada nu se afla mai mult de o cerere este inutil sa folosim orce optimizare a deplasarii capetelor de citire-scriere pentru ca efectul ar fi insezizabil.
Trebuie mentionat ca cererile de acces la disc sunt puternic influentate de metoda de alocare a spatiului disc pentru fisiere.
Un program care citeste un fisier care este alocat pe un spatiu contiguu de disc va genera cereri pe cilindrii foarte apropiati, in timp ce accesul la un fisier alocat "inlantuit" sau "indexat" va genera cereri pe cilindrii disperasati oriunde pe disc.
Deasemeni este foarte important d.p.d.v. al timpului consumat pentru pozitionarea capetelor, unde sunt plasate pe disc directoarele discului si blocurile de index. Este stiut ca pentru a utiliza un fisier el trebuie "deschis" operatie care implica citirea informatiilor de control ale fisierului adica a directoarelor si a blocurilor index. Plasarea directoarele la oricare din extremitatile discului poate fi uneori mai dezavantajoasa decat plasarea lor la mijlocul discului. Alteori din considerente "constructive" este mai eficient sa se plaseze directoarele in prima parte a discului.
Alegerea algoritmlor pentru planificare a discului se face tinand cont atat de caracteristicile constructive ale discului cat si de caracteristicile legate de particularitatile concrete date de SO.

6.8Cresterea performantei si a fiabilitatii discurilor.

Discul tinde sa devine susbsistemul SC hardware care restrange cel mai mult posibilitatile de crestere a performantelor si fiabilitatii Sistemelor Calculator.
Daca in ultimul timp performanta unitatilor de prelucrare (CPU) a crescut exponential nu acelasi lucru se poate spune despre performantele discurilor magnetice. Incepand din anul 1970 si pana acum performantele si fiabilitatea discurilor magnetice au crescut cu mult mai putin decat performanta si fiabilitatea CPU-ului , memoriei, etc. De exemplu discurile anilor 70' aveau un *timp de cautare* de cca 50-100ms in timp ce astazi discurile au acest timp de cca 5-10ms.
Deasemeni defectarea unui disc intr-un sistem de calcul este considerata cel mai "catastrofic" incident posibil in exploatarea sistemelor de calcul. Si asta nu numai prin costurile efective ale discului care trebuie inlocuit cat mai ales prin "costurile" informatiei care se pierde. De obicei se prevad proceduri de "back-up" care realizeaza o copie a datelor la perioade prestabilite de timp. Restaurarea datelor dupa schimbarea discului defect, este o operatie de durata si in plus ea nu poate reface imaginea datelor exact din momentul defectarii discului.
Cea mai cunoscuta metoda prin care se incearca sa se imbunatateasca performanta si fiabilitatea la discurile magnetice este prin paralelism a transferului si toleranta la defecte(fault tolerant).
Astfel au aparut discurile numite RAID - "Redundand Array Inexpensive (Independent)Disks".
Aceasta noua clasa de periferice de tip disc folosesc o "matrice" redundanta de unitati de disc "ieftine(Independente)". (Termenul "Inexpensive : ieftin" a fost inlocuit ulterior cu termenul "Independent").
Discurile RAID folosesc mai multe unitati clasice de disc magnetic IDE sau SCSI, care lucreaza impreuna sub controlul unei aceleiasi Unitati de Control (Controler).
Pentru cresterea performantei (vitezei de transfer) discurilor se foloseste metoda dispersiei informatiei (numita "stripping" sau "interleaving") pe unitati de disc diferite, adica pe fiecare dintre discuri se memoreaza cate un "sub-bloc" care reprezinta numai o parte a unui "bloc de date". Altfel spus informatia care se gaseste intr-un bloc de date care este transferata cu o singura comanda , este impartita (dispersata) pe mai multe discuri. Atunci cand se scrie sau se citeste un "bloc de informatie", prin lansarea unei singure comenzii de I/O, transferul se va face in paralele pentru fiecare "sub-bloc" care compun blocul de date. Acest mod de functionare conduce la o reducere a timpului de transfer. Pentru a obtine o viteza maxima de transfer discurile sunt "sincronizate" astfel incat sa nu apara intarzieri (desincronizari) intre transferurile sub-blocurilor apartinand aceluiasi bloc.
Pentru cresterea fiabilitatii se asigura o functionarea de tip *tolerant la defecte* (Fault Tolerant) prin:

