GESTIUNEA MEMORIEI

Gestiunea resurselor memoriei este un aspect complex al unui sistem de operare.Iată câţiva paşi parcurşi înspre o organizare eficientă şi performantă.

Cuprins

4. Algoritmi de inlocuire a paginilor ***Datcu Octaviana***

4.1.Introducere
4.2.Strategii de înlocuire a paginilor – algoritmi
4.2.1. Modelul înlocuirii optimale a paginilor (OPRA)
4.2.2. Not Recently Used (Neutilizată recent)
4.2.3. First In, First Out (primul intrat, primul ieşit)
4.2.4. Least Recently Used (Cel mai recent utilizat)
4.2.5. Implementarea cu stivă a LRU
4.2.6.Algoritmi de aproximare LRU
4.2.6.1. Second Chance (A Doua Şansă)
4.2.6.2. Clock (Ceasul)
4.2.6.3. Not frequently used (Neutilizată frecvent)
4.2.9.Aging (Îmbătrânirea)
4.2.7. Random
4.3. Impactul dimensiunii paginii asupra performanţelor memoriei virtuale
4.4.Comportamentul programului în cazul paginări
4.5. Working Set (Setul de lucru)
4.6. Fenomenul “thrashing”
4.7. PFF (Frecvenţa de eroare de pagină)
4.8. Alţi algoritmi
4.8.1. “Low inter-reference Recency Set Replacement Policy” LIRS
4.8.2. “Enhanced second chance/clock”
4.8.3. “Page Buffering”
4.9. Concluzii asupra performanţei principalilor algoritmi
4.10. Caracteristici ale paginării în Windows XP/ Linux
4.11. Referinţe

5. Modelarea algoritmilor de inlocuire a paginilor ***Velican Valentin***

5.1.Definitie.Scurt “istoric”.
5.2.Înlocuirea locală vs. Înlocuirea globală
5.3.Algoritmul optim de înlocuire a paginilor. Imposibilitatea realizării sale.
5.4.Algoritmi de înlocuire. Tipuri. Discuţie.
5.5.Concluzii
5.6.Rezumat
5.7.Referinte

6. Probleme de proiectare pentru sisteme de paginare ***Stoian Andrei***

6.1. Alocare globala vs alocare locala;
6.2. Controlul alocarii paginilor;
6.3. Dimensiunea paginilor;
6.4. Spatiul datelor si spatiul instructiunilor;
6.5. Pagini partajate;
6.6. Eliberarea (curatarea) paginilor;
6.7. Interfata memoriei virtuale;

7. Probleme de implementare ***Nistor Adrian***

7.1. Implicatiile Sistemului de Operare in paginare;
7.2.Gestionarea erorilor de paginare;
7.3. Backupul instructiunilor;
7.4. Blocarea paginilor in memorie;
7.5. Termenul de "Backing Store";
7.6. Separarea regulilor si a mecanismului;

8. Gestionarea memoriei în UNIX ***Ledeanu Mihai Silviu***

8.1.Concepte fundamentale
8.2.Implementarea Gestionării memoriei în Unix
8.2.1.Swapping
8.2.2.Paginarea
8.2.3.Algoritmul de înlocuire a paginilor
8.3. Rezumat

1 Elemente de baza in gestiunea memoriei ***Neculoiu Paul***

“640K ought to be enough for anybody.” – atribuit lui Bill Gates (1981)
Adevarat sau nu citatul in 1981 acest concept nu mai este de ceva timp de actualitate. Nevoile de memorie ale aplicatiilor software din ziua de azi au depasit de mult acea valoare. Desi tehnica a evoluat o data cu necesitatea acestora de memorie totusi exista inca anumite limitari tehnice sau financiare care retin utilizatorul din a atinge utopia memoriei non-volatile infinit de mari si infinit de rapida.
Din acest motiv se pune un mare accent in ziua de azi pe gestionarea cat mai eficient a memoriei limitate aflate la dispozitia utilizatorului.

1.1. Monoprogramarea (mono/single tasking) fara utilizarea de swap sau paginare

Desi tehnica nu mai este demult utilizata in statiile din ziua de azi, inca se mai poate gasi in unele sisteme dedicate sau unitati de calcul portabile (palmtop) si printre primele statii aparute desi tehnicile de implementare pentru fiecare desi acestea difera de la caz la caz.

Primele mainframeuri aveau sistemul de operare la baza memoriei RAM, restul revenindu-i memoriei de program, primele PCuri aparute fiind echipate si cu o memorie ROM rezervata pentru drivere si intrare/iesire cunoscuta sub numele de BIOS (basic input/output system), indtrodus pentru prima data pe scara larga de catre IBM si la acea data era una dintre putinele solutii care presupunea un nivel de abstractizare independent de sistemul de operare si care avea in acelasi timp si calitatea de a fi suficient de bine documentat. Acesta deservea sistemul cu o serie de functii de vaza realiza pasii necesari incarcarii sistemului de operare in RAM si totdata punea la dispozitie unele optiuni de configurare.

Necesitatea de flexibilitate a zilei de azi dar si a programarii defectuase au dus la inlocuirea memoriei ROM ca mediu de stocare in favoare unor memorii EEPROM (sau flash-ROM) aceasta fiind mai usor de actualizat.

Alternativ sistemul de operare se poate pastra intr-o unitate ROM separata de memoria RAM de program, sistem indelung utilizat in palmtopuri si unitatile de calcul embeded. Sistemul de operare era initial “incrustat” si nu putea fi inlocuit decat prin inlocuirea fizica a ROM-ului. In ziua de azi, cu evolutia memoriilor configurabile non-volatile (flash-ROM) este posibila actualizarea continutului ROM-ului respectiv fara inlocuirea fizica a acestuia. In principiu exista doua tehnici de baza pentru a schimba continutul unui flash-ROM in palmtopuri. Prima este referita ca umbrire (“shadowing”) care implica stocarea componentelor actualizabile ale ROM-ului in zona RAM, aceasta avea totusi anumite probleme, componentele respective ocupand RAM-ul respectiv in primul rand, (unele fiind chiar prea mari pentru a incapea in acesta), in al doilea rand daca unitatea ar fi fost vreodata pornita “la rece” (situatie in care RAM-ul se goleste), actualizarile respective s-ar fi pierdut. A doua tehnica utilizata implica reinscriptionarea intregii imagini, utilizand loaderul de boot (boostrap). Acesta intruieste initial unitatea portabila sa incarce anumite componente in ROM cand acesta este pornit (cald sau rece). Un dezavantaj cu aceasta tehnica este aceea ca majoritatea acestor sisteme au spatiu limitat de memorie si nu pot stoca intreaga imagine in timpul reinscriptarii.

Sistemul astfel organizat unitatea de calcul nu poate procesa decat un program intr-un anumit moment de timp, acesta copiindu-l de pe unitatile de stocare in memorie si executandu-l pana la finalizare, moment in care procesarea se opreste, unitatea de calcul asteptand urmatoarea comanda. La primirea acesteia incarca noul program in memorie suprascriindu-l pe cel dintai si executandu-l.

Modul monoprogramarii nu mai este folosit in ziua de azi decat in unele unitati embeded, pentru care costul de productie mai redus si/sau lipsa nevoiei pentru aplicatia respectiva a unui nivel superior de gestionare a justificat renuntarea la modurile mai avansate.

1. 2. Multiprogramarea cu partitii fixe.

Necesarul zilei de azi a dus la adpotarea unor tehnici ce pot rula proceste multiple simultan. Avantajul acestiu sistem este ca procesorul poate prelucra continuu, putand prelua un porgram in timp ce alt program asteapta finalizarea procedurilor de intrare/iesire, crescand, asftfel, eficienta sistemului de calcul fata de modul “clasic” de procesare fiecarui program in parte. Tehnica a fost intotdeauna utilizata in serverele de retea pentru a rula procese multiple (pentru clienti diferiti ce-i drept) simultan, dar in ziua de azi si unitatile client in sine au aceasta capabilitate, monoprogramarea disparand de mult din aceasta zona de prelucrare.

Multiprogramarea este tehnica de exploatare a sistemelor care permite existenta simultana in memoria interna a mai multor programe care se executa concomitent in partitii fixe de memoriecu conditia ca acestea sa nu utilizeze in acelasi timp simultan aceeasi resursa. Executia in multiprogramare a lucrarilor se face pe loturi, fiecare lot de lucrari avand afectata o partitie fixa din memria interna. O partitie este o zoan continua de memorie, de o lungime si adresa fixa, partitii partajate la inceput, de pilda, la pornirea sistemului.

In cadrul fiecarui lot, lucrarile sunt executate secvential, fiind lansate automat in executie. Sub controlul sistemului de operare, unitatea de prelucrare comuta de la o partitie la alta, pentru a realiza executia simultana a proceselor, comutarea facandu-se in momentul in care unitatea nu este utilizata de procesul respectiv (ex. Asteapta terminarea operatiilor de intrare/iesire), comutarea intre procese facandu-se in urma unor evenimente interne proceselor din executie. Fiecare partitie are asociata o prioritate de executi,e resursele sistemului de calcul fiind alocate proceselor conform solicitarilor acestora, si in functie de disponibilitatea reusrselor. In cazul solicitarii unei resurse care nu este disponibila, procesul respectiv intra in aspteptare, pana la eliberarea acesteia. Oridnea de alocare a resurselor intre procesele care solicita aceeasi resursa fiind determinata de prioritatea de partitiei (procesele cu prioritatea mai mare au acces la aceeasi resursa inaintea celor cu prioritatea mai mica). Sistemul de calcul dispune de un sistem de intreruperi prin intermediul caruia se semnaleaza aparitia unui eveniment care poate fi cauza comutarii intre procese.

Accesul programelor la partitii pentru organizarea loturilor devine o problema in sensul ordinii acestora, dimensiunea partitiilor fiind limitata de dimensiunea totala a memoriei, aceasta putand fi divizata in portiuni inegale pentru stocarea programelor. Un algoritm utilizat este asezarea programului la coada pentru cea mai mica partitie care o poate contine. Dezavantajul unei astfel de tehnici consta in momentul in care aplicatiile mici, desi existand spatiu suficient intr-o partitie de dimensiune mai mare pt ea, asteapta loturi multiple pentru a intra in partitia ei corespunzatoare.

O alta strategie ar fi organizarea intrarii pe rand a proceselor astfel in cat acestea sa intre in prima partitie de dimensiuni suficient de mari care devine disponibila. Din nefericire, in cazul sarcinilor mici care intra in partitiile mari, spatiul neutilizat se pierde, acest lucru fiind nedorit. Alternativ se poate cauta intreaga lista de intrare cand o partitie devine libera sa se aleaga cea mai mare care incape, desi in acest mod cele mai mici risca sa nu mai apuce sa intre.

1.3. Modelarea Multiprogramarii

Prin multiprogramare se poate obtine o utilizare crescuta a procesorului. Pe scurt, daca, in medie, un proces este procesat numai 20% din timpul pe care il petrece in memorie, cu cinci procese simultane in memorie, procesorul ar trebui sa fie utilizat in permanenta. Acest model este nerealist de optinist, totusi, deoarece presupune ca toate cele cinci procese nu vor astepta niciodata aceeasi interfata de intrare/iesire.

Un model mai bun il reprezinta privirea utilizarii procesorului dintr-un punct de vedere probabilistic. Presupunand ca un proces utilizeaza o fractiune p din timp asteptand terminarea procesului de intrare/iesire, cu n procese in memorie simultan probabilitatea ca toate cele n procese sa astepte pentru I/O (caz in care procesorul sta) este pn . Utilizarea procesului apoi fiind 1-pn. Deci, daca procesul si-ar petrece 80% din timp asteptand I/O, cel putin zece procese trebuie sa fie in memorie simultan pentru a aduce risipa de procesor sub 10%, procesele interactive asteptand de exemplu ca utilizatorul sa introduca date la un terminal, 80% risipa de procesor nefiind un lucru intocmai iesit din comun.

Pentru acuratete, modelul probabilistic descris este numai o aproximare. Acesta presupune implicit ca cele n procese sunt independente, fiind suficient de acceptabil pentru un sistem cu cinci procese in memorie sa aiba trei ruland si doua in asteptare. Dar cu un singur procesor nu putem rula trei procese simpultan drept urmare un proces devenit pregatit cat timp procesorul este ocupat va trebui sa astepte. Drept urmare procesele nu sunt independente.

De asemenea, presupunand ca o masina de calcul are, spre exemplu, 32MB de memorie, sistemul de operare ocupand 16MB si fiecare program ocupand 4MB. Aceasta ar permite patru programe sa fie in memorie in acelasi timp. Cu o medie de asteptare de 80%, avem o utilizare a procesorului (ignorand necesitatile sistemului de operare) de aproximativ 60%. Adaugarea a inca 16MB de memorie permite sistemului sa treaca de la 4 programe la 8, crescand astfel utilizarea procesorului la 83%. Adaugarea a inca 16MB nu ar creste utilizarea procesorului decat de la 83% la 93%, punand astfel sub semnul intrebarii, utilitatea reala adaugarii repsectivei extinderi de memorie.

1.4. Analiza performantelor sistemelor multiprogramate

Modelul discutat poate de asemenea fi utlizat pentru a analizarea prelucrarii pe loturi. Nu toate procesele utilizeaza procesorul pentru acelasi interval de timp. Cu o medie de asteptare de 80% un proces poate petrece de 5 ori mai mult timp in memorie fata de cat timp, utilizeaza, practic procesorul, chiar si fara concurenta pentru accesul la acesta. In cazul sosirii unei a doua sarcini, utilizarea procesorului creste datorita gradului ridicat de multiprogramare, fiecare proces avand acces jumatate de timp la procesor, pierderea de timp de acces la procesor a primei sarcini fiind redusa. In cazul sosirii unei a 3a sau a 4a sarcini timpul de acces al fiecareia scade iar utilizarea procesorului creste.

1.5. Relocarea si Protectia

Utilizarea multiprogramarii ridica doua probleme esentiale, relocarea si protectia. Sarcini diferite ruleaza de la adrese diferite. Cand programele sunt combinate sa inceapa intr-un singur spatiu de adresa, linkerul trebuie sa stie de la ce adresa va incepe programul in memorie.

Daca prima instructiune ar fi un salt la o procedura la adres absoluta 100, in cazul in care programul e incarcat in prima partitie (la adresa 100K de ex), instructiunea respectiva va sari la adresa 100 care se afla in interiorul sistemului de operare. Ceea ce este necesar este ca aceasta sa sara la 100K+100. Daca programul ar fi incarcat in partitia 2 ar trebui sa sara la 200K+100. Problema adresarii relative la locul de inceput al programului este cunoscuta ca problema relocarii.
O solutie posibila consta in modificarea instructiunilor in sine din program in timp ce acesta este incarcat in memorie. PRogramele incarcate intr-o parttitie li se adauga adresa de inceput a acelei partitii. Performarea relocarii in timpul incarcarii in acest mod, linkerul trebuia sa atribuie programului binar o lista continand care din liniile din program sunt adrese si care sunt comenzi, constante sau alte obiecte care nu trebuie relocate.
Relocarea in timpul incarcarii nu rezolva problema protectiei. Un program malitios poate oricand construi o noua instructiune la care sa sara.

Deoarece programele in acest sistem utilizeaza adrese absolute de memorie in loc de adrese relative la registru nu exista metode de a opri un program din a construi o instructiune care citeste sau scrie orice cuvand din memorie. In sistemele multiuser, nu este de dorit a lasa procesele sa scrie si sa citeasca bucati de memorie apartinand altor utilizatori.

Solutia data de IBM a fost impartirea memoriei in blocuri de 2KB si asignarea acestora a unui cod de protectie de 4 biti pentru fiecare bloc. Acesta bloca orice tentativa a vreunui proces de accesare a memoriei a carui cod de protectie diferea de al lui. De vreme ce numai sistemul de operare putea schimba codurile de protectie procesele utilizatorilor erau prevenite din a interfera cu ale altora si cu sistemul de operare in sine.

O solutie alternativa atat la relocare cat si la problema protectiei o reprezinta echiparea sistemului cu doi registrii hardware speciali denumiti base(baza) si limit(limita). Cand unui proces ii vina randul, registrul de baza este incarcat cu adresa de start a partitiei, iar registrul de limita cu lungimea acesteia. Fiecare adresa de memorie generata automat I se adauga continutul registrului de baza inainte de a fi trimisa in memorie.

Astfel, instructiunea in sine nu mai este modificata. Adresele sunt de asemenea verificate sa nu depaseasca limita, astfel incat sa nu isi paraseasca partitia curenta. Registrele sunt protejate la scriere din partea utilizatorilor.

Dezavantajul acestei scheme este nevoia de a aduna si compara fiecare referinta de memorie. Comparatia se face rapid dar adunarile sunt incete datorita propagarii bitului de carry decat daca nu sunt utilizate circuite speciale de adunare.

CDC 6600 a fost primul supercomputer din lume si utiliza aceasta tehnica. Procesoarele intel 8088 utilizate pentru primele IBM PC foloseau o versiune mai slaba a acesteia, doar registre de baza dar fara cele de limita. Putine calculatoare o mai folosesc in ziua de azi.

1.6. Rezumat

monoprogramarea (mono/single tasking) fara utilizarea de swap sau paginare

Tehnica monoprogramarii inca se mai gaseste in uz in unele sisteme dedicata sau unitati de calcul portabile nemaifiind utilizata in statii de cateva generatii.
Informatiile erau stocate in functie de necesitati fie in RAM-ul temporar sau in memorii mai nevolatile ROM, necesitatile curente inlocuind in timp pe acestea din urma cu variante mai performante.
Sistemul in schimb nu poate procesa decat un program intr-un anumit moment de timp, acesta nemaifiind utilizat in ziua de azi decat in unele aplicatii restranse.

Multiprogramarea cu partitii fixe.

Evolutia de la monoprogramare, multiprogramarea aduce avantajul ca poate prelucra multiple procese simultan, procesorul putand lucra continuu putand prelua un program in timp ce altul asteapta finalizarea procedurilor de intrare/iesire ducand la o eficienta mai mare.
Aceasta perminte existenta simultana in memoria interna a mai multor programe care se eexecuta concomitent in partitii fixe de memorie cu conditia ca acestea sa nu utilizeze in acelasi timp simultan aceeasi resursa.

Modelarea Multiprogramarii.

Prin multiprogramare se poate obtine o utilizare crescuta a procesorului.
Se pot gasi multiple metode de modelare a acesteia pentru determinarea performantelor fiecare cu partile lui slabe si tari.
Este nevoie a se tine cont de faptul ca procesele nu ruleaza simultan in mod real ci intra succesiv pe bucati in procesor, “mimand” astfel procesul de rulare simultana.

Analiza performantelor sistemelor multiprogramate

Modelul discutat poate de asemenea fi utlizat pentru a analizarea prelucrarii pe loturi. Nu toate procesele utilizeaza procesorul pentru acelasi interval de timp. In cazul sosirii unei a doua sarcini, utilizarea procesorului creste datorita gradului ridicat de multiprogramare, fiecare proces avand acces jumatate de timp la procesor, pierderea de timp de acces la procesor a primei sarcini fiind redusa.

Relocarea si Protectia

Utilizarea multiprogramarii ridica doua probleme esentiale, relocarea si protectia. Sarcini diferite ruleaza de la adrese diferite. Cand programele sunt combinate sa inceapa intr-un singur spatiu de adresa, linkerul trebuie sa stie de la ce adresa va incepe programul in memorie.

2.Swapping ***Raducanu Vlad***

2.1. Introducere

Exista mai multe metode de gestiune a memoriei,dintre acestea metodele cu partitii fixe pot fi utile in cazul unui sistem cu prelucrare pe loturi. Acest tip de metode poate fi eficient in cazul in care memoria este suficient de mare ca sa tina suficient de multe procese astfel incat procesorul sa fie ocupat cat mai mult timp.

In sisteme multiutilizator cu partajare a timpului, sau in computere care au de indeplinit sarcini consumatoare de memorie apare problema ca memoria principala nu este suficienta. Solutia in acest caz este ca o parte din procese sa fie stocate pe harddisk si aduse in memoria principala atunci cand este nevoie. Se folosesc doua strategii generale in acest caz: cea mai simpla, numita swapping, care consta in mutarea proceselor cu totul intre memorie si hard, si cealata ,memoria virtuala, care permite stocarea unor parti din proces pe hard si altora in memoria principala.

