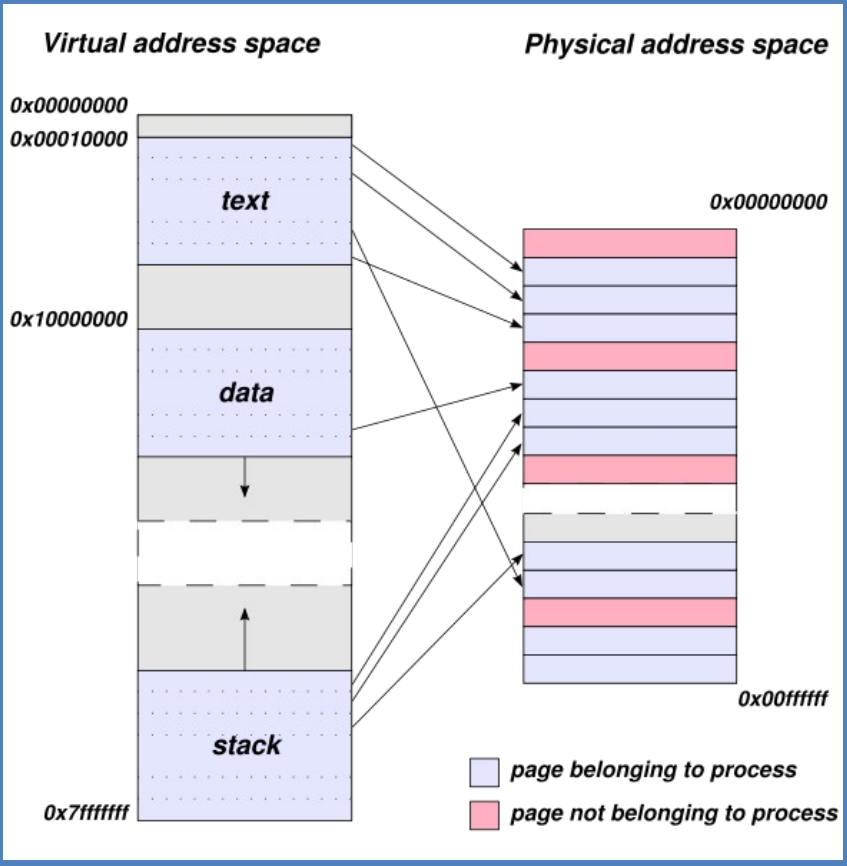
Implementarea gestiunii memoriei

Memoria virtuala

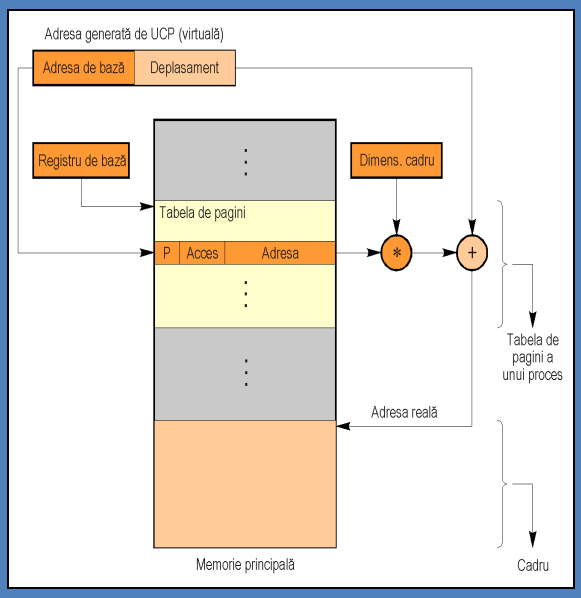
Memoria virtuală reprezintă o tehnică de organizare a memoriei unui calculator prin intermediul căreia se extinde spatiul de adrese accesat dinamic din memoria RAM prin o adresare in memoria disponibila de pe Hard Diskul atasat dandu-i totodata un aspect continuu chiar daca acesta e format din segmente diferite din RAM si Hard Disk.

Avantajele care reies din impelementarea acestui tip de memorie si a unui system de management sunt:

- Implementat în hardware cu suport din partea SO – permite astfel un management software pentru un mecanism hardware rapid ceea ce face ca accesul CPU – RAM si RAM – Hard Disk sa aiba timpi de intarziere neglijabili
- Permite ca procesele sa aiba spatii de adrese distincte – evitand astfel problemele de suprascriere intre datele accesate de procese concurente si adiacente in memorie
- Permite ca spatiul de adrese al procesului sa fie mai mare decât memoria fizic prezenta – reusind astfel sa permita rularea a mai multor procese, daca e necesar, in acelasi timp sau a unor procese mai rapid prin oferirea a mai mult spatiu de stocat date necesare acestuia
- Permite ca memoria folosita de proces sa nu fie continua – rezolva problema proceselor care sunt spre sfarsitul memoriei RAM sau care nu au loc de extindere datorita suprapunerii peste altele
- Permite partajarea unor zone de memorie de catre mai multe procese – alocand unor procese atributii de management a memoriei si oferindu-le un rol mai important in rulare



Adresarea memoriei virtuale dupa principiul paginarii



Adresarea virtuala implementata permite astfel o separare intre memoria fizica si adresele la care un program acceseaza si stocheaza date. Cel mai raspandit mecanism de implementare a acestei separari este impartirea memoriei in pagini, lucru ce permite unui system de operare sa foloseasca eficient memoria primara mutand instructiuni si date neutilizate in memoria secundara si apoi aloca spatiul fizic eliberat unui alt proces. Cand procesul original face o referinta la adresa virtuala unde se asteapta ca datele sau instructiunile sa fie, sistemul de operare aloca din nou memoria fizica primara (posibil sa fie nevoie sa mute deasemenea anumite date din memoria primara in memoria secundara) si muta instructiunile sau datele cerute in acea locatie. Este necesara evitarea unei rate de transfer de date intre memoria fizica si cea virtuala, mare deoarece sistemul va functiona cu performante scazute.