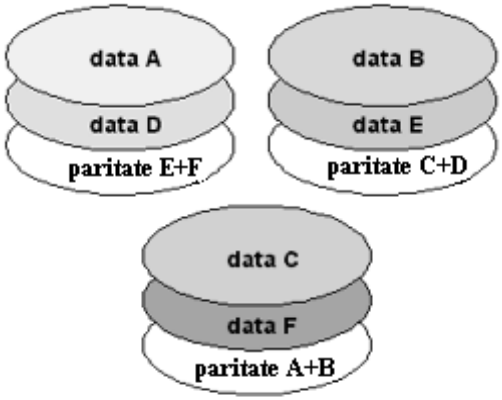
-redundanta informatiei prin duplicarea informatiei pe un alt set de discuri;

sau

-prin calcularea si inscrierea pe un alt disc a *paritatii informatiei*.

Principial acesta este modul de organizare si functionare al discurilor RAID si este transparent pentru programator sau SO. Sarcina implementarii acestui mod de functionare ii revine controlerului de disc.
Cel mai simplu mod de organizare a discurilor RAID este prin "oglindire" (mirroring) care consta in pastrarea cate unei copii a fiecarui disc. Aceasta solutie este bineanteles foarte scumpa. De obicei se foloseste o alta tehnica numita cu "paritate de bloc intretesuta". Aceasta organizare foloseste distributia informatiei unui bloc pe un numar de discuri diferite si in plus construieste un extra bloc de paritate.
Atat "dispersia" datelor pentru asigurarea cresteri vitezei de transfer cat si dublarea informatiei sau folosirea "paritatii" pentru asigurarea functionarii *tolerant la defecte* se poate face prin mai multe cai. Aceste modalitati diferite de implementare a tehnologiei RAID a condus la aparitia mai multor tipuri de discuri RAID.

Unul dintre cele mai performante implementari ale principiilor RAID este asa numita RAID Nivel5 Pentru acest tip de functionare datele sunt "imprastiate" pe 3 sau mai multe discuri iar bitul de paritate pentru 2 discuri este pastrat pe un al treilea.



Organizarea informatiei intr-un subsistem RAID Nivel 5

CAP.VII SISTEMUL DE FISIERE

Pentru cea mai mare parte a utilizatorilor *sistemul de fisiere* este cel mai vizibil aspect al SO. Fisierele stocheaza atat programele cat si datele cu care lucreaza utilizatorul. SO implementeaza conceptul abstract de *fisier* prin administrarea dispozitiveleor periferice ale SC, cum ar fi benzile sau discurile magnetice. *Fisierele* sunt implementate de catre SO pe dispozitive fizice numite dispozitive periferice.
Fisierele de obicei sunt organizate (grupate) in directoare pentru a simplifica utilizarea lor.
SO implementeaza modul in care mai multi utilizatori au acces la fisiere prin mecanisme de *protectie a fisierelor*.

7.1.Organizarea sistemului de fisiere

Administrarea fisierelor este un serviciu pe care SO il pune la dispozitia utilizatorului. SC pentru a stoca informatia pe diversi suporti de informatie, cei mai cunoscuti fiind banda si discul magnetic, Exista o varietate mare de dispozitive periferice, fiecare dintre ele avand propriul sau mediu de stocare cu caracteristici particulare.
SO ofera utilizatorului o interfata logica uniforma asupra datelor stocate indiferent de mediul de stocare. Pentru aceasta SO defineste o entitate logica de memorare numita *fisier* care ascunde caracteristicile fizice particulare ale diverselor periferice pe care sunt efectiv memorate fisierele.
Sistemul de fisiere consta din doua parti:

- colectii de fisiere reale, fiecare din ele continand diverse informatii;
- directorul de structura care furnizeaza informatii generale despre toate fisierele din sistem.