Avantajul la swapping este practic ca partitiile memoriei sunt de marime variabila, in functie de process si astfel memoria poate fi utilizata mai eficient, principalul dezavantaj insa este ca se complica algoritmul de alocare a memoriei.

Dupa folosirea swappingului un timp e posibil sa se creeze mai multe zone de memorie libera care sa fie prea mici pentru a putea incarca un proces intreg in ele, pentru a rezolva aceasta problema se poate compactata memoria astfel incat sa avem o singura zona de memorie libera foarte mare si compacta. Insa aceasta actiune este o mare consumatoare de timp si nu este in general utilizata.

Spatiul alocat pentru procese noi, sau aduse de pe harddisk trebuie sa fie mai mare decat procesul in sine, deoarece procesele au tendinta sa creasca in timpul rularii, de asemenea este util sa se aloce acest spatiu in plus intre o zona de stiva(care creste in jos) si o zona de date (care creste in sus) astfel incat utilizarea acestui spatiu sa fie cat mai eficienta.

2.2. Gestiunea memoriei cu harti de biti (bitmaps)

O metoda de gestiune a memoriei in cazul swappingului este cu ajutorul hartilor de biti (bitmaps). Metoda presupune creerea unei harti in care fiecare bit corespunde unei zone de memorie de dimensiune fixa, bitul fiind 0 pentru o zona libera si 1 pentru o zona ocupata. Problema prinicipala in acest caz este alegerea dimensiunii zonei de memorie corespunzatoare unui bit, deoarece alegerea unei dimensiuni prea mari duce la pierderea de spatiu, iar alegerea unei dimensiuni prea mici duce la marirea inutila a hartii memoriei. In cazul in care zonele de alocare au dimensiune prea mare in momentul in care o parte dintr-o zona este utilizata, in harta memoriei intreaga zona este marcata ca “folosita” si astfel se pierde restul de memorie din zona respectiva.

O alta problema cu aceasta metoda este ca atunci cand trebuie adus un proces in memorie trebuie cautata o secventa de biti 0 (corespunzatoare unei secvente de zone libere in memorie) suficient de mare cat sa poate tine procesul respectiv, cautarea unei astfel de secvente intr-o harta de biti este foarte consumator de timp.

2.3. Gestiunea memoriei cu liste inlantuite

Utilizarea listelor inlantuite pentru gestiunea memoriei presupune creerea une liste in care fiecare element se refera la o zona de memorie astfel: zona poate fi ocupata sau libera, se retine adresa de inceput a zonei si adresa de sfarsit a zonei si un pointer catre elementele vecine. Lista de obicei este ordonata dupa adrese astfel incat atunci cand trebuie modificata aceasta sa se faca usor : atunci cand se dezaloca memorie dintr-o zona elementele corespunzatoare zonelor vecine libere impreuna cu elmentul respectiv se combina formand un nou element cu adresa de inceput a primului si adresa de final a ultimului si marcat ca liber.

Pentru a aduce un proces din memorie trebuie cautata o zona de memorie adecvata, pentru aceasta exista mai multi algoritmi cum ar fi first fit (care alege prima zona suficient de mare pentru a tine procesul respectiv, dupa care la urmatorul proces cautarea se face din nou de la inceput), next fit (alege prima zona suficient de mare, dupa care la urmatorul proces cautarea se face incepand cu ultima zona unde s-a ajuns), best-fit (se cauta prin toata lista pana se gaseste cea mai mica zona libera capabila sa acomodeze procesul respectiv), worst fit (se cauta prin toata lista si se alege cea mai mare zona libera). Ideea la worst fist este ca in urma celorlalti algoritmi ajunge la un moment dat sa se creeze niste zone libere prea mici pentru a fi utile si in situatia lui worst fit spatiul liber ramas va fi cat mai mare posibil. In practica insa se dovedeste ca next fit e chiar ceva mai slab decat first fit, ca best fit e chiar mai slab deoarece dureaza f. Mult si creeaza si zonele mici de mem. libera si ca nici worst-fit nu este atat de bun.

O posibila imbunatatire este sa fie tinute 2 liste separate una pentru zonele alocate, alta pentru zonele libere, astfel cautarea se face doar in lista de zone libere si se imbunatateste performanta algoritmilor. Folosind aceasta metoda nici nu mai e nevoie pentru o structura separata pentru lista, ci ar putea pur si simplu fi implementata direct in memorie, astfel la inceputul fiecarei zone sa fie un mic element care sa descrie zona, inclusiv cu pointer catre urmatoarea zona libera si astfel se poate renunta la informatia de liber/ocupat folosind doar 2 cuvinte pentru un element din lista, in loc de 3.

O alta imbunatatire se numeste quick-fit si consta in a tine o lista separata cu zonele libere de marimi uzuale (zone care sunt mai probabil sa fie cerute). Pentru procese ce necesita zone de marimi mai ciudate se pot incarca in zone un pic mai mari, sau se poate tine o lista separata pentru zonele de marimi ciudate. Oricum pentru acest tip de swapping apar din nou problemele de fragmentare.

2.4. Swapping in cazul Linux

Memoria in cazul sistemului de operare Linux este organizata intr-un spatiu virtual de memorie pentru fiecare proces consistand din trei segmente. Segmentul text, in care este tinut “codul” procesului, acest segement este de obicei “read-only”, un segment de date in care sunt tinute variabilele si datele necesare programului, acest segment este impartit in doua o parte pentru datele neinitializate si o parte pentru datele initializate, si un segement de stiva care creste in jos (incepe de la adrese mari si creste catre 0), programele nu pot gestiona explicit marimea acestui segment si in momentul in care se depaseste marimea acestui segment apare o eroare hard si segemetnul este automat marit cu o pagina. Segmentul de date poate creste si se poate micsora si exista o functie de sistem cu ajutorul careia programele pot aloca sau dealoca memorie (brk).

Swappingul in Linux este comandat de nivelul superior al unui programator pe 2 niveluri, anume swapperul. Mutarea din memorie spre harddisk este initiata atunci cand nu mai e suficienta memorie libera dintr-unul din mai multe motive, de exemplu este apelata functia brk pentru a extinde un segment de date, sau este apelata functia fork si e nevoie de memorie pentru procesul fiu sau o stiva creste mai mult decat spatiul alocat ei. Pentru mutarea inversa, anume trebuie adus in memorie un proces care a stat prea mult pe harddisk gasirea unui proces care sa fie trimis spre harddisk pentru a face loc in memorie este gestionata tot de swapper si anume sunt intai analizate procese care asteapta ceva (de exemplu input de la utilizator), daca sunt gasite mai multe este ales cel al carui timp de rezidenta in memorie si al carui prioritate sunt mai mari.Swapperul verifica lista de procese care nu sunt in memorie o data la cateva secunde pentru a decide daca trebuie adus vreunul in memorie. Pentru a preveni suprasolicitarea hardului niciun proces nu poate fi scos din memorie daca nu a stat macar 2 secunde.

2.5. Swapping in cazul Windows 2000

In cazul Windows gestionarea memoriei este mai complicata si swappingul in sine este strans legat de mecanisme complexe de inlocuire a paginilor. Ideea de baza este ca paginile sunt tinute fie in niste seturi de lucru, adica elemente necesare in acel moment pentru procesare, fie in niste liste de pagini cu diferite tipuri de pagini de genul pagini care au fost modificate si nu au fost inca scrise pe disc, pagini care au fost folosite dar nu au fost modificate fata de varianta de pe disc, pagini libere (pagini care contin date, dar nu mai sunt necesare) si pagini cu zero (pagini libere care sunt completate cu 0 peste tot) de asemenea mai exista si o lista cu locatii defecte din RAM astfel incat sa nu se incerce scrierea acolo. Exista mai multe threaduri care muta pagini intre harddisk si memoria RAM, printre care un thread de swapping care din 4 in 4 secunde cauta procese care au fost inactive o anumita perioada de timp si le

muta pe una din listele de pagini modificate, sau de pagini in standby(pagini nemodificate).

2.6.Rezumat

Swappingul ese o metoda de gestiune a memoriei ce presupune transferul unor procese din memoria ram pe harddisk pentru a elibera memorie pentru alte procese.Spre deosebire de memoria virtuala care permite stocarea doar unor parti din proces,la swapping se stocheaza intreg procesul.

In urma utilizarii acestei strategii e posibil ca in memorie sa apara zone goale de memorie foarte mici ce practic devin inutilizabile, astfel este util sa se foloseasca un algoritm eficient pentru swapping.

Exista doua metode de a implementa swappingul si anume cu harti de biti si cu liste inlantuite. Hartile de biti ocupa mai mult spatiu in plus decat listele si algoritmii de cautare a unor spatii libere suficient de mari sunt ceva mai greoi. Pentru gestionarea memoriei cu ajutorul listelor inlantuite exsita mai multi algoritmi cu diferite avantaje si dezavanataje.

In cazul Linux-ului swappingul este comandat de un singur nivel dintr-un sistem de gestiune a memoriei pe mai multe nivele, iar algoritmul implementat evita destul de usor thrashing-ul harddiskului (adica utilizarea excesiva cauzata de un transfer nejustificat de des al proceselor intre harddisk si memorie)

In cazul Windows situatia este mai complicata iar swappingul este comandat de mai multe threaduri si in stransa legatura cu algoritmii de inlocuire a paginilor. Astfel sunt folosite diferite liste pentru a selecta paginile ce trebuie transferate spre hard si eventual reimprospatate, threadul care face swapping scaneaza memoria pentru procese ce au fost inactive mai mult de o perioada determinata de timp si trimite paginile din care este alcatuit pe una din listele respective.

Bibliografie:

Andrew S. Tanenbaum - Modern operating systems
Andrew S. Tanenbaum - Operating Systems Design and Implementation

3.Memoria virtuala ***Muşat Liana***

Memoria virtuală reprezintă o metodă de organizare a memoriei prin intermediul căreia programatorul "vede" un spaţiu virtual de adresare foarte mare care este mapat în memoria fizic disponibilă, fără ca programatorul să "simtă" aceasta.

Aceasta „soluţie" a aparut din nevoia unui spatiu mare de memorie, pentru programe prea mari pentru a intra in memoria disponibila. Soluţia folosită în mod uzual a fost împărţirea programului respectiv în bucăţi numite overlay. Overlay 0 începea să ruleze primul. Atunci când acesta termină, chemă un alt overlay. Unele sisteme bazate pe overlay erau destul de complexe permiţând mai multe overlay-uri în memorie, în acelaşi timp. Overalay-urile erau ţinute pe disc şi erau schimbate între ele în memorie, în mod dinamic, conform cerinţelor. Schimbarea efectiva a overlay-urilor si operaţia de împărţire a programului în bucăţi este făcută de sistem, cu ajutorul memoriei virtuale.

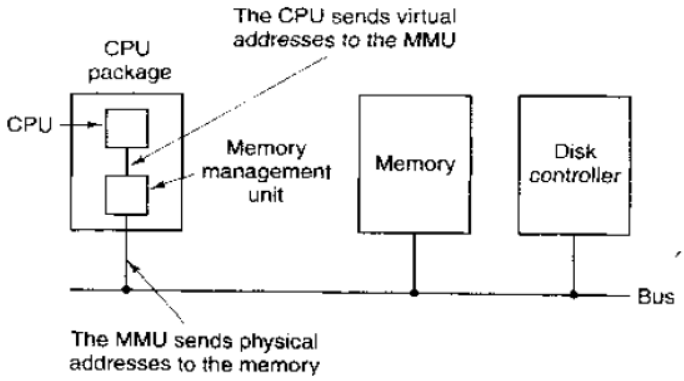
Ideea de bază a memoriei virtuale este aceea că mărimea totală a programului, a datelor, poate depăşi mărimea memoriei fizice disponibile pentru acesta. Programul menţine părţile ce sunt folosite în memorie iar restul părţilor sunt ţinute pe disc. De exemplu, un program de 16 MB poate rula pe un sistem cu 4MB alegând cu grijă care 4MB trebuie să fie menţinuţi în memorie în orice moment, iar părţile de care este nevoie se vor schimba între disc şi memorie.

Prin mecanismele de memorie virtuală se măreşte probabilitatea ca informaţia ce se doreşte a fi accesată de către CPU din spaţiul virtual (disc), să se afle în memoria principală, reducându-se astfel în mod semnificativ timpul de acces de la 10-15 ms (timp acces discuri), la 50-80 ns (timp acces DRAM-uri) în tehnologiile actuale.

3.1.Paginare

Majoritatea sistemelor cu memorie virtuală folosesc o tehnică numită paginare. De obicei, spaţiul virtual de adresare este împărţit în entităţi de capacitate fixă (4-64 Ko actualmente), numite pagini. Unităţile corespondente din memoria fizică sunt numite cadre de pagină. Paginile şi cadrele de pagină sunt mereu de aceiaşi mărime. În general, prin mecanismele de memoria virtuală, memoria principală conţine paginile cel mai recent accesate de către un program, ea fiind pe post de "cache" între CPU şi discul hard.

La orice computer există un set de adrese de memorie ce pot fi produse de programe. Aceste adrese generate de program sunt numite adrese virtuale şi formează spaţiul adresei virtuale. La computerele ce nu au memorie virtuală, adresa virtuală este pusă direct pe bus-ul memorie şi face ca, cuvântul de memorie virtuală să fie scris sau citit. Atunci când se foloseşte memoria virtuală, adresele virtuale nu ajung direct pe bus-ul de memorie. În schimb, acestea ajung la un Unitate de Management al Memoriei care marchează adresele virtuale în adrese de memorie fizică, după cum este prezentat şi în figura.



Transformarea adresei virtuale emisă de către CPU într-o adresă fizică (existentă în spaţiul MP) se numeşte ***mapare*** sau ***translatare***. Memoria virtuală oferă o funcţie de ***relocare*** a programelor (adreselor de program), pentru că adresele virtuale utilizate de un anumit program sunt mapate spre adrese fizice diferite, înainte ca ele să fie folosite pentru accesarea memoriei. Această mapare permite aceluiaşi program să fie încărcat în memoria principală, modificările de adrese realizându-se automat prin mapare (fără memorie virtuală un program depinde de obicei în execuţia sa de adresa de memorie unde este încărcat de către sistemul de operare).

Transferul dintre RAM şi disc se realizează în unităţi de mărimea unei pagini. Atunci când unitatea încearcă să acceseze o adresă 0, folosind instrucţiunea:

MOV REG 0

adresa virtuală 0 este trimisă la Unitate de Management al Memoriei. Acesta observa că adresa intră în pagina 0 (adică de la 0 la 4095). Memoria nu stie nimic despre Unitate de Management al Memoriei si vede doar o cerere de scriere sau de citire, cerere pe care o onorează. Astfel Unitate de Management al Memoriei marchează toate adresele virtuale intre 0 şi 4095. Avem posibilitate de a marca 16 pagini virtuale pe oricare 8 cadre de pagina setând harta Unitate de Management al Memoriei în mod corespunzător nu rezolvă problema prin care adresa virtuală este mai mare decât memoria fizică. Din moment ce dispunem doar de 8 cadre numai 8 din paginile virtuale sunt marcate pe adresa fizică. Celelalte nu sunt marcate. Astfel un bit prezent/absent urmăreşte care pagini sunt fizic prezente în memorie.

Atunci când programul încearcă să folosească o pagină nemarcată Unitate de Management al Memoriei observă că pagina nu este marcată şi face ca CPU-ul să execute o operaţie. Această operaţie se numeşte **eroare de pagină.** Sistemul de operare ia un cadru de pagină puţin uzat şi scrie conţinutul acestuia înapoi pe disc. După aceea ia pagina la care s-a făcut referinţă şi o aplică cadrului de pagină, schimbă marcarea şi reporneşte instrucţiunea prinsă.

Adresa virtuală este împărţită într-un număr de pagini, numărul de pagină este folosit ca un index în tabelul paginii, obţinându-se numărul cadrului conform paginii virtuale. De exemplu, adresa virtuală, 8196 (0010000000000100 în cod binar), o adresa de 16 biţi de intrare care este împărţită într-un număr de pagină de 4 biţi, şi un rest de 12 biţi. Cu 4 biţi pentru numărul paginii, avem 16 paginii şi cu 12 biţi pentru offset putem adresa cei 4096 de biţi în cadrul unei singure pagini. Dacă bit-ul prezent/absent este 0, se declanşează o restricţionare la sistemul de operare. Dacă bit-ul este 1, numărul cadrului ce se găseşte în tabelul din pagină este copiat în comanda superioară de 3 biţi a registrului de ieşire, împreună cu offset-ul de 12 biţi, acesta din urmă fiind copiat nemodificat din adresa virtuală de intrare. Împreună se formează o adresă fizică de 15 biţi. După aceea registrul de ieşire este pus pe bus-ul de memorie sub forma adresei memoriei fizice.

3.2.Tabele pagină

În cazul cel mai simplu, marcarea adreselor virtuale în adresele fizice se realizează conform procedurii descrise mai sus. Adresa virtuală este împărţită într-o număr de pagină virtuală (biţi de ordine mare) şi într-un offset (biţi de ordine inferioară). De exemplu, la o adresă de 16 şi o mărime a paginii de 4KB cei 4 biţi superiori pot specifica una dintre cele 16 adrese virtuale şi cei 12 biţi pot specifica offset-ul (0 la 4095) în cadrul adresei selectate. Însă se poate realiza şi o împărţire cu 3 sau 5 sau cu un alt număr de biţi. Împărţiri diferite implică mărimi diferite de pagină.

Numărul paginii virtuale este folosit drept index în tabelul paginii pentru a găsi intrarea acelei paginii virtuale. Prin intermediul intrării din tabel (dacă aceasta există) se găseşte numărul cadrului paginii. Numărul cadrului este ataşat la partea superioară a offest-ului, înlocuind numărul paginii virtuale, pentru a forma o adresă fizică ce poate fi trimisă memoriei.

Scopul tabelului din pagină este acela de marca paginile virtuale în cadre. Vorbind din punct de vedere matematic tabelul este o funcţie, având numărul paginii virtuale drept argument iar numărul cadrului fizic este rezultatul. Folosind rezultatul acestei funcţii câmpul paginii virtuale este o adresă virtuală ce poate fi înlocuită de un cadru de câmp de pagină , astfel formându-se o adresă în memoria fizică.

trebuie să se ţină cont de două probleme importante:

1. tabelul din pagină poate fi foarte mare
2. marcarea trebuie să se realizeze rapid.

Primul punct rezultă din faptul că, calculatoarele moderne folosesc adrese virtuale de cel puţin 32 de biţi. Luând în considerare o pagină cu mărimea de 4 Kb, un spaţiu de adresă de 32 biţi are 1 milion de pagini, şi un spaţiu de adresă de 64 de biţi are mai mult decât aţi putea calcula. Cu un milion de pagini în spaţiul pentru adresa virtuală tabelul paginii trebuie să aibă 1 milion de intrări. Şi ţineţi cont de faptul că fiecare proces are nevoie de propriul tabel de pagină (deoarece are propriul spaţiu pentru adresa virtuală).

Al doilea punct este o consecinţă a faptului că marcarea de la virtual la fizic trebuie să fie făcută la fiecare referinţă a memoriei. O instrucţiune tipică conţine un cuvânt de instrucţiune şi deseori un operand. În consecinţă este nevoie să faceţi 1, 2 sau chiar mai multe referinţe la tabelul paginii, pentru fiecare instrucţiune în parte. Dacă o instrucţiune durează, să spunem, 4 ns, căutarea în tabelul paginii trebuie să se realizeze în mai puţin de 1 ns pentru a evita stagnarea.

Cel mai simplu design (cel puţin conceptual vorbind) este de a avea un singur tabel de pagină ce este alcătuit dintr-o gamă de registre de componente rapide, cu câte o intrare pentru fiecare pagină virtuală, intrare indexată de numărul paginii virtuale. Atunci când un proces este iniţializat, sistemul de operare încarcă regiştrii cu tabelul paginii procesului, dintr-o copie ce este ţinută în memoria principală. În timpul execuţie procesului nu mai sunt necesare referinţe pentru tabel. Avantajele acestei metode sunt acelea că este o metodă directă şi nu necesită accesul la memorie în timpul procesului de marcare. Dezavantajul constă în faptul că poate fi scump (în cazul în care tabelul este mare). Necesitatea de a încărca întreg conţinutul tabelului duce la diminuarea performanţelor.