Una dintre cele mai importante beneficii ale adresarii virtuale o reprezinta faptul ca fiecare proces are dreptul sa scrie doar in memoria fizica pentru care sistemul de operare a creat o referinta in tabela de pagini folosite. Acest lucru impiedica ca procesele sa isi suprascrie date unele altora. Alte beneficii importante includ abilitatea sistemului de operare de a detecta referinte la adrese invalide, adrese read only sau adrese a caror memorie fizica nu este valida.

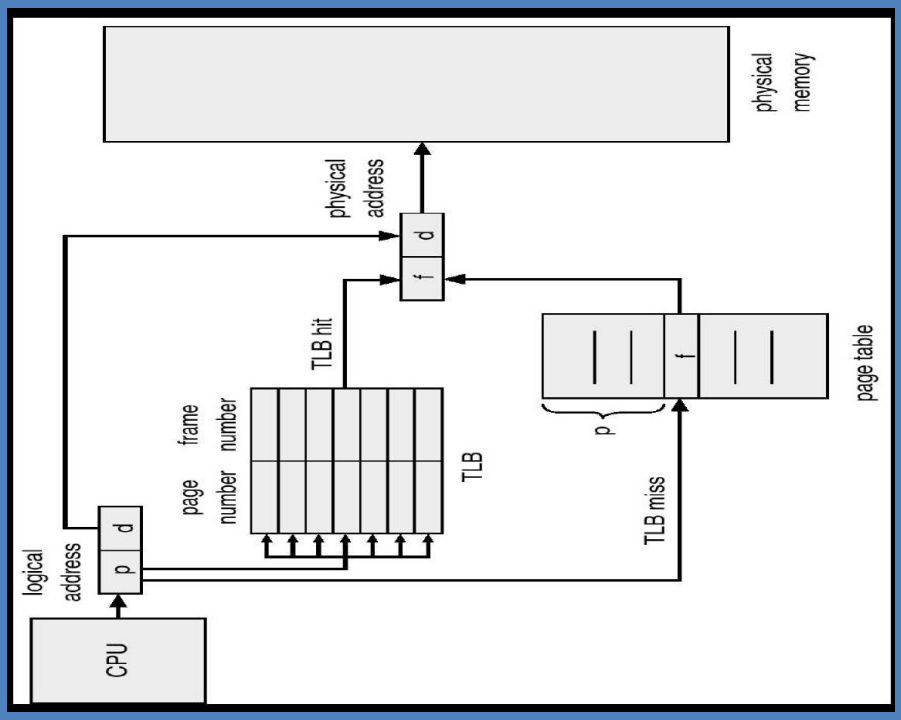
Aproape toate implementarile memoriei paginate folosesc tabele de pagini care asociaza adresele virtuale vazute de aplicatii in adrese fizice (denumite adeseori adrese reale) folosite de partea hardware pentru a procesa instructiuni. Fiecare intrare din tabela de pagini contine partea de inceput a adresei paginii, adresa memoriei fizice la care este localizata pagina sau un indicator ca pagina este intr-un anumit fisier de disc (daca sistemul foloseste fisiere disc pentru a lasa aplicatiile sa foloseasca memorii virtuale de marime mai mare decat memoria reala). Sistemele pot avea o tabela de pagini pentru tot sistemul sau poate avea cate o tabela de pagini separate pentru fiecare aplicatie. Daca este doar una, mai multe aplicatii care ruleaza concomitent impart acelasi spatiu virtual de adrese. Sistemele care folosesc tabele multiple de pagini ofera spatii de adrese virtuale separate, aplicatiile concurente avand impresia ca folosesc aceeasi gama de adrese insa paginile lor separate redirecteaza catre adrese reale diferite.

Arhitecturile procesoarelor curente sunt pe 32 biti, arhitecturi ce permit o adresare de 2^{32} pentru spatial de adrese, putand astfel forma adrese de la 0x00000000 la 0xFFFFFFFF acumuland astfel 4GB, numit si spatial virtual de adresare. Deoarece acestea nu necesita corespondent fizic nu este necesar ca memoria fizica sa fie cel putin egala cu valoarea de 4GB maxim adresabila. Astfel daca avem doar 16MB de memorie toate adresele de peste 0x01000000 vor fi invalide. Insa aproape nici un program nu foloseste cei

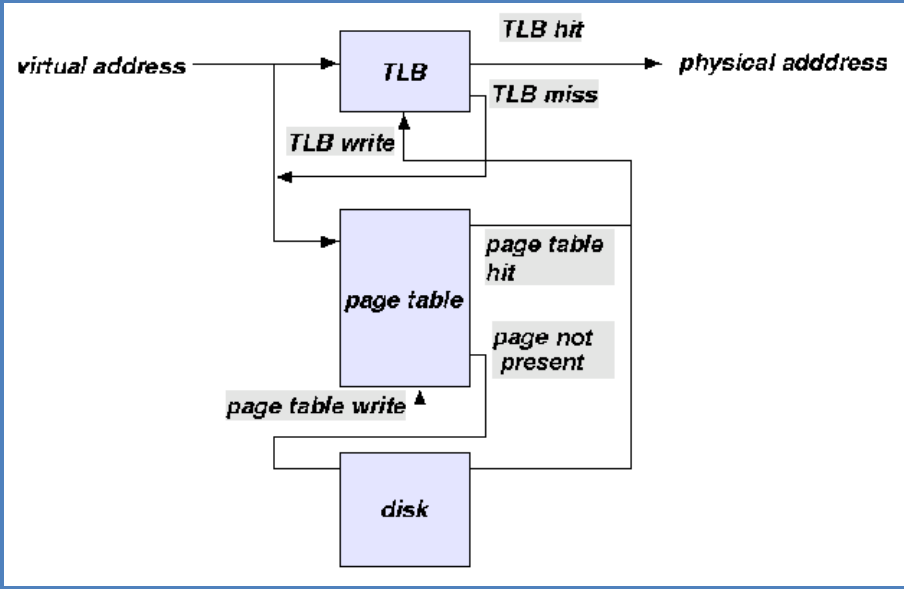
4GB de memorie atunci cand ruleaza ci doar parti ale acesteia. De exemplu un partea de text, date si stive ale unui program, care sunt necesare sa ruleze doar impreuna, ocupa 1MB in total in timpul in care programul ruleaza. Aceste tipuri de bucati poarta numele de bucati speciale. Spatiul de adrese virtual de 4GB este impartit in bucati de acest gen care au deobicei marimea de 4KB denumite pagini. Memoria fizica este si ea impartita deasemenea in bucati, majoritatea de 4KB, denumite frame-uri. Segmentul de text a unui program poate incepe la adresa virtuala 0x00000004 – pagina numarul 0x0 si decalaj de 0x4, insa in realitate aceasta poate corespunde adresei fizice 0xff0e0004 – frame-ul numarul 0xff0e0 si decalaj 0x4. Ceea ce face memoria virtuala este de a converti adresele virtual in adrese fizice si in esenta creeaza legaturile dintre pagini si frame-uri folosindu-se in acest scop tabelele de pagini.