7.1.1.Conceptul de *fisier*

Ce este acela un fisier?
Un fisier este o colectie da date inrudite (legate logic unele de altele) definite de cel ce creaza fisierul. De obicei fisierele reprezinta programe (in format sursa sau obiect) si date. Fisierele pot fi neformatate cum ar fi fisierele text, sau pot fi formatate rigid adica xu o structura fixa prestabilita.
In general fisierul este o secventa de biti, bytes (octeti), linii sau inregistrari al caror inteles este definit de utilizatorul care l-a creat.
Un fisier are asociat un numar cu ajutorul caruia poate fi referit (identificat) in programe.
Un fisier are cateva caracteristici: tipul fisierului, data cand a fost creat, numele sau numarul de cont al celui care l-a creat, lungimea, etc.
Intr-un fisier pot fi memorate diverse tipuri de date (informatii): programe sura, obiect, date numerice, texte, imagini sunete, etc.
Orice informatie care poate fi manipulata de SC poate fi stocata in fisiere.
Un fisier are definita o anumita structura corespunzatoare tipului sau.
SO poate sa opereze (manipuleze) cu un fisier corespunzator cu tipul fisierului. Acest lucru creste complexiatatea SO si nu este intotdeauna recomandata.
SO UNIX ofera maximum de flexibilitate, dar minim de suport neimpunand in nici un fel tipul fisierului. In SO UNIX fiecare aplicatie interpreteaza structura (tipul) fisierului de intrare.
Cel mai frecvent fisierele sunt pastrate pe disc. Unitatea de transfer intre SO (memoria operativa) si disc este *blocul*. Dimensiunea blocului este multiplu de sectoare. Toate blocurile au aceiasi dimensiune. Blocurile se mai numesc si *inregistrari fizice* si nu au de obicei aceiasi lungime cu a *inregistrarii logice*.

Dimensiunea *inregistrarii logice* determina cate *inregistrari logice* sunt cuprinse intr-o *inregistrare fizica* (bloc). Fisierul se considera ca fiind o secventa de *inregistrari logice* care ocupa fizic (pe disc) o succesiune de blocuri (*inregistrari fizice*) nu neaparat alaturate fizic.
Conversia din *inregistrari logice* in *inregistrari fizice* si invers este realizata de catre SO.
Nu intotdeauna numarul *inregistrarilor logice* al unui fisier este multiplu sau submultiplu al numarului de *inregistrari fizice* (blocuri). Unui fisier ii este alocat intotdeuna un numar intreg de blocuri. Acest lucru poate determina aparitia *fragmentarii interne*.
Toate sistemele de fisiere sufera de fragmentare interna. Cu cat creste dimensiunea *inregistrarii fizice* (dimensiunea blocului) ca atat creste fragmentarea interna.

7.1.2. Directorul de structura

Fisierele in SC sunt reprezentate prin intrari in *directorul de periferic* numit si *Tabela de Volum*.
Directorul perifericului contine informatii despre toate fisierele aflate pe disc. Fiecare intrare a *directorului perifericului* contine informatii despre cate un fisier (nume, locul unde se afla, dimensiunea, tip, etc.).
Datorita cresterii dimensiunilor discurilor, numarul fisierelor pe un periferic a crescut. In conditiile in care si numarul utilizatorilor creste a devenit dificil pentru utilizatori de a-si gestiona fisierele. Rezolvarea acestei probleme s-a facut prin introducerea unei organizari a sistemului de fisiere bazata pe *directoare*.
Structura de Directoare ofera un mecanism eficient si flexibil pentru organizarea si administrarea unui numar foarte mare de fisiere aflate pe un acelasi disc, discuri diferite si chiar discuri aflate pe siteme de calcul diferite. In acest ultim caz utilizatorul nu este necesar sa foloseasca tehnici diferite pentru manipularea fisierelor sale el tratandul-le ca si cand ar fi pe discul "local".
In prezent cele mai multe *sisteme de fisiere* utilizeaza doua tipuri de structuri de directoare separate:

- directorul de periferic (Tabela de Volum);
- directoare de fisiere.