La cealaltă extremitate se află posibilitatea conform căreia tabelul din pagină este în întregime în memoria principală. În această situaţie hardware-ul are nevoie de un singur registru ce indică începutul tabelului paginii. Acest design permite schimbarea identificării memoriei la un comutator de context, reîncărcând un singur registru. Desigur, acesta are dezavantajul de a necesita una sau mai multe referinţe de memorie pentru a citi intrările din tabel, în timpul execuţiei fiecărei instrucţiuni. Din acest motiv, această abordare este rar folosită în forma ei cea mai pură, dar mai jos vom studia câteva variaţii care au performanţe mult mai bune.

3.2.1.Tabele de pagină cu nivel multiplu

Pentru a evita problema ce ţine de stocarea continuă a unor tabele imense, multe calculatoare folosesc un tabel cu nivel multiplu. Secretul metodei de tabel de pagină cu nivel multiplu este de a evita ţinerea tuturor tabelelor în memorie în orice moment. În special, acelea de care nu este nevoie nu ar trebui să fie ţinute în memorie.

Pentru un tabel cu doua nivele, cu 1024 de intrări . Fiecare din aceste 1024 de intrări reprezintă 4M deoarece întreg spaţiul adresei virtuale de 4 Gb este împărţit în bucăţi de 1024 de bytes. Intrarea localizată cu ajutorul indexării în partea de sus a tabelului produce adresa sau numărul cadrului paginii pentru un tabel de pagină de nivelul doi. Intrarea 0 din partea de sus a tabelului de pagini indică tabelul de pagină al programului următor, intrarea 1 indica tabelul de pagină ce conţine datele, şi intrarea 1023 indică tabelul de pagină pentru majoritatea informaţiei. Celelalte intrări nu sunt folosite.

3.2.2.Structura unei intrări în tabel

Aspectul unei intrări în tabel este dependent de maşină, dar tipul de informaţii prezent este în mare acelaşi de la maşină la maşină.

Cel mai important câmp este *Page frame number*. Scopul mapării este de a localiza această valoare. Lângă acesta avem *Presen/absent bit*. Dacă aceste bit este 1 intrarea este validă şi poate fi folosită. Dacă intrarea este 0 pagina virtuală corespondentă intrării nu este în memorie. Accesarea unei intrări din tabel ce are acest bit 0 duce la apariţia unei erori de pagină. Biţii *Protection* spun ce fel de acces este permis. În forma cea mai simplă câmpul conţine 1 bit, cu 0 pentru citire/scriere şi 1 bit doar pentru citire. O aranjare mai sofisticată constă în a avea 3 biţi, câte unul pentru permiterea citirii, scrierii şi executării paginii.

Biţii *Modified* şi *Referenced* ţin cont de folosirea paginii. Când se scrie pe o pagină, hardware-ul declanşează automat bitul Modified. Acest bit este de valoare atunci când sistemul de operare decide să revendice un cadru. Dacă pagina din acesta a fost modificată (de exemplu este „murdară”) aceasta trebuie să fie scrisă pe disc. Dacă aceasta nu a fost modificată (de exemplu este „curată”) poate pur şi simplu să fie abandonată din moment ce copia de pe disc este încă valabilă. Bit-ul este uneori numit **bit-ul murdar** din moment ce acesta reflectă starea în care se află pagina.

Bit-ul *Referenced* este setat atunci când o pagină este referită, fie pentru citire sau scriere. Scopul acestuia este de a ajuta sistemul de operare în alegerea unei pagini ce va fi evacuată atunci când apare o eroare de pagină. Paginile care nu sunt folosite sunt candidaţi mai buni decât paginile care sunt, şi acest bit joacă un rol important în câţiva dintre algoritmii de înlocuire a paginii, algoritmi ce vor fi studiaţi mai târziu în această lucrare.

Ultimul bit permite dezactivarea cache-ului pentru această pagină. Această caracteristică este importantă pentru paginile care se mapează pe regiştri în loc de memorie. Dacă sistemul de operare are un timp restrâns de acţionare aşteptând răspunsul unui dispozitiv I/O la o comandă ce tocmai a fost dată, atunci este esenţial ca hardware-ul să ia cuvântul de la dispozitiv şi să nu folosească copia care se află în cache. Cu ajutorul acestui bit cache-ul poate fi oprit. Sistemele care un spaţiu I/O separat şi nu folosesc I/O mapate pe memorie nu au nevoie de acest bit.

Adresa discului ce este folosită pentru a reţine pagina atunci când aceasta nu este în memorie nu face parte din tabel. Motivul este simplu. Tabelul conţine numai acea informaţie de care hardware-ul are nevoie pentru a transforma o adresă virtuală într-o adresă fizică. Informaţia de care sistemul de operare nu are nevoie pentru a manevra erorile de pagină este

ţinută în tabele de program, în interiorul sistemului de operare. Hardware-ul nu are nevoie de ea.

3.3.TLB-uri – Translation Lookaside Buffers

În majoritatea schemelor de paginare tabelele sunt ţinute în memorie din cauza mărimii lor. Din punct de vedere teoretic, acest tip de design are un impact enorm asupra performanţelor. Considerăm, de exemplu, o instrucţiune ce copiază un registru la altul. În lipsa paginaţiei această instrucţiune face o singură referinţă la memorie pentru a lua instrucţiunea. Cu paginarea, va fi nevoie de referinţe de memorie suplimentare pentru a accesa tabelul. Din moment ce viteza de execuţie este limitată de rata cu care CPU-ul poate lua instrucţiuni şi date din memorie, faptul că este necesar să se facă referinţe la tabel pentru două pagini duce la reducerea performanţei cu 2/3. În aceste condiţii nimeni nu ar folosi acest sistem.

Soluţia are la bază observaţia conform căreia majoritatea programelor au tendinţa de a face un număr mare de referinţe la un număr mic de pagini şi nu invers. Astfel, numai o fracţiune mică din intrările din tabel sunt citite în mod constant; celelalte sunt rar folosite.

Soluţia cu care s-a venit a fost echiparea calculatoarelor cu un dispozitiv hardware mic pentru maparea adreselor virtuale în adrese fizice fără a trece prin tabel. Dispozitivul se numeşte **TLB** sau uneori o **memorie asociativă,** este de obicei în interiorul MMU şi este alcătuit dintr-un număr mic de intrări, opt în acest exemplu, dar rareori sunt mai mult de 64. Fiecare intrare conţine informaţii cu privire la o pagină inclusiv numărul paginii virtuale, un bit ce este setat când pagina este modificată, codul de protecţie (permisii de citire/scriere/executare), şi cadrul de paginii fizice în care pagina este situată. Aceste câmpuri au o corespondenţă unu la unu cu câmpurile din tabelul de pagină. Un alt bit indică dacă o intrare este validă sau nu. Atunci când o adresă virtuală este prezentată la MMU pentru translaţie, hardware-ul verifică mai întâi să vadă dacă numărul paginii virtuale este presetat în TLB comparându-l simultan cu toate intrările (în paralel). Dacă se găseşte o pereche validă şi accesul nu încalcă biţii de protecţie cadrul paginii este luat din TLB, fără a mai merge la tabel. Dacă numărul paginii virtual este prezent în TLB dar instrucţiunea încearcă să scrie pe o pagină ce nu poate fi scrisă, o eroare de protecţie este generată, la fel cum s-ar fi întâmplat şi din tabel.

Intrările TLB sun încărcate în mod explicit de către sistemul de operare. Când un TLB dă o eroare, în loc ca MMU să se ducă la tabele pentru a găsi şi extrage referinţa de pagină necesara acesta generează o eroare TLB şi pune problema în seama sistemului de operare. Sistemul trebuie să găsească pagina, să extragă o intrare din TLB, să introducă una nouă şi să repornească instrucţiunea ce a dat eroare. dacă TLB este destul de mare (să spunem 64 de intrări) pentru a reduce rata de ratare, managementul programului corespondent TLB se dovedeşte a fi eficient. Principalul câştig aici este un MMU mult mai simplu, care eliberează suprafaţă semnificativă pe CPU, pentru cache-uri şi alte caracteristici ce ajută la îmbunătăţirea performanţei. Mai multe strategii au fost dezvoltate pentru a îmbunătăţii performanţa maşinilor ce realizează managementul TLB prin intermediul programului. O abordare acoperă atât reducerea numărului de ratări ale TLB cât şi reducerea costului unei ratări TLB, atunci când aceasta apare.

Modul normal de procesare al unei ratări TLB fie că este în program sau în hardware, este să se meargă la tabel şi să se realizeze operaţiile de indexare pentru a localiza pagina la care se face referinţă.

3.4.Table de pagină inversate

Tabelele tradiţionale din tipul celor descrise până acum au nevoie de o intrare pentru fiecare pagină virtuală din moment ce sunt indexate conform numărului paginii virtuale.

Tabelele de pagină inversate economisesc cantităţi mari de spaţiu, urmărind ce proces este localizat in cadrul paginii, cel puţin atunci când spaţiul adresei virtual este mult mai mare decât memoria fizică, acestea au o parte negativă: translaţia virtual-fizic devine mult mai dificilă. Când procesul nu face referire la pagina virtuală, hardware-ul nu va mai putea găsi pagina fizică folosind ca index in tabel.

TLB poate reţine toate paginile care sunt folosite des, translaţia poate avea loc la fel de rapid ca în cazul tabelelor normale. Însă, la o ratare TLB, tabelul de pagină inversat trebuie să fie căutat în program. Un mod fezabil de a realiza această căutare este de a avea un tabel mărunţit, în adresa virtuală. Dacă tabelul mărunţit are la fel de multe sloturi ca şi numărul de pagini fizice al maşinii, lanţul mediu va fi de o intrare, fapt ce măreşte viteza de mapare. Odată ce numărul cadrului paginii a fost găsit, noua pereche (virtual, fizic) este introdusă în TLB.

Tabelele inversate sunt în prezent folosite pe câteva staţii de lucru IBM şi HP şi vor deveni din ce în ce mai des întâlnite pe măsură ce maşinile pe 64 de biţi sunt mai des folosite.

Bibliografie

1) Patterson D., Hennessy J.- Computer Organization and Design: The Hardware - Software Interface

2) Tanenbaum – Modern operating systems

3) Bensoussan, A. & Clingen, C. T. (May 1972), The Multics Virtual Memory: Concepts and Design

4.Algoritmi de înlocuire a paginilor ***Datcu Octaviana***

4.1.Introducere

Atunci când are loc un *defect de pagină** este necesar ca sistemul de operare să elibereze un cadru de pagină prin expulzarea unei pagini care are cea mai mică probabilitate să fie solicitată în lucru, în viitorul apropiat. Algoritmii de înlocuire a paginilor sunt cei care fac posibilă o selecţie cât mai bună a paginii "victimă".

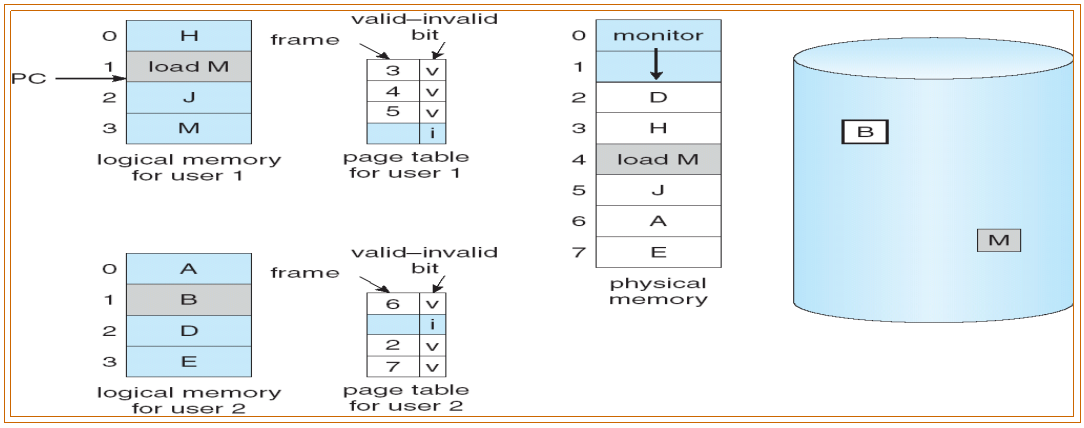


Figura 1 *Este necesară înlocuirea paginilor*

Aceasta are loc într-un sistem de operare ce utilizează paginarea pentru memoria virtuală.
Paginarea survine atunci când o pagină "liberă" nu poate fi folosită pentru ca alocarea să fie facută cu succes, fie pentru că nu mai există pagini libere, fie din cauză că numărul acestora este mai mic decât un anumit prag.

Structurile de date şi mecanismele utilizate pentru alocarea şi dealocarea paginilor sunt critice pentru menţinerea eficienţei subsistemului format de memoria virtuală.

Performanţa acestuia este mult mai bună dacă este aleasă pentru înlocuire o pagină care nu este intens folosită.

În continuare vor fi prezentaţi algoritmii cel mai adesea utilizati pentru a realiza înlocuirea paginilor.

Au fost efectuate studii referitoare la performanţele acestor algoritmi şi au fost propuse variante de îmbunataţire a lor.Vom prezenta, deci, şi câteva dintre aceste variante.

*defect de pagină** = *procesorul doreşte să acceseze o adresă virtuală care nu se află în memoria principală.*

4.2.Strategii de înlocuire a paginilor – algoritmi

4.2.1.Modelul înlocuirii optimale a paginilor (OPRA)

Modelul înlocuirii optime a paginilor a fost descris de către L.A.Belady.

Principiul acestui algoritm este: "înlocuieste pagina care nu va fi folosita perioada cea mai indelungata in viitor".Acest deziderat îi conferă statutul de cel mai bun algoritm de înlocuire a paginilor (din punct de vedere teoretic).Cu implementarea însă lucrurile stau diferit, căci sistemul de operare nu poate privi în viitor, deci este imposibil de realizat practic.

O altă importantă calitate a acestui algoritm este aceea că el oferă suportul de evaluare a celorlalţi algoritmi de înlocuire a paginilor.

Iată, pe scurt, descrierea OPRA :

1. are loc o eroare de pagină ;
2. un set de pagini există in memorie;
3. eticheteaza fiecare pagină cu numărul de instrucţiuni ce vor fi executate înainte ca această pagină să fie folosită din nou, în viitor;
4. înlocuieşte pagina ce are eticheta de valoare cea mai mare.

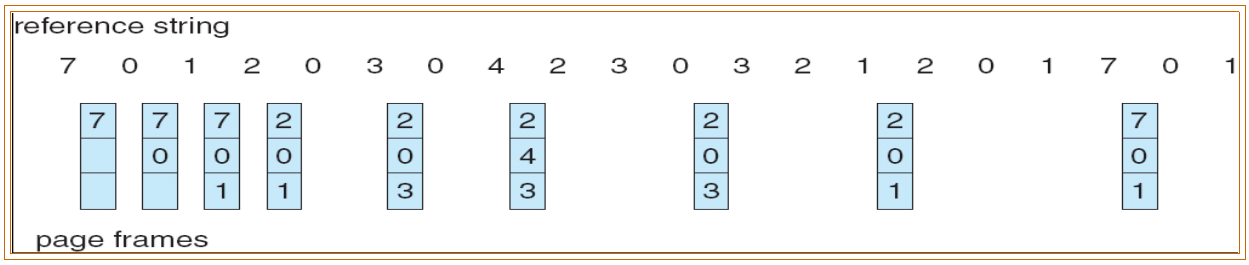


Figura 2 *Exemplu*

Surse bibliografice: [1],[2],[4],[5],[6],[8],[9],[10],[11],[12],[14]

4.2.2. Not Recently Used (Neutilizată recent)

NRU utilizează biţi de stare, R şi M, asociaţi fiecărei pagini, cu următoarea semnificaţie:
R: pagină la care s-a făcut referire (citită sau scrisă);
M: pagină modificată (scrisă).

În functie de valoarea acestor biţi, 0 sau 1, algoritmul face distincţia între patru clase posibile:

		R	M
Clasă 1	**Pagină nereferită, nemodificată**	0	0
Clasă 2	**Pagină nereferită, modificată**	0	1
Clasă 3	**Pagină referită,nemodificată**	1	0
Clasă 4	**Pagină referită, modificată**	1	1

Figura 3 *Tabel ce face distincţia între clase (de pagini)*

La prima vedere, paginile de clasă 1 nu pot fi întâlnite (modificată, dar totuşi nereferită ?!).Este însă posibilă existenţa lor, atunci când o pagina de clasă 3 (referită, dar nemodificată) are, datorită unei întreruperi (clock interrupt), bitul R modificat la 0.Întreruperile

nu modifică si bitul M, deoarece informaţia continuta de acesta este utilă, necesară chiar, pentru a şti dacă pagina respectivă trebuie, sau nu, să fie rescrisă pe disk.Deci, iată o pagină (0,0), de clasă 1.

Algoritmul înlocuieşte, în mod aleator, o pagină din cea mai joasă (ca număr) clasă nevidă;este, deci, înlocuită o pagină modificată, care nu a fost referită în cel puţin un tact de ceas (20 ms, tipic), mai curând decât o pagină nemodificată utilizată intens.

NRU este accesibil ca nivel de înţelegere, implementare si are o performanţă satisfăcătoare.

Surse bibliografice: [1],[4],[10],[12],[14]

4.2.3.First In, First Out (primul intrat, primul ieşit)

Menţine o listă înlănţuită a tuturor paginilor, în ordinea în care acestea apar în memorie .Principiul First In First Out implică faptul că pagina de la ţnceputul listei va fi cea înlocuită.

FIFO are marele dezavantaj de a permite îndepărtarea unei pagini necesare.Din această cauză, el este rar utilizat ţn formă “pură”

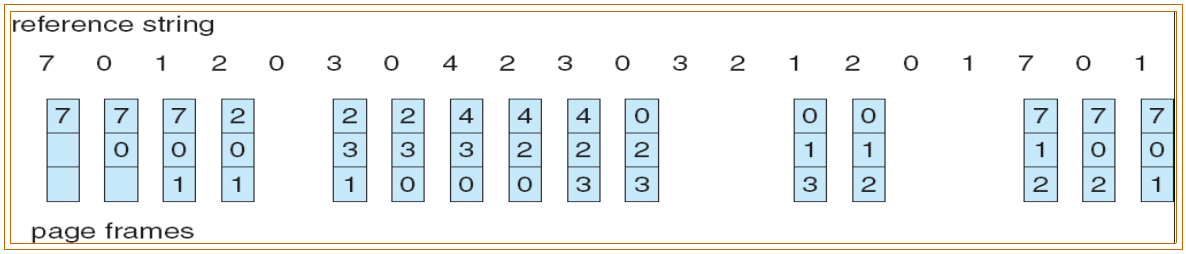


Figura 4 *Exemplu*

Anomalia lui Belady:

Anomalia lui Belady demonstrează (1969) că este posibil să existe mai multe erori de pagină atunci când cadrele de pagină cresc în timpul utilizării algoritmului FIFO.

Procesorul poate încărca un număr limitat de pagini la un moment dat.El pretinde un cadru pentru fiecare pagină ce o poate încărca.

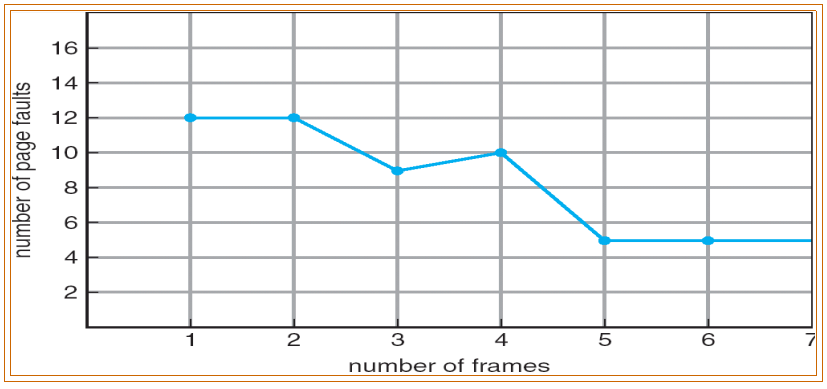


Figura 5 Ilustrarea anomaliei lui Belady

Surse bibliografice: [1],[2],[4],[5],[6],[7],[8],[10],[11],[12],[14]

4.2.4. Least Recently Used (Cel mai recent utilizat)

Paginile intens folosite în ultimele câteva instrucţiuni vor fi probabil intens utilizate din nou în urmatoarele câteva instrucţiuni.Paginile care nu au fost folosite timp îndelungat, au probabilitate mare de a rămâne neutilizate pentru o perioada mare, în viitor.Această idee se apropie promiţator de aceea a algoritmului lui Belady.