Totodata multe arhitecturi au support hardware direct pentru memoria virtual, cu ajutorul bufferului TLB (Translation Lookaside Buffer) care este plin cu legaturi pagina – frame si in loc de a folosi un system, complet software, de memorie virtuala, atunci cand hardware-ul cauta o adresa unei locatii de memorie si face translatarea pagina – frame , informatiile sunt cache-uite in TLB, lucru ce introduce o crestere a performantei sistemului.

Insa, bufferul TLB poate memora doar un numar fix de mapari page – frame. Este de datoria sistemului complet de memorie virtual de a extinde acest lucru in software si a retine mapari page – frame in plus cu ajutorul folosirii corespunzatoare a tabelelor de pagini.



Presupunand ca un program aflat in derulare incearca sa acceseza memoria de la adresa virtuala 0xd09fbabe, adresa va fi imparitita in doua: 0xd09f este numarul pagini si 0xbabe este offsetul din pagina. Cu ajutorul Unitatii de Management al Memoriei (MMU) adresa este cautata in TLB, buffer care a fost proiectat special ca aceasta cautare sa fie facuta in parallel ca procesul sa fie foarte rapid, si daca a fost gasita o atribuire in TLB (un hit de TLB), numarul frame-ului din memoria fizica este returnat, offsetul inlocuit si accesul la memorie poate continua. Insa daca nu se gaseste o atribuire (un TLB miss) se va incepe cautarea in tabelul de pagini.



Atunci cand mecanismul hardware nu gaseste frame-ul fizic pentru o pagina virtuala, el va genera o intrerupere de processor numita Page Fault. In arhitecturile curente se poate folosi un controller de intreruperi instalat in sistemul de operare care sa solutioneze aceste Page Faults. Controlerul poate verifica maparea adreselor in tabelul de pagini si poate verifica daca ele corespund. Daca sunt o mapare exista legatura este trecuta in TLB iar sistemul va cauta in TLB iar si legatura va fi utila.

Cautarea in tabela de pagini poate fi fara succes din doua motive:

- Nu exista o translatie valida pentru acea adresa, accesul la memoria de la acea adresa este ori, gresit ori invalid si sistemul de operare trebuie sa actioneze pentru a remedia problema. In sistemele modern de operare se va trimite o eroare de segmentare programului in cauza.
- Pagina nu se gaseste in memoria fizica (a fost evacuata in memoria secundara), pagina regasindu-se in alta locatie cum ar fi pe un disc. Pentru a controla acest caz, e nevoie ca pagina sa fie luata de pe disc si pusa in memoria fizica. Cand memoria fizica nu este plina, acesta e un lucru simplu, este necesar sa se copieze pagina ceruta in memoria fizica, sa se modifice intrare din tabela de pagini ca sa indice ca este prezenta in memoria fizica, se scrie maparea in TLB si se reincepe instructiunea. Daca memoria fizica este plina, si nu sunt frame-uri libere, pagini din memoria fizica ar putea fi necesar sa fie schimbate cu paginile necesare. Tabela de pagini trebuie sa fie updatata ca sa fie marcate paginile care au fost inainte in memoria fizica si nu mai sunt, si sa se marcheze si cele care au fost pe disc si mutate in memoria fizica si totodata trebuie updatat si TLB-ul si restartata instructiunea.

Sistemele care folosesc tabele de pagini deobicei folosesc o tabela cu frame-uri si una cu pagini.

Tabelul de pagini, in implementarea cea mai simpla, detine informatii despre care frame-uri sunt mapate. In sisteme mai avansate, tabela de frame-uri poate sa contina si informatii despre spatial de adrese la care o pagina la care o adresa se afla, sau informatii statistice sau alte informatii istorice.

Pagina de tabele contine maparea intre adresa virtual a unei pagini si adresa fizica a unui frame. Exista deasemenea si informatii auxiliare despre pagina cum ar fi bitul de prezenta, bitul dirty sau modificat, spatiu de adresa si informatii despre ID-ul procesului insa aceste informatii nu sunt exacte deoarece orcine le poate modifica. Mediile de stocare secundare cum ar fi un hard disk pot fi folosite pentru a extinde memoria fizica. Paginile se pot introduce si scoate din memoria fizica si disc. Bitul de prezenta poate indica care pagini sunt in prezent in memoria fizica sau disc si poate indica cum sa fie tratate daca sa fie aduse in memoria fizica sau puse pe disc. Bitul dirty permite optimizarea performantei. Sa presupunem ca avem o pagina care necesita sa fie mutate in memoria fizica. Putem sa scriem in aceasta pagina sau doar sa citim din ea. Daca doar citim si avem nevoie sa o inlocuim cu alta nu mai este nevoie sa o scriem pe disc

deoarece nu a fost modificata. Insa daca am scris in pagina vom pune fanionul dirty si asta va insemna ca va fi nevoie sa scriem pagina inapoi pe disc ca sa avem informatia updatata. Informatiile despre spatiul de adrese sau ID-ul procesului sunt necesare pentru ca sistemul de management al memoriei virtuale sa stie ce pagini sa asocieze fiecarui process. Deoarece harta memoriei virtuale este aceeasi pentru fiecare process, intre doua procese, doua adrese virtuale identice pot fi folosite in scopuri diferite, si deci adresele trebuiesc intr-un fel diferite identificandu-le cu cu procesul in tabela de pagini. Acest lucru poate fi facut utilizand un identificator de spatiu de adrese unic sau ID-ul procesului.