Directorul de periferic este stocat pe fiecare periferic fizic(disc). In principal el contine informatii referitoare la proprietatile fizice ale fisierelor: unde se afla fisierul,, numarul de blocuri alocate fisierului, lungimea efectiva a fisierului, etc.
Directoarele de fisiere contin cate o intrare pentru fiecare fisier existent. Fiecare intrare cuprinde proprietati logice ale fisierului: nume, tip, utilizatorul proprietar, informatii de protectie acces si intrarea in directorul de periferic care descrie proprietatile fizice ale perifericului, etc.
Fisierul directoarelor are o structura logica de organizare identica cu cea a fisierelor de pe disc. Fiecare intrare in *directorul de fisiere* (referistoare la un fisier) contine o referinta (pointer) la intrarea corespunzatoare in *directorul de periferic* corespunzatoare fisierului unde se apla caracteristicile lui fizice.
Informatiile pastrate pentru fiecare fisier in directoare difera de la un SO la altul. In cele mai multe cazuri aceste informatii sunt:

Numele fiserului. Reprezinta un nume "simbolic" atribuit fisierului de catre creatorul sau folosit de utilizatorii SC pentru referirea la fisier. Este singura infoarmatie pastrata in format ASCII.
*Tipul fisierului. S*e foloseste in SO care accepta diferite tipuri de fisiere.
Locatia. Acest parametru indica perifericul si locul de pe periferic unde se afla fisierul.
Dimensiunea. Reprezinta dimensiunea actuala a fisierului (in octeti, cuvinte sau blocuri).
Pozitia curenta. Indica pozitia in fisier unde se va face *citirea* sau *scrierea*.
Protectia. Sunt informatii de control al accesului la fisier.
Data, timpul. Sunt informatii referitoare la momentul creerii fisierului, ultima modificare, ultima utilizare, etc.

Pentru fiecare fisier pentru pastrarea acestor informatii se poate ajunge la un spatiu disc de 16 pana la 1000 bytes/fisier. Atunci cand numarul fisierelor de pe disc este mare dimensiunea directorului poate creste foarte mult.
Directoarele sistem pot fi privite ca o tabela de simboluri care translateaza numele fisierului in intrarea corespunzatoare fisierului in director. Directoarele sistem sunt organizate in asa fel incat sa permita:

inserarea unei intrari;
stergerea unei intrari;
cautarea unei anumite intrari si
listarea tuturor intrarilor din directorului

Directoarele sistem pot fi organizate in diferite feluri pentru a putea oferi mecanisme eficiente care sa permita realizarea acestor functiuni.
Cel mai simplu mod de organizare al intrarilor in directoare este sub forma de lista liniara, cautarea unei intrari facandu-se printr-o cautare liniara. Acest lucru se implementeaza simplu dar nu este eficient din punct de vedere al timpului consumat.
O alta solutie de organizare a intrarilor in directoare este sortarea intrarilor ceeace ar permite scaderea timpului de cautare prin utilizarea algoritmilor de cautare binara. Totusi acesti algoritmi sunt mai complexi si implica ca inca de la creare sa se pastreze o lista sortata a intrarilor. Aceasta metoda complica inserarea unei intrari sau stergerea.
O alta metoda de organizare care poate fi utilizata pentru fisierul director este utilizarea *tabelelor de dispersie(tabelei hash)*. Utilizarea *tabelelor de dispersie* poate reduce foarte mult timpul de cautare , inserare sau stergere in director dar trebuie prevazute proceduri suplimentare pentru evitarea *colisiunilor* (situatia cand doua nume de fisier diferite prin algoritmul de *dispersie(hashing)* furnizeaza aceiasi pozitie in fisierul director). Dezavantajul utilizarii structurii cu *tabele de dispersie* este dimensiunea fixa a a acesteia si dependenta intre *functia de dispersie(hash)* si dimensiunea *tabelei de dispersie.*

7.2.Operatii cu fisiere

Fisierul reprezinta o structura abstracta de date. Pentru definirea completa a *structurii de tip fisier* este necesar sa definim operatiile care se pot face asupra fisierelor.
SO furnizeaza urmatoarele operatii asupra fisierelor: creare, scriere, citire, resetare (reinitializare) si stergere.
Acestea sunt functii puse la dispozitia utilizatorului ele fiind operatii de baza cu ajutorul lor putand implementa si alte operatii (de ex.: Renumirea unui fisier). Utilizatorul pentru a executa operatii cu fisiere utilizeaza asa numitele "apeluri sistem" care sunt tratate de catre SO care executa efectiv functia respectiva.