LRU foloseşte, deci, informatia referinţei, pentru a lua o decizie în privinţa înlocuirii unei pagini într-o mai bună “cunoştinţă de cauza”.Urmând idea de predicţie a viitorului sugerată de algoritmul optimal, “ghiceşte” viitorul pe baza experienţei anterioare.

Implementarea LRU nu este ieftină, pentru că pentru aceasta este necesar a menţine o listă înlănţuită a tuturor paginilor din memorie, cu cel mai recent folosită pagină în prim plan, cea care a fost utilizată cel mai puţin recent, ramânând pe o ultimă poziţie.Lista trebuie sa fie reactualizată la fiecare referire a memorie.De aceea, se consumă timp mult prea mare (se caută pagina în listă, este găsită şi ştearsă din memorie, şi rescrisă la începutul listei).

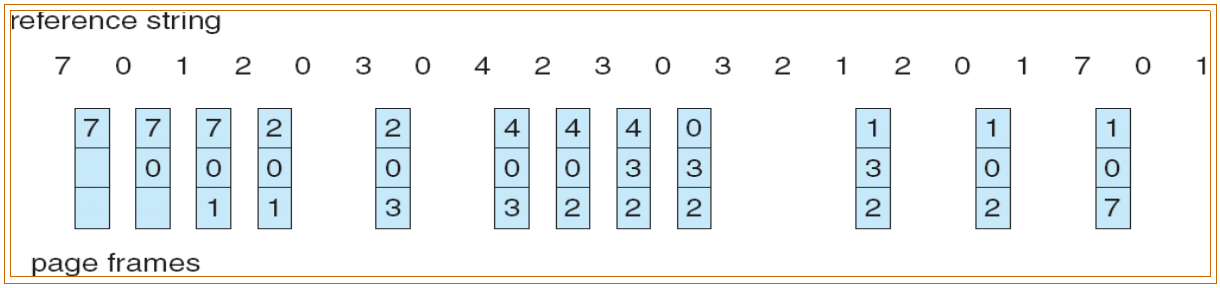


Figura 6 *Exemplu*

Este necesară, deci, o aproximare a acestui algoritm, pentru a-i conferi o imbunătăţă.

Surse bibliografice: [1], [2], [3],[4],[5],[6],[7],[8],[10],[11],[12],[14]

4.2.5. Implementarea cu stivă a LRU

Este păstrată o stivă a numerelor paginilor.Pagina referită este mutată în capul listei.Dezavantajul acestei operaţii constă în utilizarea a 6 pointeri pentru a fi modificată.Avantajul metodei este că nu se face căutare pentru înlocuire.

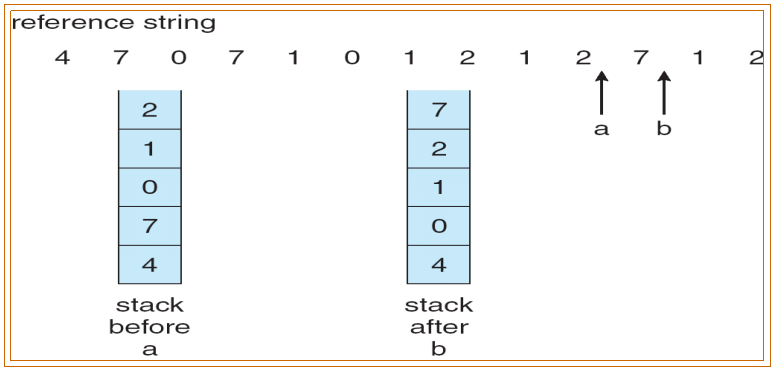


Figura 7 *Exemplu*

4.2.6.Algoritmi de aproximare LRU

4.2.6.1. Second Chance (A Doua Şansă)

Bitul de referire:

1. asociem fiecărei pagini un bit, R, iniţializat cu 0;
2. când pagina este referită, bitul R devine 1;
3. înlocuieşte pe cea cu R=0, dacă există.

A doua şansa:

1. este nevoie de bit de referire;
2. este o înlocuire cu ordine;
3. dacă pagina de înlocuit (în ordinea ceasului) are R = 1 atunci:
   1) setează R = 0;
   2) păstrează pagina în memorie;
   3) înlocuieşte următoarea pagină (în ordinea acelor de ceasornic);
   4) se repetă algoritmul de la 3) pentru fiecare pagină.

Second Chance (A Doua Şansă) este o formă modifictă a FIFO, îmbunătăţită.În aceeaşi manieră ca şi FIFO, puncteza la capătul cozii.Dar, faţă de FIFO, verifică, în plus, dacă bitul său R este setat.În cazul în care nu este setat, pagina este “scoasă din joc”.Pentru R setat, acesta este resetat, pagina este inserata la sfârşitul cozii, şi procedeul este repetat.

A Doua Şansă dă fiecărei pagini o “a doua oportunitate”- o pagină care a fost referită este cel mai probabil în folosinţă, şi nu ar trebui îndepărtată, spre deosebire de o nouă pagină care nu a fost referită.

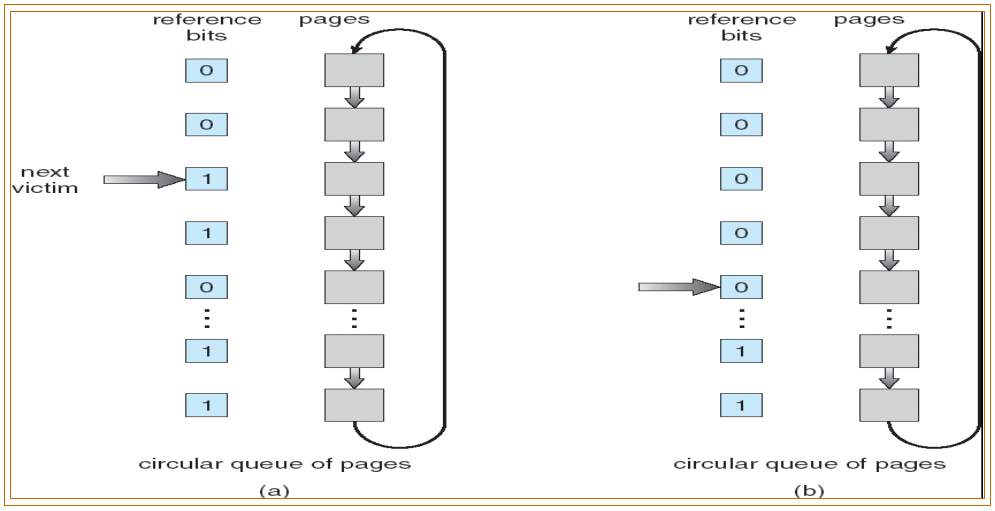


Figura 8 *Exemplu*

Surse bibliografice: [1],[2],[3],[5],[6]

4.2.6.2. Clock (Ceasul)

Algoritmul A Doua Şansă este rezonabil, dar ineficient întrucât roteşte paginile în listă în mod constant.O abordare mai bună constă în păstrarea tuturor cadrelor într-o listă circulară în forma unui ceas.

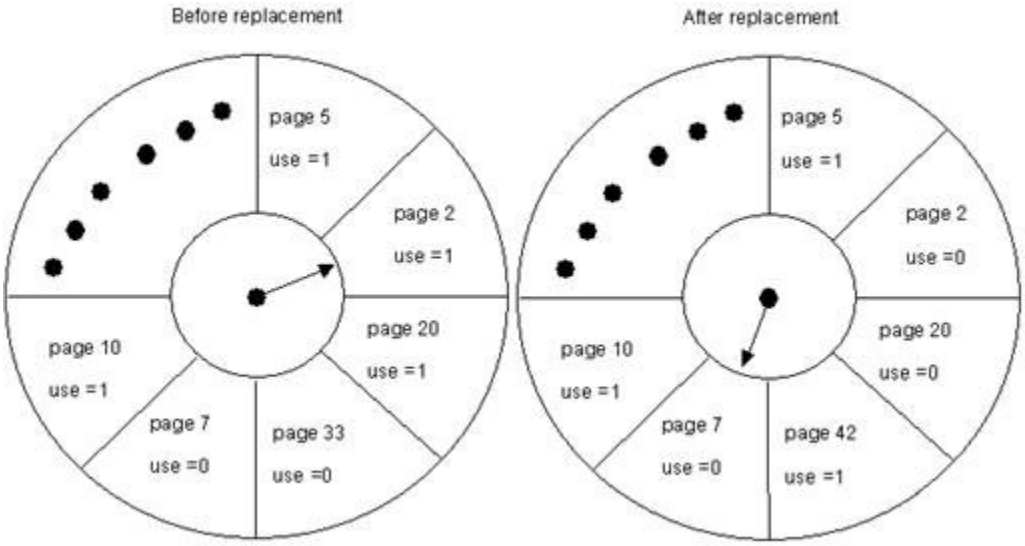


Figura 9 *Exemplu*

Pagina utilizată în urmă cu cel mai mult timp (prima din coadă) este analizată.Dacă bitul ei R=0, pagina este înlocuită, o nouă pagină este inserată în ceas în locul său.Dacă *R*=1, el re este resetat la 0 şi se trece la următoarea pagină în ordinea impusă de FIFO (pagina curentă este păstrată în memorie).

Algoritmul diferă de cel numit A doua Şansă doar prin implementare.

Surse bibliografice: [3],[10],[11[,[14]

4.2.6.3. Not frequently used (Neutilizată frecvent)

NFU generează mai puţine defecte de pagină decât LRU atunci când tabelul de pagini conţine valori nule de pointeri.

Presupune existenţa unui numărător, fiecare pagină având un contor propriu, iniţializat cu 0.La fiecare interval de ceas, pentru toate paginile care au fost referite în acel interval numărătorul le va fi incrementat cu 1, aceste numărătoare reţinând frecvenţa cu care o anumită pagină a fost folosită.Astfel, pagina cu numărătorul având valoarea ce mai mică poate fi îndepartată atunci cand este necesar.

Dezavantajul acestei strategii este că ţine cont de frecvenţa de utilizare, fără a reţine şi *timpul de utilizare*.De aceea, o pagină care a fost intens utilizată în timpul primului pas, va fi favorizată faţă de una care este utilizată mai puţin, în cel de-al doilea pas, deoarece are o frecvenţă mai mare.Acest lucru conduce la o performanţă scăzută.

Surse bibliografice: [2],[10],[11],[14]

Există un algoritm similar, dar mai bun:

4.2.6.4.Aging (Îmbătrânirea)

Modificare care îi este adusă lui NFU, transformându-l în “Aging” este aceea că, noul algoritm ţine cont nu numai de frecvenţa de utilizare a paginilor, ci şi de timpul cât dureaza această utilizare.

Numărătorul nu mai este pur şi simplu incrementat cu 1, ci, anterior acesta este deplasat la dreapta (deci, ţmpărţit la 2), la stânga a ceea ce s-a obţinut fiind adăugat bitul de referinţă.

Paginile referite cel mai recent, chiar dacă mai puţin referite, vor avea o prioritate mai mare decât paginile referite mai frecvent, dar ţn trecut.

Când este necesar să fie îndepărtată o pagină, atunci va fi aleasă aceea care are cea mai mică valoare în numărător.

Iată, deci, că, “Aging” poate oferi o performanţă apropiată celei optimale, contra unui cost mediu.

Surse bibliografice: [2]

4.2.7. Random

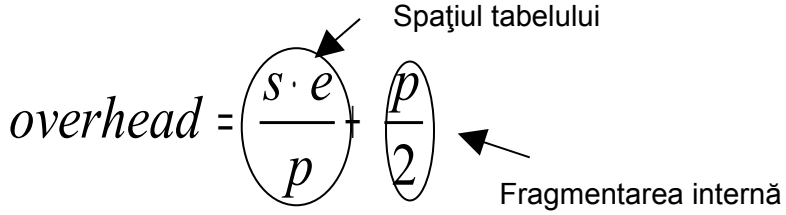
Algoritmul de înlocuire “Random” alege în mod aleator, aşa cum sugereaza şi denumirea sa, o pagină pentru a fi înlocuită.Aceasta elimină inconvenientul costului necesar memorării referirii la pagini.

Este mai bun decât FIFO.Pentu referiri ale memoriei în buclă este mai bun chiar şi decât LRU, deşi acesta îşi găseşte o mai bună aplicabilitate în practică.

Surse bibliografice: [4],[7]

4.3. Impactul dimensiunii paginii asupra performanţelor memoriei virtuale

Dimensiunea paginilor este de dorit a fi aleasă astfel încât să se reducă fragmentarea internă.



s = dimensiunea medie a unui proces [bytes];
p = dimensiunea paginii [bytes];
e = dimensiunea intrării de pagină;

Dimensiunea optimă se determină prin calculul derivatei expresiei în funcţie de p şi anularea acesteia.

$-(s*e)/(p*p) + ½ = 0 \rightarrow p = (2 * s * e)^{1/2}$

Soluţionarea unei erori de pagină:

1. sistemul de operare consultă un alt tabel:
   - ¤ dacă adresa nu este validă: sfârşitul procesului;
   - ¤ dacă adresa este validă: tabelul indică adresa paginii pe disc.
2. se alocă un frame liber;

3. se încarcă pagina referită în memorie;
4. se reactualizează tabelul paginilor (bitul de invalidare = 1);
5. se porneşte reexecuţia instrucţiunii

Această modalitate se numeşte a face *paginarea la cerere.*

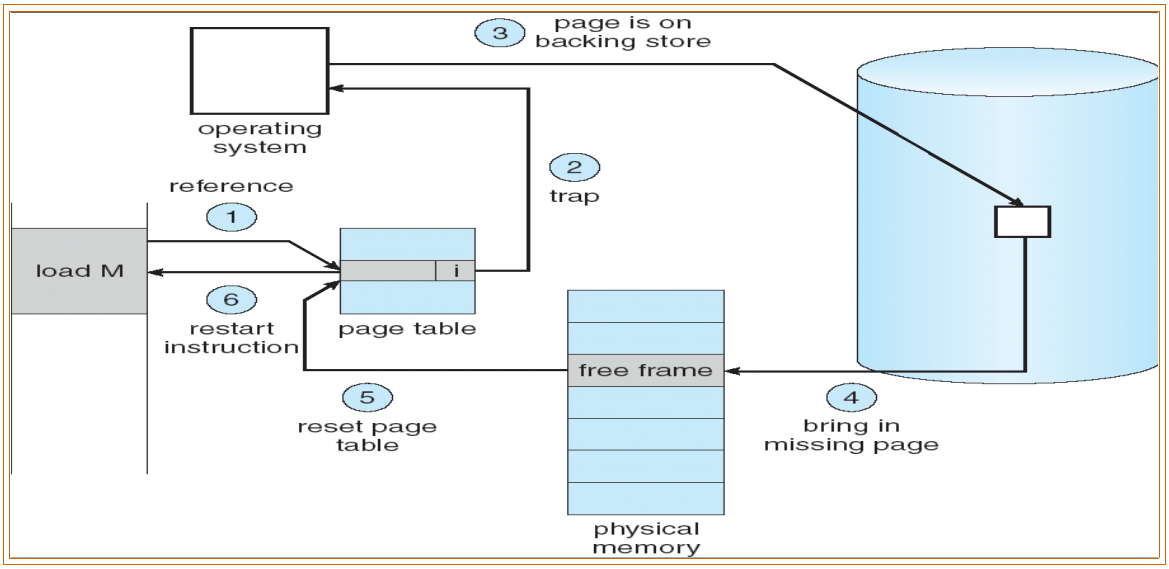


Figura 9 *Soluţionarea unei erori de pagină*

Surse bibliografice: [1],[3],[6]

4.4.Comportamentul programului în cazul paginării

Sistemul de operare determină dimensiunea datelor şi pe cea a instrucţiunilor.Alocă şi iniţializează tabelul de pagini.Apoi alocă aria de swap pe disc pentru a stoca tabelul de pagini când procesul este swapped out.Este memorat tabelul de pagini şi aria de swap în tabelul procesului.Posibila este şi preîncărcarea de pagini.

La începerea execuţiei programului, este resetat MMU (Memory Management Unit), încărcat TLB-ul şi copiat tabelul de pagini sau adresele de start/sfârşit în registrele hardware.

Încheierea procesului presupune eliberarea tabelului de pagini, a cadrelor de pagini şi spaţiului de swap de pe disc.

În cazul *defectului de pagină,* S.O. determină care adresă virtuală a cauzat eroarea de pagină şi află ce tabel este necesar pentru a o localiza pe disc.Apoi expulzează o pagină pentru a face loc în memoria principală (dacă e nevoie).Întoarce numărătorul de program pentru a indica la instrucţiunea eronată astfel încât aceasta să poată fi executată din nou (de acestă dată, cu pagina necesară aflată în memorie).

Aici îşi găsesc utilitatea algoritmii prezentaţi la punctul 4.2.

Surse bibliografice: [3],[6]

4.5. Working Set (Setul de lucru)

Localitatea programelor este o proprietate observată experimental.În fiecare interval de timp, un process face referire doar la un subansamblu al paginilor sale.
Formalizare:

- procesul ocupă n pagini : 1, …, n-1
- execuţia procesului generează o secvenţă de referinţe la aceste pagini
- $w_{\Delta(\tau)}(t)$ = paginile referiteîn intervalul [t-Δt,t]

Principiul localităţii : $w_{\Delta(t)}(t) = w!_{\Delta(\tau\square}(t+\Delta t)$

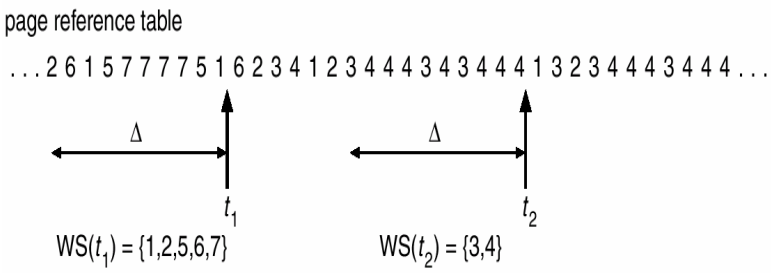


Figura 10 *Setul de lucru*

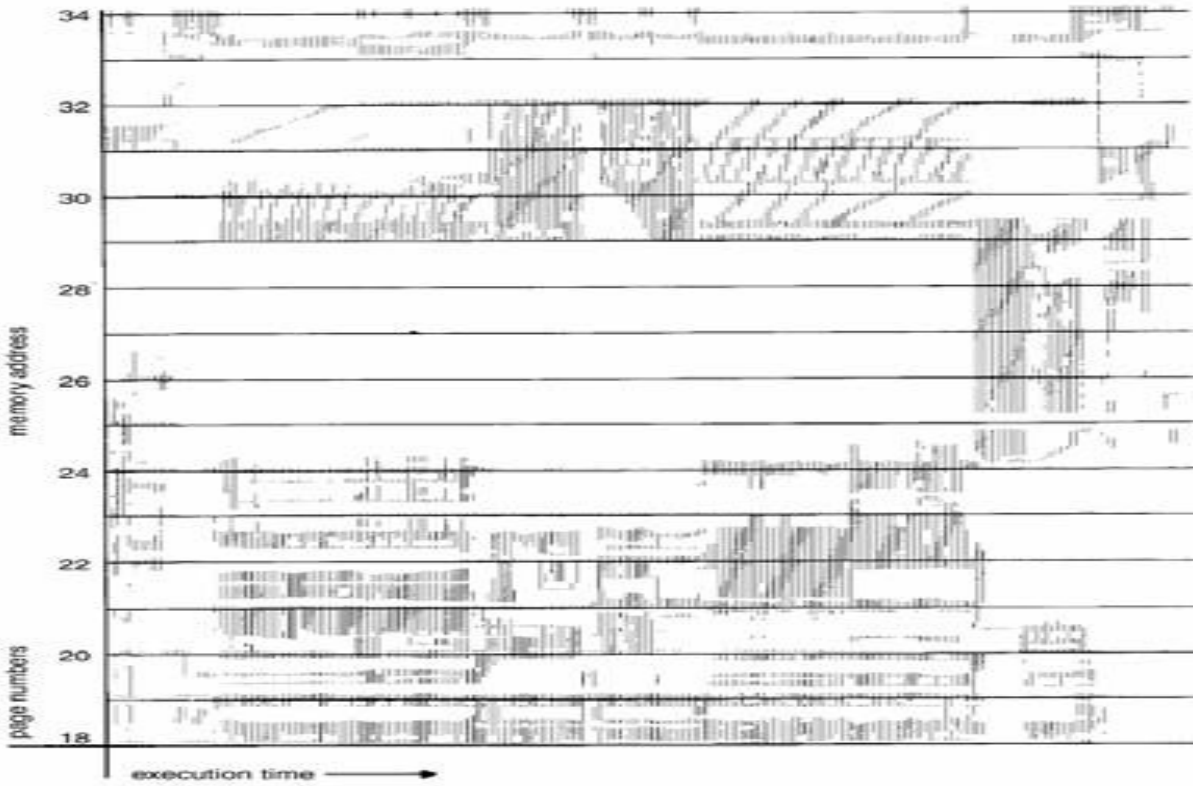


Figura 11 *Localitatea*

1. *Localitatea temporală :* o adresă care tocmai a fost referită are mari şanse să fie referită în imediata apropiere temporală.
   Exemplu : bucle, variabile locale ale unei proceduri, proceduri etc.
2. *Localitatea spaţială*:dacă o adresă a a fost referită, există şanse mari ca o adresă vecină lui a (adică în aceeaşi pagină) să fie referită în curând.
   Exemplu: execuţie secvenţială a codului, parcurgerea tablourilor etc.