Toate sistemele de memorie virtual au arii de memorii care sunt blocate si deci nu pot fi scoase spre medii de stocare secundare:

- Mecanismele de intreruperi se bazeaza in general pe un vectori de pointeri catre controlerele de diferite tipuri de intreruperi(sfarsitul I/O, mesaje de timer, erori ale programului, erori de pagina) si e convenabil ca intreruperile sa fie servite fara schimburi de pagini.
- Tabelele depagini sunt deobicei ne paginabile
- Bufferele de date care sunt accesate in afara CPU, de exemplu de la periferice care folosesc accesul direct la memorie(DMA) sau de care folosesc canale de I/O. Deobicei astfel de dispozitive si bufferele de care sunt atasate folosesc adrese de memorie fizica in locul adreselor virtuale de memorie.
- Orice alt Kernel sau arii de aplicatii a caror operativitate este foarte dependent de timp si nu poate fi acceptata variatia timpului de raspuns

In proiectarea sistemelor de memorie paginata trebuie luate in considerare cateva aspecte:

- Marimea paginilor versus marimea tabelei de pagini – Un system cu pagini de marime mica foloseste mai multe pagini deci necesita o tabela de pagini care ocupa mai mult spatiu si este mai lenta. De exemplu, daca un spatiu de adrese virtuale de 2^{32} este mapata pe pagini de 4KB (2^{12} biti), numarul de pagini virtual este 2^{20}. Insa, daca marimea paginii este marita la 32KB (2^{15} biti), doar 2^{17} pagini sunt necesare.
- Marimea paginii versus folosirea de TLB – procesoarele au nevoie sa detina un buffer TLB, mapand adrese virtuale spre fizice, care sunt verificate la fiecare acces de memorie. Bufferul TLB are totusi marime limitata, si cand nu poate fi satisfacuta o cerere (TLB miss) tabelele de pagini trebuie cautate manual(fie in hardware sau software, depizand de arhitectura) pentru o mapare corecta, un process ce consuma timp. Pagini mai mari inseamna ca un cache TLB de aceeasi marime poate accessa parti de memorie mai mari care evita miss-urile de TLB costisitoare.
- Fragmentarea interna a paginilor – procesele rareori au nevoie de un numar exact de pagini. Astfel ultima pagina deseori va fi doar partial plina, irosindu-se memorie. Paginile mari cresc probabilitatea de memorie irosita astfel deoarece se incarca multe pagini cu memorie nefolosita. Paginile mici asigura o folosire mai buna a memoriei. De exemplu presupunand ca marimea paginii este de 1MB, daca un process aloca 1025KB, doua pagini terbuie folosite, rezultand in 1023KB de spatiu nefolosit.
- Marimea paginii versus accesul la disc – cand se transfera de pe disc, majoritatea intarzierilor se datoreaza timpului de cautare. Din aceasta cauza, transferuri secventiale mari sunt mult mai eficiente decat transferul de date mai mici. Transferul de pagini mari de la disc la memorie nu necesita mai mult timp decat pentru paginile mici.
- Determinarea marimii paginii intr-un program – majoritatea sistemelor de operare permit programelor sa determine marimea paginii in timp ce sunt rulate. Acest lucru permite programelor sa foloseasca memoria mai efficient aliniind alocarile si reducand fragmentarea interna a paginilor.

Alocarea memoriei de catre compilatoarele sub Linux si Windows

Sistemele de tip UNIX si POSIX utilizeaza functia de system sysconf(), cum este ilustrat mai jos folosind un program scris in limbajul C

```
#include <stdio.h>
#include <unistd.h>    // sysconf(3)

int main()
```

```
{
	printf("The page size for this system is %ld bytes\n", sysconf(_SC_PAGESIZE)); //_SC_PAGE_SIZE is OK too.
	return 0;
}
```

Sistemele de operare bazate pe Win32 cum ar fi Windows 9x, NT, ReactOS, utilizeaza functia de system GetSystemInfo() din kernel32.dll.

```
#include <stdio.h>
#include <windows.h>

int main()
{
	SYSTEM_INFO si;

	GetSystemInfo(&si);
	printf("The page size for this system is %u bytes\n", si.dwPageSize);

	return 0;
}
```

Mecanismul memoriei virtuale a fost conceput in mod special pentru a permite executia in paralel a mai multor procese al caror necesar de memorie este mai mare decat disponibilul de memorie fizica. Astfel, la aducerea unei noi pagini in memorie, este deci posibil (si chiar probabil) ca toata memoria fizica sa fie ocupata. Deoarece insa respectiva pagina de memorie este strict necesara, va trebui sa i se faca loc. Altfel spus, va trebui eliminata din memorie o alta pagina, care va fi salvata pe disc. Teoretic, orice pagina poate fi eliminata din memoria fizica. In practica insa, modul in care se alege pagina ce va fi salvata pe disc influenteaza performantele sistemului, deoarece mai devreme sau mai tarziu acea pagina va trebui adusa din nou in memoria fizica. Ceea ce se urmareste este minimizarea numarului de inlocuiri de pagini din memoria fizica, operatie care consuma mult timp. In continuare, vor fi prezentate cateva politici de inlocuire a paginilor din memoria fizica.

- **Algoritmul FIFO (First In, First Out)** -> se inlocuieste pagina care a stat in memorie cel mai mare interval de timp. S.O. mentine o lista cu paginile din memorie, noile pagini sunt adaugate la sfarsitul listei si se inlocuieste pagina din capul listei (pagina cea mai veche).

  Astfel, pentru utilizarea acestei metode se creaza o lista a paginilor fizice, in ordinea incarcarii lor. Lista se actualizeaza la fiecare incarcare: pagina virtuala este incarcata in pagina de pe prima pozitie din lista, pozitie care este apoi inlaturata din lista, pentru a fi adaugata pe ultima pozitie din lista. Practic, se poate parcurge circular aceeasi lista, modificand doar pozitia pointerului in lista. Bitul M asociat fiecarei pagini poate indica daca pagina fizica a fost modificata si trebuie salvata, inainte de a fi reacoperita.