1. Crearea unui fisier. Se face in doi pasi. Primul este gasirea spatiului pentru fisier si al doilea trebuie gasita o intrare pentru fisier in director. In intrarea in directorul de periferic se inregistreaza numele si localizarea fisierului.
2. Scrierea fisierului. Pentru a scrie un fisier in apelul sistem se precizeaza numele fisierului si informatia ce trebuie scrisa in fisier. Sistemul cauta in director pentru a gasi localizarea fisierului. In intrarea corespunzatoare in director se gaseste si un "pointer" la blocul curent (la prima scrire este chiar inceputul fisierului). Utilizand acest pointer se calculeaza adresa urmatorului bloc si se poate scrie informatia.
3. Citirea unui fisier. Se apeleaza functia sistem de citire care are ca parametrii numele fisierului si adresa de memorie unde va fi citit urmatorul bloc. SO cauta in director intrarea corespunzatoare fisierului unde se gaseste pointerul la blocul urmator ce se citeste. Se citeste si apoi se actualizeaza pointerul pentru urmatorul bloc. In general un fisier fie este

citit fie este scris. In acest fel atat la citire cat si la scriere se foloseste acelasi pointer care furnizeaza pozitia curenta in fisier .

4. Resetarea fisierului. Se cerceteaza directorul si la intrarea corespunzatoare fisierului se "reseteaza" pointerul "pozitie curenta in fisier" la adresa de inceput a fisierului.Aceasta operatie nu implica decat actualizarea directorului fisierului.
5. Stergerea fisierului. Pentru stergerea unui fisier se cauta in director numele fisierului. Daca se gaseste fisierul (fisierul exista) in intrarea corespunzatoare se invalideaza sau se completeaza cu spatii eliberandu-se intrarea respectiva.

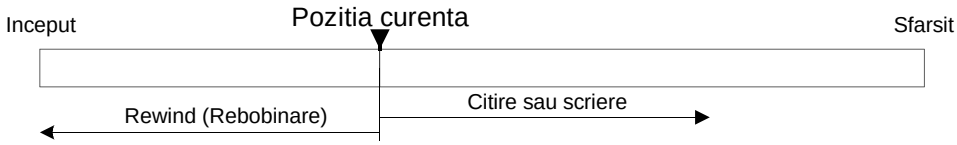
Toate aceste operatii implica cautarea in directoare dupa nume pentru a gasi intrarea asociata numelui. Intrarea in director asociata unui fisier contine cele mai importante informatii necesare pentru a lucra cu fisierul.
Pentru a evita cautarile repetate pe un acelasi fisier cele mai multe sisteme folosesc o operatie suplimentara OPEN (DESCHIDERE) fisier. SO creaza in memorie la deschiderea unui fisier (prima actiune asupra fisierului) o tabela pentru fiecare fisier. In aceasta tabela se copiaza informatiile corespunzatoare din director asociate fisierului. Atunci cand este solicitata o operatie asupra fisierului este utilizat un index la aceasta tabela si nu mai este necesara cautarea in director. Atunci cand programul se termina fisierul este inchis (CLOSE).
Unele sisteme deschid automat fisierul la prima referire iar altele necesita deschiderea explicita de catre utilizator printr-o functie OPEN. Fisierul este inchis automat, in ambele cazuri la terminarea programului care a determinat deschiderea fisierului.
Aceste cinci operatii descrise reprezinta un set minimal de operatii asupra fisierelor. Cu ajutorul lor se pot compune operatii mai complexe asupra fisierelor. Deseori utilizatorul trebuie sa execute operatii asupra fisierelor, cum ar fi: editarea unui fisier si modificarea continutului fisierului, adaugarea informatiilor noi la sfarsitul fisierului, etc.

7.3.Metode de acces

Fisierele stocheaza informatia. Atunci cand sunt utilizate aceasta informatie trebuie accesata si citita in memoria principalaa (operativa). Exista diverse modalitati prin care poate fi accesata informatia continuta in fisiere.