Deşi costul paginării este ridicat, dacă frecvenţa producerii acesteia este suficient de mică, paginarea este acceptabilă.

De regulă, procesele implică ambele tipuri de localitate în timpul execuţiei lor, dând paginării practibilitate.

Din persepectiva sistemului de operare, când memoria este plină, pagini sunt expulzate, şi încărcate de pe disc atunci când sunt referite din nou.Referirea la pagini expulzate cauzează erori în TLB.

Cererea de pagini are loc şi atunci când un proces este lansat pentru prima dată;procesul are un tabel al paginilor nou, cu toţi biţii valizi 0 şi nici o pagină în memorie.Când îşi începe execuţia, instucţiunile produc defecte în cod şi pagini de date.Erorile încetează atunci când întregul cod şi paginile de date necesare se află în memorie.Înărcarea este necesară doar pentru codul şi paginile active ale procesului.

Spaţii fixe / spaţii variabile

Într-un sistem cu multiprogramare este nevoie să alocăm memorie proceselor concurente.Dar cum să determinăm ce cantitate de memorie să dăm fiecărui proces?

Algoritmi de spaţiu fix:

- fiecărui process îi este dat un anumit număr de pagini limitat ca să le poată folosi;
- când procesul atinge această limită începe să înlocuiască din propriile pagini;
- înlocuirea este locală;
- unele procese pot să funcţioneze foarte bine, în detrimentul altora.

Algoritmi de spaţiu variabil:

- setul de pagini al procesului creşte şi se reduce dinamic;
- înlocuirea este globală;
- unul dintre procese poate să deterioreze funcţionarea celorlalte.

Setul de lucru al unui process este utilizat pentru a modela localitatea dinamică a folosirii memoriei în cazul său.Reprezintă numărul adecvat de pagini pentru ca un proces să funcţioneze eficient, deci, acel număr de pagini pe care procesul le foloseşte active, la un moment dat, care trebuie să fie încărcat în memorie la acel moment pentru ca procesul să fie executat.

Colocvial, WS este setul de pagini pe care procesul le foloseşte în chiar acest moment.Mai formal, WS este setul tuturor paginilor pe care procesul le-a referit în ultimele T (Δt) secunde.

Modul în care sistemul de operare alege T-ul este următorul:la un defect de pagină corespund 10 ms = 2 milioane de instrucţiuni.Dar T trebuie să fie mult mai mare ca 2 milioane de instrucţiuni.Dacă T este prea mic, sunt îndepărtate pagini aflate încă în uz, la cealaltă extremă (T prea mare) aflându-se utilizarea ineficientă a memoriei (reducerea inutilă a nivelului de multiprogramare).

Paginile setului de lucru sunt înlănţuite (listă circulară).Fiecare pagină a WS are un bit de referinţă setat la 1 pe durata tuturor referinţelor la pagina respectivă.Căutarea paginii care va fi adusă presupune parcurgerea listei, pornind de la poziţia curentă.Dacă pagina are R=1, bitul este resetat la 0.Dacă R= 0, pagina este adusă în memorie.

Numărul de pagini referite în intervalul [t,t-T] este dimensiunea WS-ului.Setul de lucru se schimbă cu localitatea programului.Pe durata unei localităţi slabe sunt referite mai multe pagini, dimensiunea WS-ului fiind mai mare.

Problemele care se pun sunt: 1) cum determinăm T;2) cum ştim care este momentul în care se schimbă WS-ul.Din această cauză WS nu este folosit în practică ca un algoritm de înlocuire a paginilor.Este folosit ca o abstractizare.

Surse bibliografice: [1],[2],[4],[5],[6],[7],[8]

4.6. Fenomenul “thrashing”

Thrashing-ul are loc atunci când cea mai mare parte a timpului sistemul de operare se ocupă de paginarea datelor pe/de pe disc, în loc să progreseze dând prioritate lucrului util.Sistemul este astfel suprasolicitat.

Nu ştim care pagini sunt cele care ar trebui să fie în memorie pentru a reduce defectele.Este posibil să nu existe suficientă memorie fizică pentru toate procesele din sistem, pur şi simplu.Posibilele soluţii sunt swapping-ul sau cumpărarea a mai multă memorie.

Algoritmii de înlocuire a pginilor evită thrashing-ul .

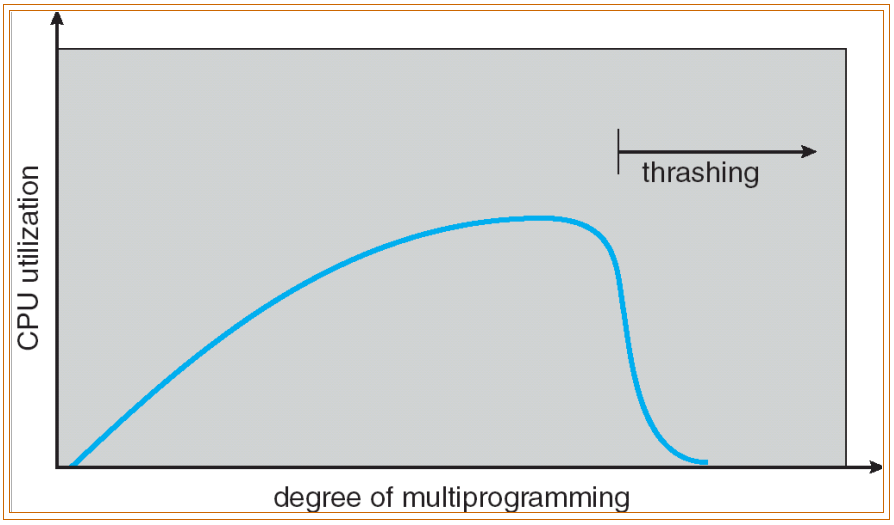


Figura 12 *Thrashing*

Surse bibliografice: [2],[5],[6]

4.7. PFF (Frecvenţa de eroare de pagină)

Este un algoritm de spaţiu variabil care foloseşte o abordare mult mai “ad-hoc”.

Monitorizează frecvenţa de eroare pentru fiecare proces.Dacă aceasta este peste un prag înalt ales, i se dă procesului mai multă memorie, astfel încât să fie mai puţine defecte, dar nu funcţionează întotdeauna (Ex.: FIFO, anomalia lui Belady).Dacă frecvenţa de eroare este sub un alt prag ales, jos, se ia memorie de la proces.Ar fi normal ca el să producă mai multe defecte, dar nu este regulă.

Este greu să se utilizeze PFF pentru a distinge între schimbările de localitate şi schimbările în dimensiunea WS-ului.

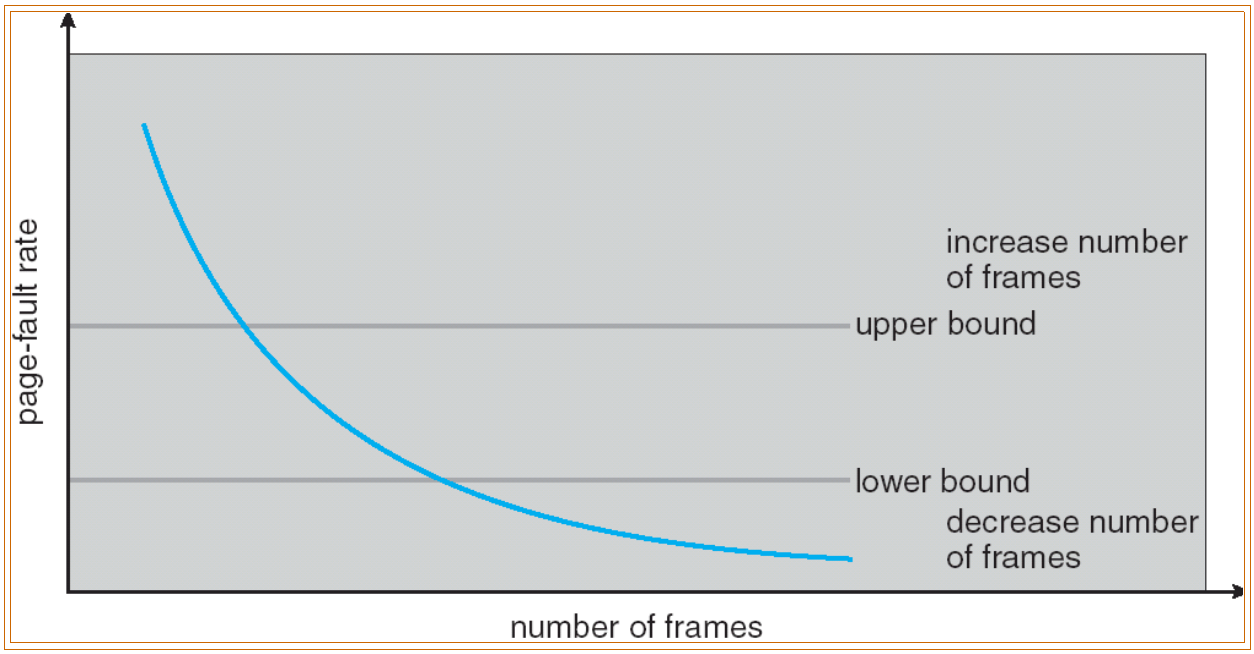


Figura 13 *Frecvenţa de defect de pagina în funcţie de numărul de cadre.*

Surse bibliografice: [5],[6]

4.8. Alţi algoritmi

4.8.1. "Low inter-reference Recency Set Replacement Policy" LIRS

Departe de a pretinde să acoperim toate aspectele acestui algoritm, vom prezenta modul în care acesta funcţionează:

Blocurile de referinţă sunt împarţite în două clase: HIR (High Inter-reference Recency) şi LIR (Low Inter-reference Recency).Fiecare bloc deţine informaţii referitoare la istoria paginilor conţinute, în cache, având intrările în înregistrările din cache ca HIR de non-rezidenţă.

Cache-ul, cu dimensiunea L (blocuri) este împărţit, la rândul său, într-o parte majoră (de dimensiune L_{lirs}) şi una minoră (de dimensiune L_{hirs}).

$L = L_{lirs} + L_{hirs}$

											"Recency"	IRR
E									**x**		**0**	**Inf**
D		**x**					**x**				**2**	**3**
C				**x**							**4**	**Inf**
B			**x**		**X**						**3**	**1**
A	**x**					**x**		**x**			**1**	**1**
blocuri/timp virtual	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**		

Figura 14 Cum este ales blocul "victimă" şi cum sunt interschimbate stările LIR/HIR

x prezent într-o casuţă a tabelului înseamnă că rândul corespunzător este referit la momentul virtual din coloana corespunzătoare.Recency şi IIR reprezintă valorile ce corespund momentului virtual 10, pentru fiecare bloc.

Astfel, presupunând L_{lirs}= 2 şi L_{hirs} = 1, la momentul 10, setul HIR = { C,D,E}, setul LIR = { A, B}.Singurul bloc rezident este E.

Surse bibliografice: [13]

4.8.2. "Enhanced second chance/clock"

Algoritmul foloseşte clasele (R,M), definite în tabelul din **Figura 3**.Înlocuieşte prima pagină pe care o găseşte în clasa nevidă cu cea mai mică prioritate.

Un dezavantaj este acela ca poate fi necesar să se scaneze coada de mai multe ori până la înlocuire.

Dacă "victima " este "murdară" (M=1), atunci se consumă timp cât aceasta este swappată.Este, deci, mai convenabil să se aleagă o victimă "curată" (M=0).

Astfel este determinată prioritatea claselor:

- clasa 1 (R=0,M=0): pagina ce îi aparţine este un bun candidat pentru înlocuire;
- clasa 2 (R=0, M=1): vechile pagine trebuie să fie scrise;nu este un candidat la fel de bun ca anterioara;
- clasa 3: (R=1,M=0): pagina este recent referită, ceea ce o face un candidat prost;
- clasa 4: (R=1,M=1) :în nici un caz un bun candidat, din cauză că trebuie, de asemenea, să fie scrisă.

Surse bibliografice: [3],[5]

4.8.3. "Page Buffering"

Până acum a fost presupusă aplicarea algoritmilor ori de câte ori este nevoie ca o pagină să fie adusă în memorie.

Majoritatea algoritmilor discutaţi anterior sunt prea costisitori pentru a fi utilizaţi la fiece defect de pagină.

"Page Buffering" reţine o "piscină" a paginilor libere.Algoritmul de înlocuire este utilizat abia atunci când aceasta devine prea neîncăpătoare ("low water mark"), eliberând sufficient spaţiu pentru o nouă aprovizionare cu pagini("high water mark") .

Când apare un defect de pagină, se alege un cadru din lista liberă.Cadrele din această listă conţin încă informaţia precedentă, şi pot fi salvate dacă pagina virtuală este referită anterior realocării.

Surse bibliografice: [1]

4.9. Concluzii asupra performanţei principalilor algoritmi

Atunci când are loc un defect de pagină este necesar ca sistemul de operare să elibereze un cadru de pagină prin expulzarea unei pagini care are cea mai mică probabilitate să fie solicitată în lucru, în viitorul apropiat. Algoritmii de înlocuire a paginilor sunt cei care fac posibilă o selecţie cât mai bună a paginii "victimă".

Iată cum “se descurcă” algoritmii de înlocuire a paginilor întâlniţi frecvent şi prezentaţi în cele anterior spuse:

1. OPRA este un algoritm optimal, ce conferă (teoretic) numărul minim de defecte de pagină.Utilizat ca etalon pentru ceilalţi algoritmi.
2. FIFO înlocuieşte pagina încărcată în cel mai îndepărtat moment de timp. Poate îndepărta pagini importante (necesare)
3. LRU înlocuieşte pagina referită la momentul cel mai îndepărtat în viitor.Excelent ca performanţă, dar dificil de implementat.
4. CLOCK înlocuieşte pagina cu vechimea cea mai mare.
5. WORKING SET păstrează în memorie setul de pagini cu frecvenţă de eroare minimă.Costisitor.
6. PFF măreşte/micşorează setul de pagini în funcţie de frecvenţa de eroare.
7. NRU prea radical.
8. A Doua Şansă ameliorare considerabila a FIFO.
9. CEASUL (clock) realist.
10. NFU aproximare grosieră a LRU.
11. AGING aproximare eficientă a LRU.

4.10. Caracteristici ale paginării în Windows XP/ Linux

Windows XP

- **înlocuire a paginilor locală;**
- **FIFO pe process;**
- **Sunt luate pagini de la procese ce folosesc mai multe pagini decât dimensiunea minimă a WS-ului lor, şi oferite altor procese care suferă de lipsa paginilor disponibile;**
- **Procesele sunt lansate cu WS = 50 de pagini, implicit;**
- **Sistemul monitorizează frecvenţa de eroare (defect) şi ajustează WS-ul în concordanţă cu aceasta;**
- **La apariţia unui defect de pagină, grupuri de pagini din jurul paginii lipsă sunt aduse în memorie.**

Linux

- **înlocuire a paginilor globală;**
- **utilizează algoritmul clock pentru înlocuire;**
- **paginile imbătrânesc cu fiecare pas parcurs de “mâna” indicatoare a clock-ului;**
- **paginile ce nu au fost folosite vreme îndelungată vor avea în final valoarea 0.**
- **sistemul este în plină dezvoltare (noutăţi apar încontinuu).**

Surse bibliografice: [5]

4.11. Referinţe

1. LABORATOIRE DE SYSTEMES REPARTIS, Systèmes d'exploitation, Implantation de programmes en mémoire, Mémoire virtuelle, Applications,Gestion de la mémoire, Alain Sandoz, Semestre été 2007;
2. OS @ SEP501 (Spring 2007) -- Jin-Soo Kim (jinsoo@cs.kaist.ac.kr);
3. Department of Computer Science, UMass Amherst, Andrew H. Fagg, CMPSCI 377: Operating Systems, Lecture 22;
4. CS 241 Fall 2007 System Programming,Memory Paging and Replacement, Lawrence Angrave;
5. CSCI 4401 Principles of Operating Systems I,Virtual Memory ,Vassil Roussev vassil@cs.uno.edu;
6. Operating Systems Concepts, Silberschatz, Galvin and Gagne, 2005;
7. Sistemi a microprocessore, Memoria Virtuale,Corso di Laurea in Ingegneria dell'Informazione,Università degli Studi di Firenze ;
8. Virtual Memory:Page Replacement, Victoria University of Wellington, New Zeeland;
9. A study of replacement algorithms for a virtual-storage computer by L. A. Belady;
10.Tanenbaum, Andrew S. - Operating Systems Design and Implementation, 2Ed;
11.Jackson libri - Tanenbaum - Moderni Sistemi Operativi;
12. CSE 120, Principles of Operating Systems, Fall 2004, Lecture 11: Page Replacement, Geoffrey M.Voelker;
13. LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance ;
14.Andrew Tanenbaum, Modern Operating Systems.

Notă: Internetul a fost cel cu ajutorul căruia au fost accumulate materialeel bibliografice.

5.Modelarea algoritmilor de inlocuire a paginilor *Velican Valentin*

5.1. Definitie.Scurt "istoric".

Într-un sistem de operare ce utilizează paginarea pentru organizarea memoriei virtuale, **algoritmii de înlocuire a paginilor** decid care dintre pagini trebuie înlocuită (swap out) când o nouă paginã din memorie trebuie alocatã.

Din punct de vedere istoric acest subiect a fost foarte dezbătut în perioada anilor '60; '70 rezultând, prin studierea sa, algoritmi LRU foarte sofisticaţi. Însă, din momentul respectiv şi până în prezent, câteva ipoteze "tradiţionale", luate în calcul la dezvoltarea acestor algoritmi, au devenit un real obstacol pentru performanţă având drept finalitate o nouă intensificare a cercetărilor. Spre exemplu pot fi reamintite:

- dimensiunea unităţii de stocare ce a crescut cu cateva ordine de mărime; în acest fel algoritmii ce necesitau o verificare periodică a fiecărei locaţie de memorie devin din ce în ce mai puţin practici.
- ierarhizarea memoriei a crescut; astfel, "costul" unui "cache miss" raportat de procesor a devenit din ce în ce mai ridicat.

- răspândirea programării orientate pe obiect, ce utilizează un număr ridicat de funcţii precum şi a unor tipuri de date sofisticate (e.g. structuri arborescente) aduce cu sine o utilizare “haoticã” a memoriei.

Cerinţele de la algoritmi s-au schimbat datoritã diferenţelor dintre sistemele de operare; în particular majoritatea sistemelor de operare moderne au reunit memoria virtualã cu fişierele cache de sistem. Astfel algoritmul este nevoit sã aleagã o paginã atât din paginile adreselor virtuale cât şi din cel al fişierelor cache. Cele din urmã au anumite particularitãţi, precum cerinţelor de scriere într-o anumitã ordine. În plus algoritmul trebuie sã ia in calcul şi cerinţele de memorie impuse de alte sub-sisteme ale kernel-ului (ce la rândul lor alocã memorie). Ca rezultat înlocuirea paginilor în kernel-uri moderne (Linux, FreeBSD, Solaris) tinde sa lucreze la nivel mai scãzut faţã de un subsistem al memoriei virtuale.