  Avantajul acestei strategii de inlocuire il constituie faptul ca este simplu de implementat:

  - Fiecarui bloc i se asociaza un contor in lista spatiilor ocupate
  - La transferul unui bloc in memoria M_1, contoarele sunt actualizate

  FIFO prezinta insa si un dezavantaj: nu ia in considerare principiul localitatii referintelor:

  - Poate mari in mod semnificativ timpul necesar pentru executia unui proces
  - Poate inlocui cu aceeasi probabilitate blocuri utilizate intens si blocuri utilizate rar

- **Algoritmul optimal al lui Belady (MIN)** –> se inlocuieste pagina ce va fi referita cel mai tarziu, relativ la momentul de timp curent

Aceasta metoda presupune astfel ca, de fiecare data cand trebuie eliminata o pagina dintre cele existente in acel moment in memoria fizica, sa fie aleasa acea pagina care va fi accesata cel mai tarziu, in viitor. Se poate demonstra ca, in acest fel, se realizeaza numarul minim posibil de inlocuiri de pagini (intre memoria fizica si fisierul de paginare). Din pacate, aceasta metoda are o importanta mai mult teoretica, deoarece este practic imposibil sa cunoastem asemenea informatii despre accesarea in viitor a paginilor de memorie. Din acest motiv s-au dezvoltat alti algoritmi care incearca sa aproximeze cat mai bine metoda optimala, prevazand modul de accesare in viitor al paginilor de memorie prin analiza modului in care s-a desfasurat aceasta in trecut. Algoritmul optim este astfel un algoritm imposibil de implementat. Fiecare din pagini ar trebui marcata cu numarul de instructiuni de executat pana la momentul referirii sale. Se va inlocui pagina care are asociata cel mai mare numar de instructiuni. Acest algoritm este insa utilizat pentru a se compara rezultatele cu diversi algoritmi de inlocuire a paginilor si poate fi implementat prin efectuarea a doua treceri peste programul de executat: intai se ruleaza programul si se observa referirea paginilor la fiecare instructiune, iar a doua rulare foloseste informatiile primei rulari si se numara page fault-urile. Un page-fault apare numai daca se refera o pagina si ea nu este prezenta in memorie.

- **Algoritmul LRU (Least Recently Used)** –> se inlocuieste pagina cea mai putin utilizata, relativ la momentul de timp curent

Este cea mai folosita politica de inlocuire, bazata pe urmatoarea observatie practica: un program care a accesat o adresa de memorie va accesa, in viitorul apropiat, foarte probabil, adrese de memorie aflate in vecinatatea primei adrese.Aceasta observatie, numita principiul localizarii, are doua aspecte:

- ✓ daca un proces a executat o instructiune, este foarte probabil ca, in viitorul apropiat, o va executa din nou, impreuna cu instructiunile din jurul sau, de mai multe ori, deoarece programele constau, in principal, din bucle;
- ✓ daca un program a accesat o variabila, este probabil ca, in viitorul apropiat, o va accesa din nou (eventual impreuna cu variabilele invecinate - de exemplu in cazul parcurgerii unui tablou)

Ideea algoritmului **LRU** este urmatoarea: va fi eliminata de fiecare data pagina care nu a mai fost accesata de un timp mai lung decat toate celelalte pagini din memoria fizica. Din nou poate fi necesar un suport hardware pentru a masura timpul cand a fost accesata ultima data o pagina de memorie. Pentru majoritatea aplicatiilor, LRU da cele mai bune rezultate. Un alt avantaj il constituie faptul ca evita inlocuirea blocurilor incarcate de mai mult timp, dar frecvent utilizate, ca in cazul strategiei FIFO. Exista insa anumite cazuri particulare in care performantele LRU sunt foarte slabe.

Dezavantajul acestei politici de inlocuire este acela ca este mai dificil de implementat:

- Se asociaza un contor cu fiecare bloc din M_1
- La fiecare referire a unui bloc, contorul acestuia este setat la o valoare pozitiva
- La intervale fixe de timp, contoarele tuturor blocurilor sunt decrementate
- Se inlocuieste blocul al carui contor contine valoarea cea mai mica

- **Algoritmul LFU (Least Frequently Used) ->** se inlocuieste pagina care a fost referita cel mai putin frecvent.

Este un algoritm asemanator cu LRU, bazat pe aceeasi idee. In acest caz, pentru fiecare pagina din memoria fizica se retine numarul de accese efectuate in ultimul interval de timp (de lungime fixata). De fiecare data este eliminata pagina care a avut cel mai mic numar de accesari in acest ultim interval. Performantele sale sunt de asemenea dintre cele mai bune.

- **Algoritmul Second Chance** -> modifica FIFO astfel incat sa tina cont de bitul R (bitul Referenced -> daca acest bit nu este disponibil hardware, el poate fi determinat, estimat, software: se mapeaza paginile ca fiind inaccesibile, iar la page-fault pagina se marcheaza read-only si bitul R pe 1).

Algoritmul consta in inspectarea paginii din capul listei:

- ➔ Daca R este 0 atunci aceasta pagina este selectata pentru inlocuire
- ➔ Daca R este 1 atunci R este setat pe 0 iar pagina este mutata la coada listei
- ➔ Se continua inspectarea noii pagini din capul listei

- **Algoritmul CLCK** -> varianta a algoritmului FIFO, ce aproximeaza algoritmul LRU

Algoritmul ceasului este similar cu Second Chance. De aceasta data paginile sunt tinute intr-o lista circulara, iar la un page-fault este analizata pagina din dreptul bratului. Daca R este 0, pagina este inlocuita. Daca R este 1 atunci R este setat pe 0 si se avanseaza pozitia bratului.

- **Al goritmul Random (RAND)** –> se alege in mod aleator o pagina care va fi inlocuita.

Se selecteaza astfel, la intamplare, o pagina dintre cele aflate in memoria fizica pentru a fi salvata pe disc. O asemenea abordare, evident, nu ofera performante satisfacatoare deoarece este posibil ca o pagina astfel eliminata sa fie accesata foarte curand in viitor.