7.3.1.Accesul secvential

In cazul *accesului secvential* informatia din fisier este prelucrata in ordine, o inrgistrare dupa alta. Acesta este cel mai raspandit mod de acces. Exemplul cel mai banal sunt editoarele de texte care acceseaza informatiaa in aceasta maniera.
Cea mai mare parte a operatiilor cu fisiere sunt *citirea* (read) si *scrierea* (write).
Operatia de *citire secventiala* inseamna citirea urmatoarei portiuni a fisierului si avansarea pointerului . Similar operatia de scriere secventiala inseamna scrierea in continuarea ultimei inregistrari din fisier (dupa aceasta) si adaugarea dupa inregistrarea scrisa a marcii de sfarsit de fisier (end of file).
Deasemenea fisierul poate fi *resetat*: la inceputul fisierului (pointer=0) sau se pot *sari* (skip) inaintr sau unapoi un numar n de *inregistrari logice*. Pentru anumite sisteme n poate fi orice numar intreg sau pentru alte sisteme n poate fi numai 1 . Acest mod de prelucrare al unui fisier la care "pasul" de deplasare in fisier pote fi numai "1" se numeste *acces secvential*. *Accesul secvential* se bazeaza pe modelul benzii magnetice .



Accesul Secvential la Fisiere

7.3.2.Accesul direct

O alta metoda de acces este accesul direct care se bazeaza pe modelul disc al fisierului.
Pentru accesul direct fisierul este vazut ca o secventa numeroatat de inregistrari sau blocuri. Accesul direct permite ca un bloc oarecare sa fie citit sau scris. De exemplu se poate citi blocul 21, apoi citi blocul 47 apoi se poate citi blocul 4.
In cazul accesului direct nu este nici o restrictie privitor la ordinea in care se citesc sau se scriu blocurile. Accesul direct este folosit cel mai frecvent in accesul informatiilor in baze de date de dimensiuni mari. Atunci cand un utilizator lanseaza o interogare referitor la un anumit subiect se calculeaza care bloc contine raspunsul si se citeste direct acest bloc.
Numarul de bloc furnizat de catre utilizator SO este in mod normal numar relativ de bloc . Numarul relativ de bloc este numarul blocului raportat la inceputul fisierului. Astfel primul bloc relativ al fisierului este 0, urmatorul 1 s.a.m.d. Adresele absolute ale acestor blocuri putand fi de exemplu 1321 pentru primul bloc si 3602 pentru al doilea s.a.m.d. Bineinteles ca blocurile fizice alocate fisierului depind de algoritmul de alocare.

7.3.3Concluzii

Nu toate SO suporta accesul secvential si accesul direct la fisiere. Anumite sisteme suporta numai accesul secvential altele accesul direct. Anumite sisteme (necesita) impun ca inca de la crearea fisierului sa se defineasca fisierul secvential sau direct si ele pot fi accesate apoi numai in concordanta cu declaratia de la crearea fisierului.
Pentru fisierele cu acces direct este foarte simplu sa fie simulat accesul secvential. Invers insa simularea accesului direct la un fisier cu acces secvential este posibil dar ineficient.
Pot fi construite si alte metode de acces pornind de la metoda accesului direct.
Aceste metode aditionale in general implica construirea unor tabele de index pentru fisier. Aceasta tabele de index (ca si cuprinsul unei carti) contine pointeri la blocurile fisierului. Pentru a gasi o intrare in fisier se cerceteaza mai intai indexul apoi cu indexul gasit se acceseaza direct blocul cu informatia cautata.
In cazul fisierelor apare si alta problema importanta si anume cum se rezolva problema accesului patajat la un fisier. Aceasta problema se mai numeste si consistenta semantica a sistemului de fisiere. Consistenta semantica este un criteriu de evaluare al oricarui sistem de fisiere care suporta partajarea fisierelor.
Sistemul de fisiere are caracteristic semantica acceselor simultane ale mai multor utilizatori (accese multiple) la un fisier. In particular aceasta semantica trebuie sa precizeze cand modificarea unei date de catre un utilizator este observabila de la un alt utilizator .