5.2.Înlocuirea localã vs. Înlocuirea globalã

Algoritmii de înlocuire pot sa fie *locali* sau *globali*. Un algoritm de înlocuire ”local” alege pentru schimb o paginã apartinând aceluiaşi proces în timp ce un algoritm “global” este liber sã aleagã o paginã de oriunde din memorie.

Înlocuirea localã a paginilor implicã existenţa unui mod de partiţionare a memoriei care determinã exact câte pagini vor fi atribuite unui anumit proces sau grup de procese. Cele mai cunoscute moduri de partiţionare sunt algortimii de “partiţionare fixã” şi “set echilibrat”. Avantajul înlocuirii la nivel local este independenţa pe care o are un proces în gestionare, nedepinzând de alte structuri de date globale.

5.3.Algoritmul optim de înlocuire a paginilor. Imposibilitatea realizării sale.

Algoritmul optim de înlocuire a paginilor (cunoscut şi sub denumirea de OPT sau algoritmul “clairvoyant”) lucrează în felul următor: când o paginã trebuie schimbatã, sistemul de operare înlocuieşte pagina ce va fi utilizatã cel mai târziu în viitor.

Acesta nu poate fi implementat în realitate datoritã imposibilitãţii de a calcula cu exactitate când vor fi utilizate paginile, decât în foarte puţine cazuri. Totuşi algoritmele actuale oferã performanţe aproape de nivelul optim – la prima rulare a programului, sistemul de operare reţine toate paginile accesate şi foloseşte aceste informaţii pentru organizarea lor ulterioarã (la urmãtoarele trimiteri în execuţie a programului). Este de remarcat faptul cã eficienţa maximã nu este obţinutã la prima lansare în execuţie şi depinde totodatã si de tiparul referinţelor cãtre memorie (ce trebuie sa ramânã asemãnãtor la lansãri în execuţie ulterioare).

5.4.Algoritmi de înlocuire. Tipuri. Discuţie.

- Not Recently Used -

Pentru ca un sistem de operare sa reţinã o statisticã utilã a modului în care paginile sunt utilizate, fiecare paginã are asociaţi doi biţi de status: R – setat atunci când pagina este citită/scrisã; M – atunci când pagina este scrisã. Biţii sunt reţinuţi în intrările din tabela de paginare. Este important de reţinut cã fiecare acces la memorie este însoţit de un update al celor doi biţi, fiind deci esenţial ca setarea sã fie fãcutã prin intermediul hardware-ului. Odatã ce un bit a fost setat, el rãmâne astfel pânã ce sistemul de operare îl reseteazã la zero.

Totuşi dacã hardware-ul nu dispunde de aceşti biţi, ei pot fi simulaţi în felul următor : Când un proces este început ,intrările sale din tabela de paginare sunt marcate ca inexistente

în memorie. În momentul în care se referenţiazã o paginã, este returnat, deci, un “page fault”. Sistemul de operare seteazã deci bitul R (în tabelele sale interne) şi schimbã intrarea din tabelã astfel încât sa indice pagina corect , cu modul READ ONLY. Dacã o operaţie de scriere apare ulterior atunci acelaşi mecanism este reluat, permiţând sistemului de operare sã seteze bitul M iar pagina va fi accesatã cu READ/WRITE.

Biţii M şi R pot fi utilizaţi pentru crearea unui algoritm in felul urmãtor:

Când un proces este început, ambii biţi ai fiecãrei pagini sunt setaţi la zero de cãtre sistemul de operare. Periodic bitul R este resetat, distingând astfel paginile care nu au fost referenţiate (recent) faţã de cele utilizate. La apariţia unui page fault, sistemul de operare verificã toate paginile şi le împarte în patru categorii:

- clasa 0: nereferнţiate, nemodificate
- clasa 1: nereferenţiate, modificate
- clasa 2: refererenţiate, nemodificate
- clasa 3: referenţiate, modificate

Deşi la o primã vedere clasa 1 pare imposibil de realizat, totuşi existã posibilitatea ca unei pagini din clasa 3 sã i se modifice bitul R la trecerea unei perioade de timp.

Algoritmul NRU scoate în mod aleatoriu o paginã din cea mai joasã clasã (din punct de vedere al valorii numerice) nevidã. Implicit, în algoritmul de faţã se considerã a fi mai bine sa fie schimbatã o paginã care nu a fost referenţiatã în perioada de timp decât o paginã liberã dar care este frecvent utilizatã. Avantajele NRU sunt uşurinţa în înţelegere precum şi eficienţa sa, chiar dacã nu optimã.

- First In First Out (FIFO) -

Ideea algoritmului FIFO este evidentã citind chiar numele sãu – sistemul de operare urmãreşte utilizarea paginilor cu ajutorul unei cozi; cele mai recente dispuse la sfârşit şi cele mai puţin utilizate la început. Când este necesarã înlocuirea unei pagini, se selectează pagina de la începutul cozii (cea mai puţin utilizatã) pentru a face schimbul.

Deşi FIFO este intuitiv si uşor, se comportã destul de slab în aplicaţii, astfel, fiind rar utilizat în forma sa clasicã (descrisã mai sus). Algoritmul prezintã aşa numita “Anomalie Belady”. Cu alte cuvinte, crescând numãrul de pagini pe care procesorul le poate încãrca la un moment dat (în acelaşi timp utilizând metoda FIFO) creşte probabilitatea de apariţie a page fault-urilor.

Este utilizat in cadrul sistemului de operare VAX / VMS.

- Second Chance –

Reprezintã un algoritm de înlocuire a paginilor ce are la bazã algoritmul FIFO. Se comportã sensibil mai bine decât FIFO cu foarte puţine pierderi în ceea ce priveşte viteza. Deosebirea faţã de algoritmul precedent este cã deşi se cautã sã se schimbe tot prima paginã aflatã în coadã, totuşi se face o verificare suplimentarã asupra acesteia, şi anume se verificã bitul dacã bitul R este setat. Dacã acesta este zero atunci pagina este schimbatã, altfel ea este adusã la sfârşitul cozii (deci coada poate fi privitã şi ca o coadã circularã), bitul R este resetat şi algoritmul este reluat. Astfel, orice paginã primeşte o a doua şansã. Dacã toţi biţii R sunt setaţi atunci la a doua întâlnire a primei pagini din listã, aceasta este schimbatã, deoarece acum ea are bitul de referenţiere zero.

Aşa dupã cum îi sugerează numele, algoritmul acordã o a doua şansã fiecãrei pagini: o paginã veche care a fost referenţiatã, este probabil în utilizare şi nu ar trebui schimbatã spre deosebire de o paginã nereferenţiatã.

- Clock –

Clock este o variantã de FIFO mult mai eficientã decât “Second Chance” deoarece nu presupune trecerea paginilor, în mod constant, la spatele listei deşi îndeplineşte, în mare mãsurã, aceleaşi funcţii precum S-C. Clock pãstreazã în memorie o listã circularã de pagini, indicând care este pagina cea mai “veche” din listã. Dacã apare un page fault, atunci este verificat bitul R al paginii celei mai puţin utilizate (indicatã), dacã acesta este gãsit ca fiind setat atunci se incrementeazã indicatorul de paginã. Procedura se repetã pânã este gãsitã o paginã ce îndeplineşte condiţiile de schimb.

- Least Recently Used (LRU) -

Algoritmul LRU, deşi similar în denumire cu NRU (not recently used), diferã prin faptul cã reţine o statisticã a utilizãrii paginilor pe o perioadã scurtã de timp, în timp ce NRU verificã utilizarea paginilor pe perioada ultimului tact. LRU lucreazã pe principiul conform cãreia paginile care au fost intens utilizate în ultimile câteva instrucţiuni, au probabilitate mare de a fi utilizate şi în urmãtoarele câteva instrucţiuni. Deşi în teorie este apropiat de nivelul de performanţã optimã, totuşi este greu de implementat. Existã câteva variante ale algoritmului care încearcã sã reducã din complexitatea sa sacrificând cât mai puţin din performanţã.

Cea mai complexã metodã este reprezentatã de ”metoda listei înlânţuite”, ce conţine o astfel de listã ce reţine toate paginile din memorie. La spatele listei este adãugatã pagina cel mai puţin utilizatã recent iar la începutul listei pagina cea mai frecvent utilizatã. Dezavantajul constã în faptul cã fiecare referenţiere a unei pagini duce la o modificare a listei ajungând astfel la un proces consumator de timp.

O altã metodã, ce are nevoie şi de suport hard este urmãtoarea: se presupune ca hardware-ul deţine un contor pe 64 de biţi ce este incrementat la fiecare instrucţiune. De fiecare datã când o paginã este accesatã, primeşte o valoare egalã cu cea a contorului la momentul accesului. Când o paginã trebuie schimbatã, sistemul de operare alege acea paginã care are valoare alocatã cea mai micã. Totuşi aceastã metodã nu este realizabilã datoritã hard-ului actual, care nu conţine un astfel de contor.

Un avantaj important al algoritmului LRU îl reprezintã faptul cã se pretează unei analize statistice complete. A fost demonstrat, spre exemplu, ca un LRU nu poate avea mai mult de N ori erori decât algoritmul optiml; unde N este proporţional cu numarul de pagini. Pe de altã parte LRU are tendinţa sã piardã din performanţã în multe cazuri des întâlnite de accesare a paginilor. (spre exemplu lucrul cu o aplicaţie ce realizează o buclã peste N+1 pagini, în timp ce existã N pagini reţinute de LRU, aduce cu siguranţã cate un page fault la fiecare acces) . Multe modificări în LRU, urmãresc deci sã prezicã diferitele moduri în care sunt utilizate paginile, alegând pentru cazurile în care algoritmul nu mai prezintã randament alte modalitãţi de înlocuire a paginilor.

Alte variante asupra LRU sunt aşa numitul LRU-K, ce aduce îmbunãtãţiri în ceea ce priveşte localitatea în timp. Este cunoscut şi sub denumirea de LRU-2

Algorituml ARC. Extinde LRU şi aduce îmbunãtãţiri datoritã unei mai bune organizãri a listei ce reţine utilizarea paginilor. El pãstreazã o istorie a paginilor recent schimbate şi o utilizează în alegerea proprietãţilor paginilor stocate pentru o mai bunã decizie în eventualitatea unui swap.

- Algoritmul de alegere aleatoare (Random) -

Acest algoritm înlocuieşte în mod aleator o paginã din memorie. Avantajul clar al acestui algoritm este simplitatea sa, timpul pierdut pentru alegerea paginii de schimb dupã o anumitã organizare fiind nul. În general se comportã mai bine decât FIFO iar pentru

referenţieri prin ciclări se comportă mai bine chiar decât LRU, deşi în general acesta din urmă este mai performant. OS/390 (sistem de operare creat de IBM pentru utilizarea în mainframe-uri) foloseşte LRU iar în cazul scăderii performanţei trece la utilizarea unui algoritm aleator.

- Not Frequently Used -

NFU generează mai puţine page fault-uri decât LRU în cazul în care tabela de pagini conţine pointeri nuli.

NFU are nevoie pentru funcţionare de un counter, fiecare paginã în parte având desemnat un astfel de counter, iniţial fiind setat cu valoarea zero. La fiecare clock, paginile care au fost referenţiate în acest interval vor avea counterul incrementat cu 1. Se va putea urmări în acest mod frecvenţa utilizării paginilor. Pagina cu valoarea cea mai mică a counterului va fi schimbată.

Principala problemă cu acest algoritm este aceea că urmăreşte frecvenţa de utilizare fără a ţine cont de timpul cât sunt utilizate. Aceasta va avea drept rezultat o scădere a performanţei.

5.5.Concluzii

Concluziile care se pot trage sunt următoarele:

- algoritmul optimal de înlocuire a paginilor nu poate fi implementat datorită imposibilităţii de a prezice cu exactitate ce paginã urmează să fie utilizată de procesul care rulează.
- algoritmul NRU se distinge prin simplitate.
- algoritmul FIFO nu are eficienta doritã, poate schimba pagini necesare (nu verificã daca pagina a fost referenţiatã). De asemena prezintă anomalia Belady.
- Second Chance, deşi mai performant decât FIFO rulează lent.
- Clock este o alternativă bună faţă de Second Chance deoarece păstrează performanţele sale şi în acelaşi timp ameliorează timpul necesar rulării algoritmului.
- LRU prezintă performanţe excelente dar este în general dificil de implementat.
- NFU este o aproximare destul de “brutală” a LRU; prezintă performanţe superioare doar în anumite conditii.
- pentru atingerea unor performanţe cât mai apropiate de optim este bună utilizarea mai multor algoritmi şi alegerea celui mai bun în funcţie de condiţiile de moment. (exemplu: OS/390)

5.6. Rezumat

- Din punct de vedere istoric acest subiect a fost foarte dezbătut în perioada anilor ’60; ’70.
- In prezent cerinţele de la algoritmi s-au schimbat datoritã diferenţelor dintre sistemele de operare.
- Algoritmii de înlocuire pot sa fie locali sau globali.
- Algoritmul optim de înlocuire a paginilor nu poate fi implementat în realitate datoritã imposibilităţii de a calcula cu exactitate când vor fi utilizate paginile
- Algoritmul NRU se distinge prin simplitate.

- Algoritmul FIFO nu are eficienta doritã, poate schimba pagini necesare (nu verificã daca pagina a fost referenţiatã). De asemena prezintã anomalia Belady.
- Second Chance, deşi mai performant decât FIFO rulează lent.
- Clock este o alternativã bunã faţã de Second Chance deoarece pãstrează performanţele sale şi în acelaşi timp ameliorează timpul necesar rulării algoritmului.
- LRU prezintã performanţe excelente dar este în general dificil de implementat.
- NFU este o aproximare destul de "brutalã" a LRU; prezintã performanţe superioare doar în anumite conditii.
- Pentru atingerea unor performanţe cât mai apropiate de optim este bunã utilizarea mai multor algoritmi şi alegerea celui mai bun în funcţie de condiţiile de moment. (exemplu: OS/390)

5.7.Referinte

* Exemplu de implementare a unui algoritm FIFO în limbajul de programare C (gãsirea paginii celei mai vechi) :

```
int fifo_alg() {
int save_reg,page,frame,earliest_time;
  save_reg = FTVR;
  earliest_time = INFINITY;
  do {
    FTVR++;
    if (FTVR >= FTLR) FTVR = 0;
    page = FTBR->entries[FTVR].page;
    if (page == NIL) return(FTVR);
    if (PTBR->entries[page].first_time < earliest_time) {
      earliest_time = PTBR->entries[page].first_time;
      frame = FTVR;
    }
  } while (FTVR != save_reg);
  FTVR = frame;
  return(FTVR);
}
```

unde: - PTBR este un pointer cãtre tabela de pagini.
- FTBR este un pointer cãtre tabela de frameuri
- FTVR este un pointer cãtre tabela de adrese
- FTLR este numãrul total de locatii (din spaţiul de adrese fizice) disponibile
- algoritmul returnează cea mai veche paginã acesatã

*

1) www.informit.com/
2) http://en.wikipedia.org/
3) http://gaia.ecs.csus.edu/~zhangd/oscal/PagingApplet.html - script ce prezinta simularea algoritmilor de înlocuire a paginilor.
4) http://www.sci.csuhayward.edu/

6.Probleme de proiectare pentru sisteme de paginare *Stoian Andrei*

6.1. Alocare globală vs alocare locală

Când un proces declansează un page fault (pagina accesată nu se găseşte în memoria principală), un algoritm de alocare locală a paginilor selecează pentru înlocuire o pagină care aparţine acelui proces (sau un grup de procese care împart aceeaşi partiţie de memorie). Algoritmul de alocare globală a paginilor poate selecta orice pagină din memorie pentru înlocuire.

Alocarea locală a paginilor presupune existenţa unei partiţionări a memoriei care să determine câte pagini sunt alocate unui proces sau unui grup de procese.

In general, algoritmii de alocare globală sunt mai eficienţi decât cei de alocare locală, mai ales când dimensiunea setului în lucru variază pe toată durata procesului. Daca se foloseşte un algoritm pentru alocare locală, iar dimeniunea setului în lucru creşte, va rezulta thrashing-ul. Daca dimeniunea setului scade, algoritmii locali folosesc memoria în mod ineficient. Pentru un algoritm de alocare globală, sistemul trebuie să decidă mereu câte pagini să aloce pentru fiecare proces. O metodă este de a monitoriza dimensiunea setului în lucru cu ajutorul biţilor de vârstă, dar în acest caz nu se poate spune cu certitudine dacă nu va rezulta thrashing-ul. Dimeniunea setului se poate modifica în câteva microsecunde, pe când biţii de vârstă durează câteva tacturi de ceas.

O altă metodă este existenţa unui algoritm care să aloce pagini proceselor în lucru. Se determină numarul proceselor în lucru şi se alocă pentru fiecare un număr egal de pagini.

La folosirea unui algoritm de alocare globală, este posibil să alocăm pentru început un număr de pagini proporţional cu dimensiunea procesului, dar alocarea trebuie updatată pe măsură ce procesele se execută. Astfel se foloseşte algoritmul frecvenţei page fault-urilor (PFF – page fault frequency), care stabileşte când să se mărească sau să scadă numărul de pagini alocate unui proces. Acest algoritm nu dă nici o informaţie asupra cărei pagini sa fie înlocuită când apare un fault, ci doar controlează dimensiunea setului alocat.

6.2. Controlul alocării paginilor

Chiar dacă se folosesc cei mai buni algoritmi de alocare a paginilor, se poate întâmpla să rezulte tharshing-ul. De fapt, de fiecare dată când suma tuturor seturile în lucru ale proceselor este mai mare decât capacitatea memoriei, rezulta thrashing-ul.

Acest lucru se poate observa atunci când algoritmul PFF indică existenţa unui proces care are nevoie de mai multă memorie, dar nici un proces nu are nevoie de mai puţină memorie. Situaţia în care se ajunge este destul de delicată: nu se poate aloca memorie în plus procesului care are nevoie, deoarece nu avem de unde. Trebuie să “scăpăm” temporar de anumite procese. Singura metodă pentru a realiza acestu lucru, fără a termina forţat procese, este să le mutăm pe disc (swap). Astfel, paginile rămase libere de la procesle mutate vor fi acum alocate proceselor care au mai multă nevoie de memorie.

6.3. Dimensiunea paginilor

Dimensiunea paginilor este de obicei determinată de arhitectura procesorului.

Un sistem cu dimensiune mică a paginilor utilizează mai multe pagini, astfel că tabela de pagini (page table), care este utilizată pentru a face legătura între adresele virtuale şi adresele fizice, va ocupa mai mult spaţiu. De exemplu, daca un spaţiu de adrese virrtuale de dimensiune 2^{32} este împarţit în pagini de dimensiune 4KB (2^{12}), numărul paginilor virtuale este de 2^{20} (20=32-12). Daca dimensiunea paginii este de 32KB (2^{15}), atunci sunt necesare mai puţine pagini, adica 2^{17}.

Pentru a face legătura între adresele fizice si adresele virtuale, procesoarele au nevoie de o tabela speciala, numita Translation Lookaside Buffer, sau TLB, care este "consultata" la fiecare acces de memorie. TLB nu are o dimensiune fixă, iar când nu poate completa cerere de adresă (TLB miss), tabelele de pagini trebuie cautate manual (hardware or software), pentru a se gasi adresa cautată. Daca dimensiunea paginilor este mare, atunci tabela TLB poate memora mai multe legături între adresele virtuale si adresele fizice, evitându-se astfel acele TLB misses, mari consumatoare de timp.

Rareori se întâmplă ca un proces sa aibă nevoie de un număr fix (întreg) de pagini. Astfel, ultima pagină va fi parţial ocupată, restul de memorie neocupată din acea pagină neputând fi utilizată. Astfel, dimensiunea mare a paginilor va creşte probabilitatea ca porţiuni mari din memorie să nu poată fi utilizate. Dimensiunea mică a paginilor asigură o mai bună egalitate între dimensiunea memoriei si dimensiunea totală a proceselor.

De exemplu, dacă dimensiunea paginii este de 1MB, iar un poces are nevoie de 1025KB, atunci pentru acest proces vor fi alocate 2 pagini, resultând astfel 1023KB care nu vor putea fi utilizaţi.