- **Algoritmul Working Set** -> La fiecare intrerupere de ceas se curata bitul R, iar la un page-fault se scaneaza paginile. Astfel, daca pagina are R setat pe 1 se pune intr-un camp (time of last use) asociat paginii timpul virtual al procesului. Daca R este 0 si age = (timpul virtual curent – time of last use) > T se selecteaza pagina pentru evacuare. Daca R este 0 si age = (timpul virtual curent – time of last use) < T, pagina nu se evacueaza decat daca toate paginile sunt in aceeasi situatie, iar pagina are cel mai mic time of last use. Astfel, acest algoritm mentine in memorie acele pagini ale fiecarui proces, care au fost referite la un interval de timp (prestabilit) in urma. Daca nu exista loc in memorie, o parte din procese vor fi dezactivate.

- **Algoritmul NRU** (Not Recently Used) -> se inlocuieste o pagina care nu a fost utilizata recent. Pentru a implementa aceasta metoda, se asociaza fiecarei pagini fizice doi biti:

  - bitul R, numit bit de referire, primeste valoarea 0 la incarcarea paginii virtuale in pagina fizica si primeste valoarea 1 la fiecare acces realizat la aceasta pagina. Periodic, toti bitii R sunt setati pe 0
  - bitul M, bitul de modificare, primeste valoarea 0 la incarcarea paginii si valoarea 1 la fiecare scriere in acea pagina.

  Utilizarea acestor doi biti determina o impartire a paginilor fizice in patru categorii:

  - categoria 1: pagini nereferite si nemodificate, cu R=0 si M=0
  - categoria 2: pagini referite si nemodificate, cu R=1 si M=0
  - categoria 3: pagini referite si modificate, cu R=1 si M=1
  - categoria 4: pagini nereferite, in ultimul interval de timp, dar modificate, cu R=0 si M=1.

Pentru a inlocui o pagina conform metodei NRU, se alege o pagina din una din categorii, in ordinea: 1, 4, 2, 3. Paginile fizice din categoria 3 sau 4 vor fi salvate in paginile virtuale corespunzatoare, inainte de a fi eliberate. Acest algoritm are avantajul de a fi simplu si eficient.

- **Algoritmul Last In First Out (LIFO)** –> se inlocuieste pagina care a stat in memorie cel mai putin timp

Algoritmii LRU, LFU, LIFO, FIFO si RAND sunt algoritmi realizabili. Algoritmul MIN este nerealizabil practic, dar aste utilizat pentru comparatie cu ceilalti algoritmi (ca referinta deoarece este algoritmul optimal). Exista si algoritmi complecsi de alocare, in acesti algoritmi regula de reamplasare al

paginilor aplicandu-se la nivel global, pentru toata memoria principala, fara a identifica procesul care utilizeaza pagina.

Tot pentru minimizarea numarului de inlocuiri de pagini din memoria fizica, deci a numarului de accese la hard-disk, sistemele de operare folosesc, de obicei, paginarea la cerere: se elimina o pagina din memoria fizica, conform cu oricare dintre politicile descrise mai sus (sau altele, care n-au fost prezentate aici), numai atunci cand trebuie adusa o noua pagina de pe disc si nu mai este loc disponibil.

Totusi, o nuantare a acestei abordari poate imbunatati performantele. Deoarece este mai rapid transferul unei cantitati mari de date spre/dinspre discul hard, intr-un singur pas, decat transferul aceleiasi cantitati de date in mai multe etape, unele politici de inlocuire realizeaza eliminarea din memoria fizica a mai multor pagini simultan, lasand mai mult loc liber in memorie, in loc sa salveze pe rand fiecare pagina pe disc, atunci cand este neaparat necesar.

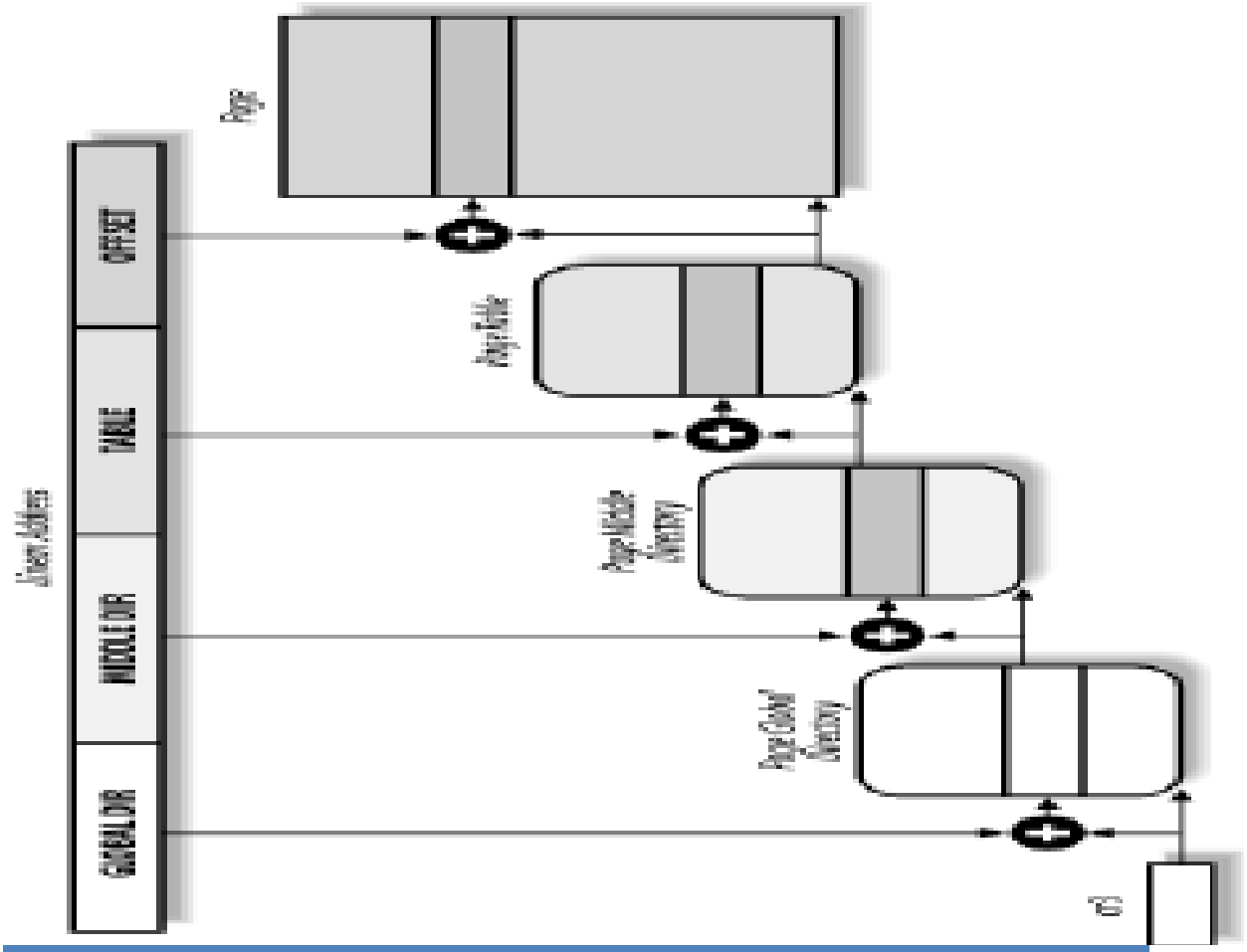
Se pot trage astfel urmatoarele concluzii referitoare la algoritmii de inlocuire a paginilor:

- Algoritmul Optimal = neimplementabil, dar folositor pentru testarea altor algortimi
- Algoritmul FIFO = poate sa inlocuiasca pagini importante
- Algoritmul Second Chance = imbunatatire a FIFO-ului
- Algoritmul Ceasului = mai eficient de implementat decat FIFO
- Algoritmul LRU = algoritm excelent, dar care este dificil de implementat fara hardware
- Working Set = algoritm bun, dar costisitor

In Linux, se folosesc trei nivele logice ptr paginare:

- Page Global Directory (PGD)
- Page Middle Directory (PMD)
- Page Table (PTE)

Pentru arhitecturile cu doar dou nivele de paginare, PMD-ul este eliminat la compilare.



Evacuarea paginilor in Linux

- **Algoritm de tip LRU**
  - se mentin doua liste: una cu paginile active, si una cu paginile inactive
  - atunci cand sistemul are nevoie de pagini, se elibereaza pagini, in ordine, din:
    - diversele cache-uri: buffer cache, dcache, icache
    - lista de pagini inactive
    - lista de pagini active
- **kswapd**
  - kernel thread care evacueaza paginile
  - este activat atunci cand numarul de pagini libere scade sub o limita

Gestiunea memoriei

Subsistemul gestiunii memoriei este una din cele mai importante componente ale unui sistem de operare. Inca de la inceputurile sistemelor de calcul a aparut nevoia de a utliza mai multa memorie decat exista fizic. Strategii au fost dezvoltate pentru a depasi aceasta limitare si cea mai de succes este memoria virtuala. Memoria virtuala face in asa fel incat sistemul sa para ca are mai multa memorie decat exista fizic prin impartirea ei in timpul derularii proceselor in functie de necesitatile fiecaruia

Alocarea si dealocarea paginilor de memorie.

Intr-un sistem exista mai multe cereri asupra paginilor fizice. De exemplu, cand o imagine este incarcata in memorie sistemul de operare trebuie sa aloce pagini. Aceste vor fi eliberate in momentul in care imaginea nu mai este procesata si este descarcata. O alta utilizare pentru paginile fizice este este de a pastra date specifice kernel-ului cum ar fi tabelul propriu de pagini. Mecanismul si structurile de date folosite pentru alocarea si eliberarea memoriei sunt probabil cele mai critice elemente in mentinerea eficientei memriei virtuale.

Toate paginile fizice din sistem sunt descrise de structura *mem_map* care este o lista de structuri *mem_map_t*** care se initializeaza la fiecare boot-are. Fiecare element *mem_map_t* descrie o singura adresa fizica in sistem. Campuri importante (in ceea ce priveste gestionarea memoriei) avem:

Count

Reprezinta o contorizare a numarului de utilizari al paginii respective. Aceste contor este mai mare de 1 atunci cand o pagina este impartita de mai multe procese.

Age

Acest camp descrie varsta unei pagini si este folositor pentru a decide daca o pagina va fi folosita pentru eliberare sau pentru schimb.

Map_nr

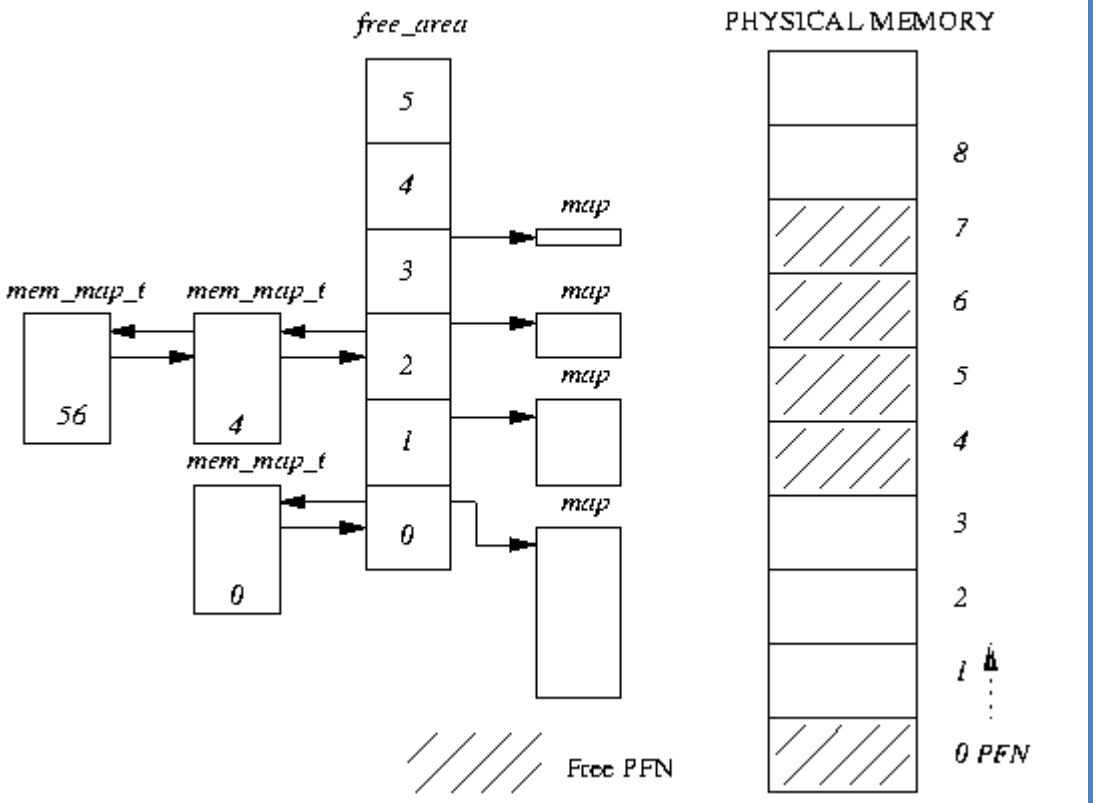
Reprezinta numarul cadrului memoriei fizice descrisa de mem_map_t