7.4.Protectia fisierelor

Pentru informatia pastrata intr-un sistem calculator o problema majora este protectia. Cum informatia se afla in fisiere vom discuta in continuare despre protectia fisierlor. Protectia fisierelor este privita din doua puncte de vedere:

protectia impotriva distrugerii fizice siguranta fisierelor (reliability);
protectia impotriva accesului neavizat protectia fisierului.

Siguranta informatiilor (fisierelor) este realizata prin duplicarea fisierelor. Multe sisteme au sisteme de programe care automat sau printr-o comanda a utilizatorului copiaza fisierele disc pe o banda magnetica (copie de rezerva) la intervale regulate de timp.
Cauzele pentru care sistemul dee fisiere se poate distruge sunt multiple: functionare defectuaoasa a sistemului calculator hard, suprasarcini sau caderi de tensiune, defectarea unitatilor de disc etc.
Protectia informatiei (fisierelor) poate fi realizata prin mai multe metode.
Pentru sisteme single-user (monoutilizator) metodele de protectie sunt putin evoluate si ele se realizeaza prin impiedicarea accesului fizic la Sistemul calculator.
Pentru sistemele multi-user mecanismele trebuie insa sa fie mai complexe.
Necesitatea protectiei fisierelor este un rezultat direct al capabilitatilor de acces la fisiere.
La sistemele la care nu este permis accesul la un fisier a altor utilizatori decat utilizatorul proprietar (cel care a creat fisierul) protectia nu este necesara. Astfel una din extreme a protectiei este de a crea protectie completa prin impiedicarea accesului oricarui alt utilizator decat proprietarul. Cealalta extrema a protectiei este de a permite accesul liber tuturor utilizatorilor. Ambele aceste abordari sunt extreme pentru a putea fi utilizate sistemele de fisiere realizeaza protectia printr-un acces controlat .
Mecanismul de protectie realizeza accesul controlat limitand tipurile de acces la fisiere care poate fi permis unui utilizator.
Accesul este permis sau interzis in functie de diferiti factori, unul din factori fiind tipul de acces solicitat.
Tipurile de operatii asupra fisierelor care sunt controlate de sistemul de fisiere sunt:

Citire (READ) Citirea dintr-un fisier.
Scriere (WRITE) Scrierea sau rescrierea unui fisier.
Executie (EXECUTE) Incarcarea unui fisier in memorie si executarea lui.
Adaugare (APPEND) Scrierea unei informatii noi la sfarsitul fisierului.
Stergere (DELETE) Stergerea unui fisier si eliberarea spatiului ocupat de ele in vederea reutilizarii.

Alte operatii cum ar fi, *renumirea* (scimbarea numelui), *copierea, editatrea* unui fisier pot fi deasemnei controlate de sistemul de fisiere. Pentru multe sisteme de operare acestea reprezinta *functiuni de nivel-inalt* care sunt implementate prin programe speciale care la randul lor apleaza *functiile (apelurile) de nivel-jos* care executa efectiv operatiile controlate de sistemul de fisiere. Protectia este implementata numai prin *functiile de nivel-jos*.
Operatiile cu directoare trebuie deasemeni controlate. Protectia operatiilor cu directoarele se face de o maniera diferita. Trebuie sa fie controlate crearea si stergera fiserelor din directoare si chiar accesul la continutul unui director. Astfel incat daca este cazul chiar tiparirea listei fisierelor aflate intr-un director sa nu fie permisa pentru anumiti utilizatori.
Pot fi imaginate foarte multe mecanisme de protectie. Ca intodeauna fiecare mecanism are avantajele si dezavantajele lui. De obicei se alege un anumit mecanism de protectie astfel incat el sa fie potrivit unui anumit tip de aplicatie. Astfel este evident ca nu vom folosi acelsi mecanism de protectie in cazul unui calculator mic care este utilizat de cativa membrii ai unui grup ca in cazul de exemplu a unui sistem calculator al unei banci la care au acces mii de utilizatori.

7.4.1.Protectia prin nume.

Aceasta metoda de protectie se bazeaza pe faptul ca pentru a avea acces la un fisier trebuie cunoscut numele acestuia. Daca un utilizator nu poate da numele unui fisier la care solicita accesul atunci el nu poate fi accesat. Aceasta schema de protectie presupune ca nu exista o metoda de a afla numele fisierelor altui utilizator iar aceste nume de fisiere nu sunt usor de "ghicit".
In realitate in sistemele actuale numele fisierelor sunt nume "mnemonice" care pot fi usor "ghicite" ceeace face aceasta metoda putin utilizabila.