La transferul de pe disk în memoria principală, mare parte a timpului necesar este datorat timpilor de "găsire" a informaţiei – seek time. Din această cauză, transferuri mari şi rare sunt mult mai eficiente decât transferuri mici şi dese.

Pe baza a doi parametri – dimensiunea medie a unui proces şi dimensiunea unei inregistrări în tabela de pagini – se poate calcula dimensiunea optimă a unei pagini:

$$\dim ens.\ optima = \sqrt{2 \cdot \dim ens.\ medie \cdot \dim ens.\ inregistrare}$$

De exemplu, pentru procese de dimensiune medie 1MB, iar înregistrări în tabela de pagini de 8 bytes, dimensiunea optimă a paginilor este de 4KB.

6.4. Spaţiul datelor şi spaţiul instrucţiunilor

Instrucţiunile şi datele ocupă spaţii de adrese diferite în memorie. Astfel, există o adresa 0 în spaţiul adreselor instrucţiunilor, care pointează către o locaţie unde este memorată o instrucţiune; la fel şi pentru date.

La un calculator cu acest design, ambele spaţii de adrese (al instrucţiunilor şi al datelor) sunt paginate independent una de cealaltă. Fiecare are tabela ei de pagini, cu legături independente între adrese virtuale si adrese fizice de date şi instrucţiuni. Când este nevoie de o instrucţiune, procesorul ştie ca trebuie să folosească spaţiul instrucţiunilor si tabela de pagini a instrucţiunilor. La fel şi pentru date.

6.5. Pagini partajate

Intr-un sistem multiprogramat, este ceva normal ca mai mulţi utilizatori să ruleze acelaşi program în acelaşi timp. Este evident că este mai eficient să se utilizeze aceleaşi pagini, decât să fie mai multe copii ale acelor pagini în memorie.

Dacă există spaţii separate pentru date şi instrucţiuni, este evident că pentru a partaja programele, procesele vor folosi aceeasi tabelă de pagini pentru instrucţiuni, dar tabele de pagini diferite pentru date. Tabelele de pagini sunt structuri de date independente de tabela de procese. Fiecare proces are doi pointeri în tabela de procese: unul pentru tabela de date şi unul pentru tabela de instrucţiuni.

Când mai multe procese folosesc aceleaşi instrucţiuni, apar probleme cu paginile pe care le folosesc. Daca un proces se execută si este eliminat din memorie, toate conţinuturile paginilor care i-au fost alocate sunt şterse, iar paginile vor fi alocate altor procese. Astfel, procesele care foloseau paginile primului proces vor genera multe fault-uri, pentru a încărca în memorie paginile necesare lor. Similar, când un procesul îşi termină execuţia, este esenţial să se ştie care dintre paginile lui sunt folosite de alte programe, pentru a nu fi golite din greşeală. Se folosesc astfel structuri de date speciale care memorează care pagini sunt folosite de mai multe procese şi care sunt aceste procese.

În cazul datelor, fiecărui proces îi este alocată o tabelă de pagini, fiecare pointând căre acelaşi set de date. Nu este nevoie să sa copieye setul de date, pentru fiecare proces. Paginile de date sunt Read Only pentru fiecare proces în parte.

Atâta timp cât procesele nu modifică datele, ci doar le citesc, nu este nici o problemă. Când un proces modifică o anume data din memorie, se crează o copie a paginii, astfel ca fiecare proces are acum pagina lui de date. Acestea sunt acum Read Write, astfel că viitoare modificări ale datelor nu vor produce nici o violare în sistem.

Această strategie măreşte performanţele sistemului, deoarece paginile al căror conţinut nu este modificat nu vor fi copiate.

6.6. Eliberarea (curăţarea) paginilor

Când este nevoie de o nouă pagină, iar nu este nici o pagină goală în memorie spre a fi scrisă cu noua informaţie, una din paginile actuale trebuie mutată pe disc (swap).

Toate sistemele au un proces numit “paging daemon“, care cercetează periodic memoria şi menţine un numar suficient de pagini goale spre a fi folosite la nevoie. El selectează şi transferă pe disc, prin algoritmii specifici de alocare a paginilor, paginile care au fost modificate după ce au fost încărcate.

6.7. Interfaţa memoriei virtuale

Memoria virtuală este transparentă proceselor şi utilizatorlor, este un spaţiu mare de adrese vituale, pe un calculator cu puţina memorie fizică.

În unele cazuri, utilizatorii au puţin control asupra memoriei şi pot modifica modul de comportare al diveselor programe.

Cea mai utiliyata tehnică de management a memoriei este DSM – Distributed Shared Memory. Scopul este acela de a permite mai multor procese într-o reţea să aibă acces la acelaşi set de pagini, printr-un singur spaţiu de adrese. Când apare un fault, procedura de tratare a fault-urilor detectează în reţea ce calculator are pagina cerută şi trimite un mesaj prin care pagina este eliminată din tabela de pagini si din memoria sursă, este trimisă prin reţea celui care a cerut-o, care o va indexa intr-o tabela de pagini şi o va aloca programului care are nevoie.

Bibliografie:

1) http://en.wikipedia.org/wiki/Main_Page

2) “Virtual memory: Issues of implementation” Bruce Jacob, Trevor Mudge, 1998

3) www.patentstorm.us/
4) http://www.windowsitlibrary.com/
5) http://www.cs.mun.ca/~paul/cs3725/material/web/notes/
6) http://www.cs.jhu.edu/~yairamir/cs418/os6/
7) http://www.ginfo.ro/8_4-5/
8) http://funinf.cs.unibuc.ro/~gheorghe/curs/arhCalc/lec/l11four.pdf
9) Modern Operating Systems 2Ed - Tanenbaum

7.Probleme de implementare *Nistor Adrian*

Abstract

Sistemul de operare are deosebita sarcina de organizare a memoriei virtuale pentru rezultate mai bune in ceea ce priveşte viteza de calcul şi volumul de informaţii manipulate. In acest sens intervin o serie de probleme care trebuie sa aibă o rezolvare cât mai bună. În cele ce urmează se iau în calcul câteva aspecte importante în ceea ce priveşte problemele de implementare ale unui sistem de operare.

In primul rând sistemul de operare trebuie sa organizeze managementul paginării în fiecare din momentele în care se poate afla un proces: crearea sa, desfăşurarea în timp, „page fault”-urile şi momentul de încheiere a unui proces. Tot sistemul de operare are rolul de a trata mesajele de tip page-fault În acest sens trebuie să aibă un mecanism corespunzător care să întrerupă procesul activ, să transfere controlul rutinei de tratare a PF-ului sistemului de operare şi apoi să redea controlul procesului întrerupt. În rezolvarea page fault-ului, datorită nevoii de a re-executa instrucţiunea pentru care s-a adus în memorie pagina căutată (ce iniţial nu era găsită), este nevoie de un sistem care sa reţină instrucţiunea curentă. Astfel backup-ul instrucţiunii este sistemul care rezolvă problema. Un alt aspect care trebuie tratat este problema ca un proces să aibă nevoie de o pagină care îi este eliminată din memorie de către sistemul de operare. Pentru ocolirea acestei situaţii sistemul de operare dispune de un sistem de blocare a paginilor în memorie, care reprezintă soluţia salvatoare. Termenul de „Backing store” reprezintă procesul prin care sistemul de operare scrie pe disc datele unei pagini care este eliminată din memorie. Acest ansamblu de acţiuni, salvarea pe disc a unor pagini, respectiv extragerea de pe disc în memorie a paginilor căutate stă la baza managementului efectuat de către sistemul de operare. De asemenea tot sistemul de operare trebuie sa fie capabil sa gestioneze mesajele page fault şi cu un paginator extern.

Sistemul de operare are deosebita sarcina de organizare a memoriei virtuale pentru rezultate mai bune in ceea ce priveşte viteza de calcul şi volumul de informaţii manipulate. In acest sens intervin o serie de probleme care trebuie sa aibă o rezolvare cât mai bună.

7.1.Implicaţiile Sistemului de Operare in paginare

Exista patru cazuri legate de paginare în care sistemul de operare trebuie sa ia în considerare un set anume de acţiuni în ceea ce priveşte desfăşurarea unui anumit proces: crearea unui nou proces, desfăşurarea în timp a procesului, „page fault”-urile şi momentul de încheiere a unui proces.

Când un nou proces este creat într-un sistem de paginare sistemul de operare trebuie să determine mărimea programului şi a datelor la momentul iniţial şi să creeze o tabelă de pagini pentru acel proces. Spaţiul trebuie alocat în memorie pentru tabela de pagini şi trebuie iniţializat. Tabela de pagini trebuie să fie în memorie atunci când procesul rulează. În plus pe

hard disk trebuie rezervată memoria astfel încât atunci când o pagina este trecută pe disc să aibă unde să fie salvată. Această zonă trebuie iniţializată cu textul programului şi datele astfel încât atunci când un nou proces începe sa primească page fault-uri să se poată aduce paginile cerute de pe disc.

Când un proces este programat pentru execuţie MMU trebuie resetat pentru noul proces şi TLB golit pentru a scăpa de eventualele urme ale procesului anterior. Tabela de pagini a procesului curent trebuie sa fie cea curenta, acest lucru făcându-se de obicei prin copierea unui pointer în nişte registre hardware corespunzătoare. Opţional paginile unora din procese pot fi aduse în memorie pentru a reduce numărul de page fault-uri iniţial.

Când apare un mesaj de tip page fault, sistemul de operare trebuie să citească registrele hardware să determine care adresă virtuală a generat eroarea. Din această informaţie trebuie sa calculeze ce pagină este necesară şi s-o identifice pe disc. Trebuie apoi să găsească un spaţiu disponibil pentru pagina nouă înlăturând unele din vechile pagini folosite rar, şi să aducă noua pagină în locaţia creată. Apoi trebuie să salveze contorul de program pentru a permite re-executarea instrucţiunii indicând către pagina căutată.

Când un proces se încheie, sistemul de operare trebuie să elibereze tabela sa de pagini, paginile sale şi spaţiul de pe disc ocupat de paginile salvate pe hard disk. Dacă unele pagini sunt partajate cu alte procese paginile din memorie sau de pe disc pot fi eliberate abia când ultimul proces se termină.

7.2.Gestionarea erorilor de tip PF (Page Fault)

Tratarea Page Fault necesită un mecanism de tratare a excepţiei care să întrerupă procesul activ, să transfere controlul rutinei de tratare a PF-ului sistemului de operare şi apoi să redea controlul procesului întrerupt. PF va fi recunoscut pe parcursul ciclurilor de acces la memorie. Deoarece instrucţiunea care a cauzat PageFault *trebuie reluată*, trebuie salvat în stivă (automat) Contorul de Program aferent acesteia. Pentru asta, există un registru special, EPC (*Exception Program Counter*). Tabela de pagini necesită prezenţa unui bit „rezident” care ne arată dacă pagina respectivă este sau nu în memorie. Uneori se utilizează termenul „valid” pentru a indica prezenţa paginii în memorie. O pagină „invalidă” este astfel o pagină nerezidentă sau care are o adresă ilegală. Cu toate acestea, mai logic este să avem doi biţi în tabela de pagini: unul care să indice faptul că pagina este validă iar al doilea să arate dacă pagina este rezidentă sau nu în memorie. Întotdeauna când apare un fenomen *page-fault*, se pune problema administrării (rezolvării) acestuia. În general, paşii care se urmează sunt

- Procesul necesită o pagină ce nu este rezidentă în memorie;
- Se verifică în tabela de pagini dacă referinţa de memorie este validă sau nu;
- Dacă referinţa este validă dar pagina nu este rezidentă, se încearcă obţinerea acesteia din memoria secundară;
- Se caută şi se alocă un cadru liber (o pagină de memorie fizică neutilizată până în prezent; uneori este necesară eliberarea unei pagini de memorie);
- Se planifică o operaţie de acces la disc pentru a se citi acea pagină din memoria secundară în cadrul nou alocat;
- După scrierea paginii în memorie se modifică tabela de pagini; pagina este acum rezidentă în memorie;
- Se reporneşte instrucţiunea ce a generat *page-fault*

7.3.Backup-ul instrucţiunilor

Când este cerută adresa unei pagini care nu există în memorie, instrucţiunea în cauză va genera un mesaj de tip page-fault către sistemul de operare. Acesta rezolvă situaţia creată şi trebuie să re-execute instrucţiunea pentru a se finaliza cu succes. Problema apare datorită faptului ca de exemplu pentru o instrucţiune cu 2 operanzi sistemul de operare nu va şti ce anume a cauzat eroarea de tip page-fault: primul operand sau al doilea; Relativitatea este întărită şi de nesiguranţa valorii păstrate de contorul de program care trebuie atent modificat la re-executarea instrucţiunii după rezolvarea problemei ce generase iniţial PF-ul. În acest scop, unele procesoare sunt construite special pentru a avea o soluţie la această problemă. Metoda este prezenţa unor registre interne ascunse care au menirea de a păstra valoarea contorului de program înainte de executarea oricărei instrucţiuni. În acest fel sistemul de operare poate şti cu exactitate cum sa revină corect la „momentul" în care trebuie sa re-execute o anumită instrucţiune. Totuşi daca procesoarele nu prezintă astfel de soluţii hardware, sistemul de operare trebuie sa vină cu modificări suplimentare asupra procesului de executare a instrucţiunilor pentru a preveni potenţiala ambiguitate în cazul unei erori de tip page-fault.

7.4. Blocarea paginilor in memorie

O alta problemă care poate apărea are legătură cu alte surse de informaţii/date, de exemplu porturile I/O. Pe scurt este vorba de şansa, sau neşansa, ca o pagină corespunzătoare unui proces de aducere a datelor dintr-un asemenea port, sa fie eliminată in favoarea unui alt proces mai puţin lent. Pentru înţelegere se poate lua un exemplu posibil, nu neapărat cu şanse foarte mari de apariţie. Pentru un proces care se presupune că trebuie să citească date de la o componentă periferică legată printr-un port de tip I/O va exista un buffer cu anumite pagini in memorie. Având în vedere ca procesul se desfăşoară mai lent, sistemul de operare va „profita de moment" să permită în acest timp unui alt proces sa intre în execuţie. Presupunem că procesul al doilea generează un mesaj page-fault iar sistemul de operare trebuie să elimine din memorie o pagină pentru a aduce în loc informaţia căutată de proces. Exemplul de faţă fiind unul cu un anumit scop, luăm în considerare cazul în care SO alege pentru eliminare chiar paginile din bufferul în care primul proces iniţial aducea date de pe portul de I/O. Acest lucru ar duce la situaţia în care unele procese ar modifica datele in paginile care nu le sunt asociate. Pentru aceasta, în special în cazul porturilor I/O, soluţia găsită este ca paginile din memorie să poată fi blocate pentru ca sistemul de operare sa nu le elimine în cazul unui mesaj page-fault.

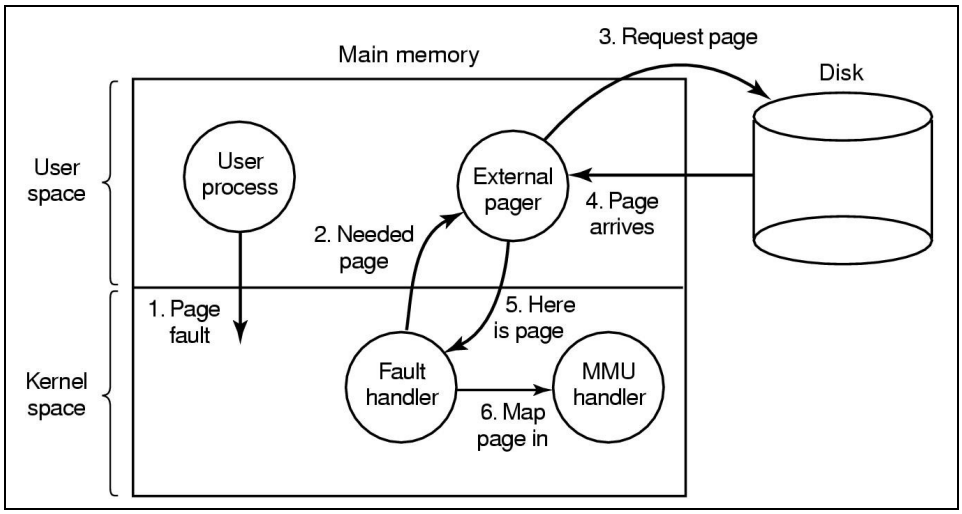
7.5. Termenul de "Backing Store"

Acest termen se referă la procesul prin care se scriu datele din memorie pe disc. Având în vedere ca SO-ul este cel care se ocupa de acest management, tot el trebuie sa aibă o bună metodă de a organiza pe disc datele corespunzătoare paginilor care „ies" din memorie. În acest sens există un spaţiu delimitat, special destinat pentru paginile proceselor care vor fi în execuţie, numit zonă „swap". Sistemul de operare trebuie sa aleagă varianta optimă prin care să permită „trecerea" paginilor din memorie in zona de swap, ţinând cont totodată şi de faptul că un proces poate avea nevoie de o zonă mai mică sau mai mare de lucru, ba chiar îşi poate modifica această dimensiune pe parcursul desfăşurării execuţiei. Din acest motiv s-au extras două situaţii prin care se poate rezolva această mică dilema. Ori să existe zone separate pe disc pentru program, date şi stivă şi sa se permită gruparea acestor zone; ori să nu se aloce nimic de la început, ci să se aloce spaţiu pe disc pentru fiecare pagină nouă care trebuie să fie salvată in zona de swap. In acest al doilea caz mai este însă necesar şi de o tabelă care sa reţină situaţia clară a paginilor salvate pe disc. Prima variantă,

paginarea în mod static a zonei de swap, presupune o corespondenţă clară de la început între paginile care trebuie scrise şi spaţiul alocat pentru aşa ceva. Al doilea caz permite adaptarea în mod dinamic a spaţiului în funcţie de necesităţile de moment.

7.6. Separarea regulilor si a mecanismului

Pentru o buna organizare în cazul sistemelor complexe e nevoie de separarea mecanismului de reguli. Principiul poate fi aplicat şi în cazul memoriei virtuale. Un exemplu simplu de separare este in figura de mai jos.

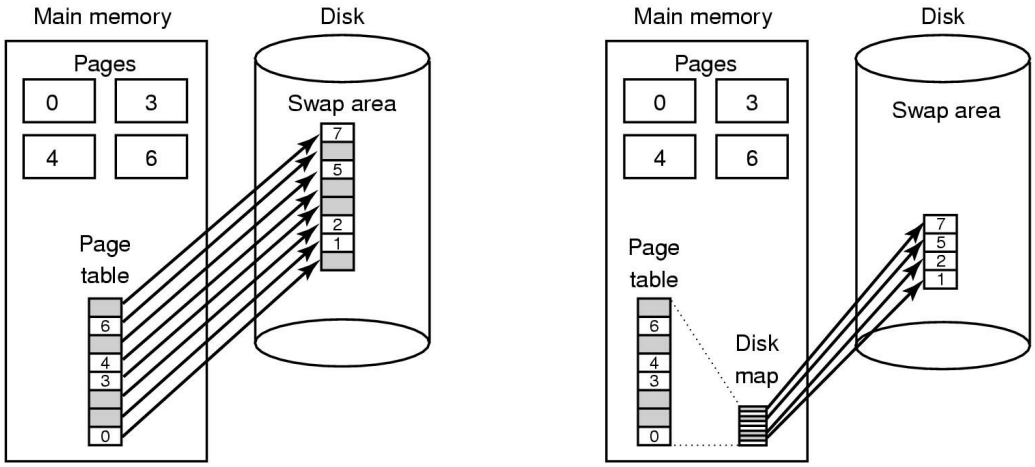


1 – O unitate de management a memoriei (MMU) la nivel scăzut
2 – O componentă de manevrare a mesajelor PageFault integrată în nucleu
3 – Un „paginator” extern ce rulează în spaţiul de utilizator.

Toate detaliile despre cum ar trebui sa funcţioneze MMU sunt încapsulate in controlerul MMU, care are un cod dependent de maşină deci trebuie rescris pentru fiecare platformă pe care sistemul de operare este instalat. Controlerul mesajelor de tip "page fault" nu este dependent de maşină şi conţine marea parte a mecanismului de paginare.

Când un proces porneşte, paginatorul extern este notificat pentru a pregăti harta paginilor de procese si sa aloce pe hard disk spaţiul necesar pentru "backing store". Cat timp procesul rulează paginatorul extern poate fi notificat în continuare.