Vectorul *free_area* este folosit de codul alocarii paginilor pentru a gasi si elibera pagini. Intreaga schema tampon este suportata de acest mecanism si, atat timp cat ne intereseaza codul, marimea paginii si mecanismul paginarii fizice sunt irelevante.
Fiecare element free_area contine informatii despre blocuri de pagini. Primul element din sir descri pagini singulare, al doilea blocuri de cate 2 pagini, urmatorul blocuri de cate 4 pagini si tot asa multipli putere ai lui 2. Elementul *list* este folosit ca un cap de tabel si are indicatori catre structurile *page* din sirul *mem_map*. Aici sunt stocate blocurile de pagini libere. *Map* este un indicator catre o „harta a bitilor” care pastreaza urma si marimea grupurilor de pagini alocate. Bitul N din harta bitilor este setat daca al N-ulea bloc este liber.

Alocarea paginilor

Linux foloseste algoritmul buddy_2 pentru a aloca efectiv si dealoca blocuri de pagini. Codul alocarii paginilor incearca sa aloce un bloc format din una sau mai multe pagini. Paginile sunt alocate in blocuri puteri ale lui 2. Asta inseamna ca se pot aloca blocuri de 1 pagina, 2 pagini, 4 pagini si asa mai departe. Atat timp cat in memorie existe destule pagini libere pentru a acorda aceasta cerere (nr_free_pages > min_free_pages) codul de alocare va cauta in *free_area* un bloc egal ca si marime cu cel solicitat. Fiecare element din *free_area* contine o harta a blocurilor libere si alocate pentru o anumita marime. De exemplu, al 2-lea element al sirului contine o harta de memorie ce descrie blocurile libere si cele alocate avand lungimea de 4 pagini
Algoritmul de alocare cauta la inceput un bloc de pagini egal ca si marime cu cel solicitat. Urmareste lantul de pagini libere care sunt stocate in elementul *list* din structura *free_area* . Daca nu exista blocuri libere de marimea solicitata atunci blocuri de urmatoarea marime(care sunt duble decat cele solicitate) vor fi cautate.

Acest proces va continua pana cand intreaga structura *free_area* va fi scanata ori pana cand se va gasi un bloc potrivit. Daca blocul de pagini gasit este de o marime mai mare decat cea solicitata atunci acesta va fi spart in blocuri de marimi mai mici pana cand se ajunge la marimea potrivita. Datorita faptului ca marimile blocurilor sunt puteri ale lui 2 procesul de impartire este simplu deoarece se injumtateste marimea de fiecare data. Blocul liber este stocat in cea mai apropiata lista iar blocul alocat este returnat chematorului.



De exemplu, in figura de mai sus daca un bloc de 2 pagini este solicitat, primul bloc de 4 pagini (incepand cu numarul de cadru 4) ar fi spart in 2 blocuri a cate 2 pagini. Primul, incepand la numarul de cadru 4 ar fi returnat chematorului iar cel de-al doilea, incepand cu numarul de cadru 6 ar fi stocat ca si bloc de 2 pagini in elementul 1 din sirul *free_area*.

Dealocare paginilor

Alocand blocuri de memorie tendinta este sa se fragmenteze memoria si sa se imparta blocurile mari de memorie in blocuri mai mici. Codul de dealocare recombina paginile in blocuri mai mari de pagini libere de fiecare data cand este posibil. De fapt marimea uui bloc este importanta deoarece permite gruparea facila a blocurilor in blocuri mai mari.
De fiecare data cand un bloc de memorie este eliberat, blocul adiacent sau prieten de aceeasi marime este verificat daca este liber. Daca este liber atunci este combinaat cu proaspatul bloc eliberat pentru a forma un nou bloc liber de pagini avand urmatorul ordin de marime. De fiecare data cand 2 blocuri de aceeasi marime se combina intr-un bloc mai mare de pagini libere codul de dealocare incearca iarasi sa combine noul bloc format cu un altul de aceasi marime. In acest fel blocurile de memorie libera vor fi sunt atat de mari cat permite gradul de utilizare al memoriei.

De exemplu in figura de mai sus daca elementul cu numarul de cadru 1 s-ar elibera atunci el s-ar combina cu elementul liber 0 si s-ar stoca in lista 1 din *free_area* ca si un bloc liber de 2 pagini.

Bibliografie :

http://www.ibm.com/developerworks/linux/library/l-memmod/
http://www.ibm.com/developerworks/linux/library/l-memory/
http://www.ibm.com/developerworks/linux/library/l-mem26/
http://cs.pub.ro/~so/Cursuri/curs05/memorie.pdf
http://www.cs.utexas.edu/users/lorenzo/corsi/cs372/03F/notes/11-20.pdf
http://people.msoe.edu/~mccrawt/resume/papers/CS384/mccrawt_cs384_virtual.pdf
http://courses.cs.vt.edu/csonline/OS/Lessons/VirtualMemory/index.html
http://courses.cs.vt.edu/%7Ecs3204/fall2007/gback/index.html
http://www.linux-tutorial.info/modules.php?name=MContent&pageid=308
http://en.wikipedia.org/wiki/Comparison_of_Windows_and_Linux
http://en.wikipedia.org/wiki/Linux_kernel