7.4.2.Protectia prin *parola* (*password* = cuvant de trecere).

O alta metoda de protectie a accesului la fisiere este asocierea unei *parole* fiecarui fisier. Accesul la un fisier nu este permis decat dupa ce utilizatorul care solicita accesul la fisier furnizeaza *parola* asociata fisierului.
Daca *parola* se alege ca un sir de caractere "aleator" si este schimbata frecvent aceasta metoda de protectie poate fi suficient de sigura. ea are insa cateva dezavantaje:

Daca se alege cate o *parola* diferita pentru fiecare fisier numarul de *parole* necesar a fi pastrat devine foarte mare si metoda devine imparcticabila. dac se utilizeaza o singura parola pentru toate fisirerele atunci metoda de protectie devine nesigura . Se poate folosi o protectie la nivel de "subdirector" adica se utilizeaza o *parola* asociata pentru toate fisierele existente in acel subdirector.

Utilizand o singura *parola* pentru fiecare fisier se creaza o protectie de tipul "totul sau nimic" Adica daca utilizatorul cunoaste *parola* atunci el are acces deplin la fisier iar daca nu o cunoaste nu -i este permis nici-un tip de acces.

Acest tip de protectie nu permite sa se asigure un acces "nauntat" pe tipuri de acces permis anumitor utilizatori.

7.4.3.Protectia prin *liste de acces.*

O alta abordare a problemei protectiei fisierelor este prin "filtararea" accesului in functie de identitatea utilizatorului. Diferiti utilizatori pot avea nevoie de diferite drepturi de acces la fisiere si directoare. SO asociaza o *lista de acces* la fiecare fisier si director. In aceasta lista se specifica numele utilizatorului si tipul de acces permis pentru fiecare utilizator al SC.
Atunci cand un utilizator solicita un acces la un anume fisier SO verifica lista de acces asociata cu fisierul. Daca accesul soliciatat este in lista atunci utilizatorului i se permite accesul cerut la fisier. Daca accesul solicitat nu se afla in *lista de acces* la intrarea corespunzatoare utilizatorului atunci aceesul nu este permis.

7.4.4.Grupuri de acces.

Principala problema in cazul utilizarii *listelor de acces* o reprezinta lungimea acestor liste. Daca toti utilizatorii au drept de acces in citire atunci in lista apar toti utilizatorii cu acest drept la citire. Tehnica *listelor de acces* are 2 consecinte nedorite:

constructia *listelor de acces* poate complicata daca nu se cunoaste in avans lista utilizatorilor SC.

directorul va avea lungime variabila dinamica ceeace complica administrarea sa.

Aceste probleme se pot rezolvate utilizandu-se o forma condensata a listelor de acces prin gruparea (clasificarea) utilizatorilor si stabilirea unor drepturi de acces pentru fiecare din aceste grupe raportate la fiecare fisier.
In raport cu un fisier un utilizator se incadreaza in una din urmatoarele grupe:

Proprietar (Owner). Utilizatorul care a creat fisierul este *proprietar.*

Grup. Este clasa de utilizatori care folosesc in comun fisierul si care este necesar sa aiba aceleasi drepturi de acces. Aceasta clasa de utilizatori se mai numeste si "grup de lucru".

Universal sau "altii" (Others). Este multimea utilizatorilor din sistem care nu au legatura cu un anume fisier si care nu au drepturi de acces la acel fisier.

Cu o asemenea impartire in clase a utilizatorilor vor fi necesare numai 3 campuri pentru a defini tipul de acces permis unui utilizator la un fisier.
In sistemul UNIX de exemplu, se utilizeaza un asemnea mecanism de protectie stabilind pentru fiecare grupa de utilizatori permisiunea de a citi (r), de scrie (w), de a executa un fisier (x) . Astfel cu o structura de 9 biti asociata fiecarui fisier se pot stabili pentru fiecare clasa de utilizatori drepturile de acces la fisierul respectiv.