Imediat ce procesul porneşte, pot apărea mesaje tip page fault. Controlerul determină care pagina virtuală este necesara pentru folosire şi trimite mesaj paginatorului extern "comunicând" situaţia. Paginatorul extern copiază apoi pagina în o porţiune din zona sa de adresa; transmite apoi controlerului de erori "page fault" unde se afla pagina. Controlerul de-mapează pagina din adresele paginatorului extern şi întreabă controlerul MMU unde să pună in zona de adrese a utilizatorului la locul potrivit. Abia după acestea procesul poate fi restartat.



Avantajul principal al acestui tip de implementare este axarea codului mai mult pe module şi flexibilitatea sporită. Dezavantajul, în schimb, este depăşirea repetată a graniţelor nucleului cât şi a mesajelor transmise intre diverse componente ale sistemului. În acest moment problema este controversată, dar pe măsură ce computerele devin din ce în ce mai rapide şi software.ul devine mai complex, se va trece cu vederea peste acest sacrificiu în favoarea unui soft mai deschis şi acceptabil pentru implementare.

Bibliografie:

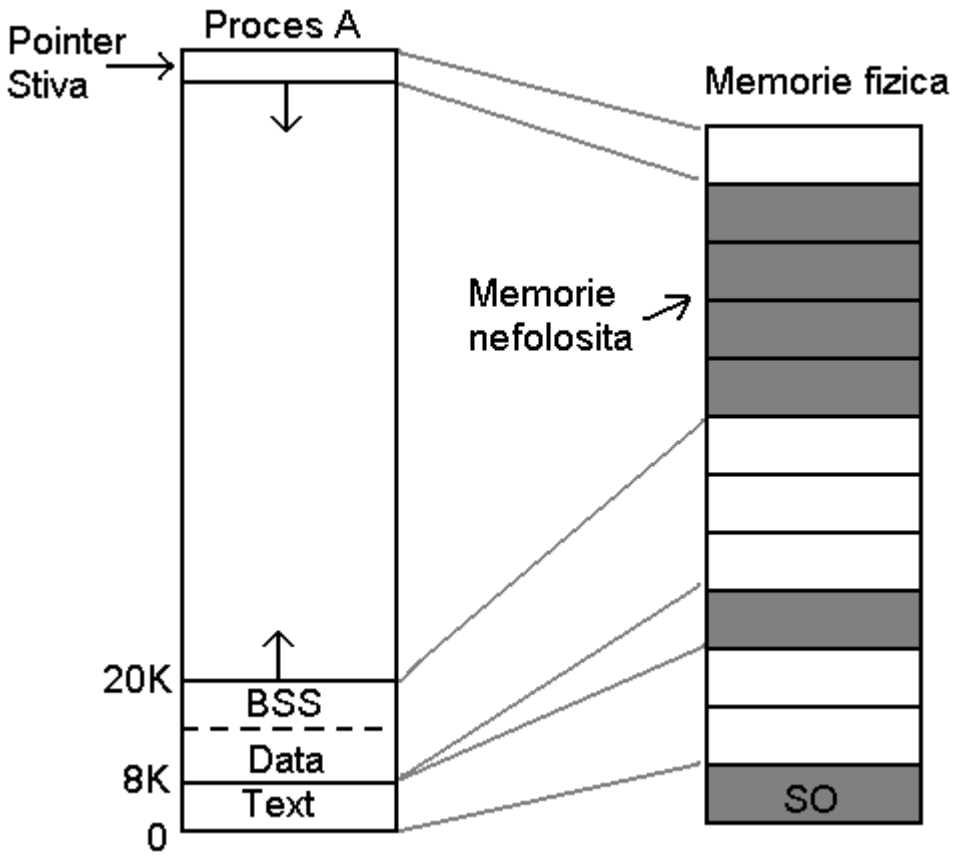
1) Modern Operating Systems 2Ed - Tanenbaum

8. Ledeanu Mihai Silviu

Spre deosebire de alte sisteme de operare, Unix-ul utilizează algoritmi foarte sofisticaţi de gestionare a memoriei pentru a folosi foarte eficient resursele de memorie disponibile. De asemenea, modelul de memorie Unix încearca să facă programele portabile şi să facă posibilă implementarea Unix pe un număr mare de maşini diferite. Modelul de memorie s-a schimbat puţin odata cu trecerea timpului, el a functionat aşa de bine încat nu s-a simţit nevoia multor îmbunătăţiri ulterioare.

8.1.Concepte fundamentale

Fiecare proces Unix are un spaţiu de adrese conţinând trei segmente: text, data şi stivă. Un exemplu de adrese de proces este Procesul A din figură:

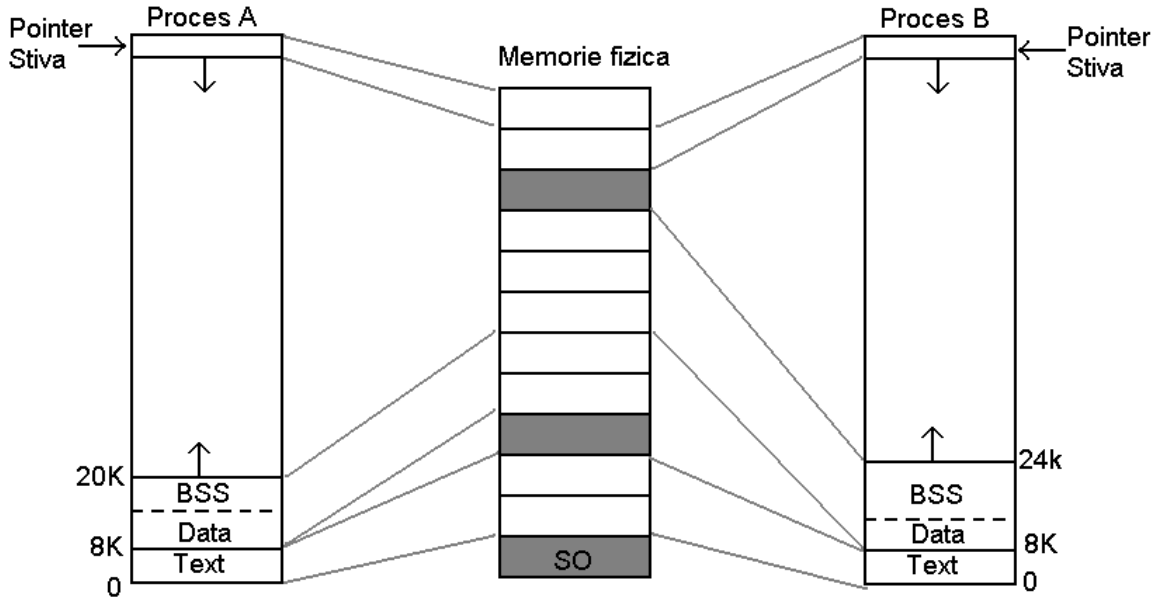


Segmentul de text conţine intructiunile maşinii ce formează codul executabil al programului. Acesta e produs de compilator şi asamblor translatând programul dintr-un limbaj în cod maşină. Acesta este de obicei read-only, programele care se automodifică au ieşit din circulaţie înca din anii 50, fiind prea greu de inteles şi de facut debug.

Segmentul de date conţine variabilele programului, şiruri, tabele şi alte date folosite de program. Acesta e format din două părţi: data iniţializată şi data neiniţializată (numită BSS). Partea cu data iniţializată conţine variabile şi constante ale compilatorului care au nevoie sa fie iniţializate când programul e pornit.
Spre deosebire de segmentul de text care nu se poate schimba, segmentul de data se schimbă, programele modificând tot timpul variabilele lor. Multe programe au nevoie să şi aloce dinamic spaţiu în timpul execuţiei. Unix permite segementului de date sa le mareasca şi micsoreze pe masură ce memoria e alocată şi dealocată. Există un system call, *brk*, ce permite unui program să seteze mărimea segmentului de dată, iar procedura *malloc*, din C/C++ e deseori folosită de programe pentru alocarea memoriei

Al treilea segment e cel de stivă. Aceasta porneşte din varful spaţiului de adrese virtuale şi creşte in jos către 0. Daca stiva creşte mai mult de partea de jos a segmentului de stivă, este generat un fault de hardware iar partea de jos a segmentului e coborată cu o pagina (adica este mărită cu o pagină). Programele nu gestionează marimea segmentului de stivă. La inceputul execuţiei unui program, stiva sa nu este goală, ci conţine variabilele shell-ului şi linia de comanda scrisă în shell pentru a rula programul. Astfel, programul are acces la argumentele sale.

Cand doi utilizatori rulează acelaşi program, ar fi ineficient să se pastreze două copii ale segmentului text. Astfel Unix suportă segmente de text partajate (împarţite). În figură vedem o posibilă împarţire a memoriei în care două procese (A şi B) impart acelaşi segment de text.



Segmentele de dată şi stivă nu sunt niciodată partajate, decat după un *fork* (procedeu prin care un proces işi crează o copie a sa, numita proces-copil) iar atunci numai acele pagini nu sunt modificate.

Pe unele calculatoare, hardware-ul suportă spaţii de adrese separate pentru program şi pentru data. Unix-ul foloseste aceasta proprietate, daca este disponibila. Astfel spaţiile de adrese disponibile sunt dublate. De exemplu, pe un calulator cu adrese pe 16 biţi, vor fi 2^{16} biţi de adrese pentru program şi 2^{16} pentru segmentul de dată şi stivă.

Multe versiuni de Unix suportă memory-mapped files (fişiere cu memoria “mappată”). Astfel că e posibil sa faci o hartă (map) al unui fişier într-o porţiune a spaţiului de adrese a unui proces, aşa că fişierul poate să fie scris şi citit ca şi cum ar fi un şir de octeţi în memorie. Daca mai multe procese mapează şi accesează acelaşi fişier, aceeaşi memorie reală şi pagini vor fi partajate tuturor proceselor. Aceasta dă voie mai multor programe să îşi partajeze data fără să fie nevoite să ţina mai multe copii ale datelor în memorie.

8.2.Implementarea gestiunii de memorie în Unix

La începuturi, toate versiunile de Unix erau bazate pe swapping: când există mai multe procese ce pot fi ţinute în memorie, unele dintre ele sunt înlocuite („swappate”) pe disc. Un proces este tot timpul swappat în întregime. Astfel ca un proces se afla ori in memorie, ori pe disc.
Odata cu apariţia 3BSD (o distrubuţie Unix - Berkeley Software Distribution, versiunea 3BSD apare în 1979) universitatea Berkeley a adăugat şi paginarea pentru a face faţă programelor tot mai mari ce erau scrise. Acum toate versiunile de Unix folosesc paginarea.

8.2.1.Swapping

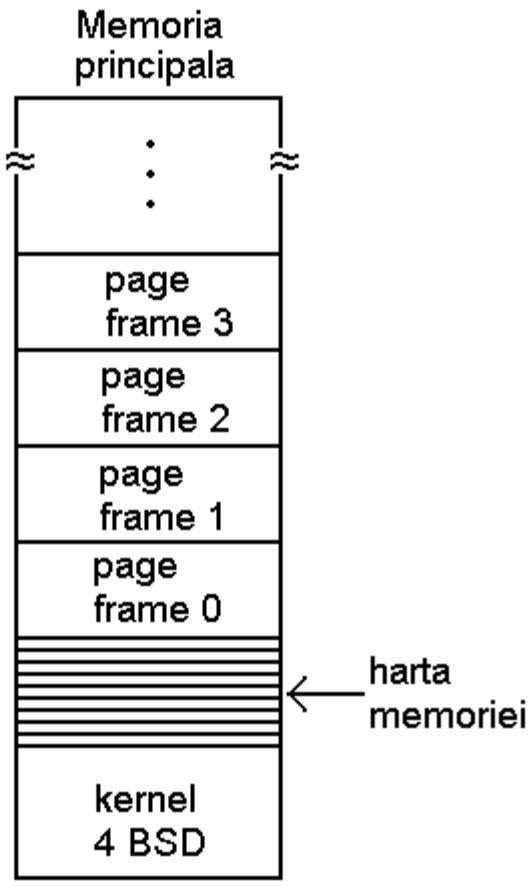
Procesul în care unele pagini din memorie sunt mutate pe disc, şi altele mutate in memorie, se numeste swapping. Mutarea intre memorie şi disc este facuta de **swapper**. Mutarea (swappingul) din memorie catre disc este facuta cand kernelul ramana fara memorie libera din cauza unuia dintre evenimente:

- Un system call *fork* are nevoie de memorie pentru un proces-copil
- Un system call *brk* are nevoie sa isi exitinda segemenul de data.
- O stiva creste şi ramanae fara spaţiu alocat.

De asemenea, cand trebuia sa fie aduc in memorie un proces ce a stat pe disc multa vreme, de obicei un alt proces trebuia sa fie el eliminat pentru a face loc. Pentru a decide care proces sa fie dus pe disc, swapperul cauta procesele blocate, in asteptare (de exemplu pentru un periferic), fiind mai bine de eliminat un proces ce nu ruleaza decat unu care ruleaza. Din cele gasite este eliminat cel cu prioritatea mai mare. Daca nici un proces nu este blocat, atunci se cauta un alt proces dupa aceleasi criterii.
O data la cateva secunde swapperul verifica daca unul dintre procesele duse pe disc este gata de a reveni in memorie. Daca exista, este selectat cel care a ramas pe disc cel mai mult timp. Swapperul mai verifică şi dacă swap-ul va fi unul uşor sau unul greu. Un swap uşor presupune că deja există destulă memori liberă pentru a fi adus procesul in memorie, un swap greu presupune că mai trebuie eliberată memorie pentru a fi adus procesul in memorie.

8.2.2.Paginarea

Odată cu apariţia 3BSD de la Berkeley, paginarea a fost introdusă in aproape toate versiunile ulterioare de Unix.
Aici voi descrie principiul paginării distribuţiei 4BSD (apărută în noiembrie 1980) unde ideea este că un proces nu trebuie neapărat să fie adus în întregime în memorie ca să poata rula. Este nevoie numai de structura utilizatorului şi de tabela paginii. Dacă acestea au fost aduse de swapper în memorie, atunci se spune că procesul este în memorie şi poate rula. Paginile segmentelor de text, de date şi de stivă sunt aduse in memorie dinamic, pe rând atunci când se face o referinţă către una din ele.
Paginarea este implementată parţial de kernel şi parţial de un proces numit daemonul paginii. Daemon-ul paginilor este pornit periodic şi verifică dacă are sarcini de făcut. Dacă el descoperă că numărul paginilor de pe lista de pagini libere este prea mic, atunci el începe sa mai elibereze pagini.
Memoria principală, la 4BSD este organizată ca în figură, având trei parţi: primele două parţi, kernelul şi core map (harta memoriei) sunt ţinute permanent în memorie. Restul memoriei este divizată în cadre de pagini (page frame), fiecare putând conţine o pagină de text, data sau stivă, o pagină de tabela, sau să fie liberă.



Harta memoriei (core map) conţine informaţii despre conţinutul cadrelor de pagini. Intrarea 0 din harta memoriei descrie cadrul de pagina 0, intrarea 1 descrie cadrul de pagină 1 şi ala mai departe. Cu cadre de 1 KB şi intrari în harta memoriei de 16 octeţi, mai puţin de 2% din memorie este ocupată de harta memoriei.
Când un proces este pornit, este posifil să primeasca un page fault, pentru că una sau mai multe din paginile sale nu se găseşte în memorie. Dacă are loc un page fault, sistemul de operare ia primul cadru de pagini din lista de cadre libere, o elimina din listă li citeşte pagina de care este nevoie. Dacă lista de pagini libere este goală, procesul este suspendat până când daemon-ul de pagini eliberează o pagină

8.2.3.Algoritmul de înlocuire a paginilor

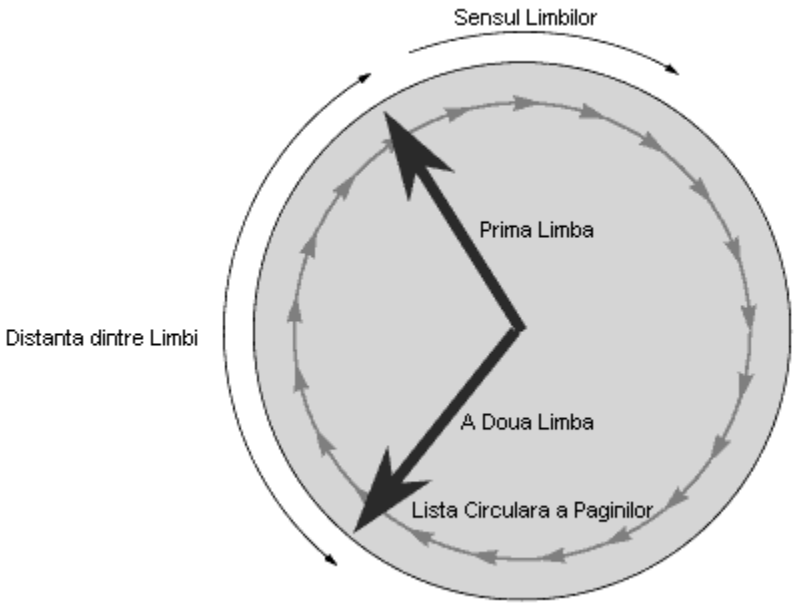
Algoritmul de înlocuire a paginilor este executat de către daemon-ul de pagini. Acesta la fiecare 250 de milisecunde verifica dacă numărul de cadre de pagini libere este cel puţin egal cu un parametru al sistemului numit *lotsfree* (de obicei având valoarea unui sfert din memorie). Dacă sunt libere un număr insuficient de pagini, atunci daemonul incepe sa transfere pagini din memorie catre disc, pâna ce se va ajunge la un număr de lotsfree de pagini libere. Dacă daemonul verifică că un număr mai mare de lotsfree de pagini sunt libere atunci se întoarce în modul de sleep. Dacă un calculator are multă memorie şi puţine procese active, atunci daemon-ul va sta cea mai mare parte a timpului in modul sleep.

Daemon-ul de pagini foloseşte o versiune modificată a algoritmului ceasului (clock algorithm). Algoritmul ceasului scanează circular toate cadrele, asemenea unei limbi de ceas care se

primba de-a lungul tuturor orelor. La prima trecere, biţii de folosire a cadrelor sunt setaţi zero. La a doua trecere, toate cadrele care nu au fost accesate de la ultima trecere a “limbii” sunt trecute în lista de cadre libere, după ce au fost mutate pe disc (mutarea pe disc are loc numai daca acestea sunt “murdare”, adică dacă au fost modificate de când au fost aduse in memorie. Dacă nu au fost modificate, atunci nu are sens sa fie remutate pe disc.)

Algoritmul acesta a fost folosit iniţial de distribuţiile Unix ale universităţii Berkeley, dar mai târziu s-a descoperit că pentru memorii mari, trecerea peste toate paginile dura prea mult. Aşa că algoritmul a fost modificat în algoritmul ceasului cu două limbi (*two-handed clock algorithm*).

În acest algoritm daemon-ul ţine doi pointeri către harta memoriei (cele două “limbi” ale ceasului. Prima seteaza biţii de folosire zero, iar a doua verifică biţii de folosire şi mută paginile nefolosite pe disc.După ce amandouă limbile îşi fac verificările, fiecare avansează cu o poziţie. Depinzand de deschiderea dintre cele doua limbi, paginile vor avea mai mult sau mai puţin timp la dispoziţie sa nu fie accesate până sunt mutate pe disc.



Algoritmul Ceasului cu Doua Limbi

De fiecare dată când daemon-ul este pornit limbile avansează mai puţin ce un cerc întreg, numărul de avansări ale lor depinde de cât timp e nevoie pentru a ajunge ca numărul de pagini libere sa ajungă la lotsfree. Dacă sistemul observă că rata de pagini este prea mare şi numărul de pagini libere este tot simpul sub lotsfree, atunci sweeperul începe să înlăture procese din memorie.

Feluri de memorii:

- Principală – Memoria fizica RAM aflată pe placa de bază a calculatorului. Mai este numită şi memorie reală. Aceasta nu include memoria video, cache-ul procesorului sau alte memorii periferice.

- Sistemul de fişiere – Memoria discurilor accesibilă prin pathname-uri (Pathname-urile sunt o secvenţă de simboluri şi nume care identifică un fişier). Aceasta nu include spaţiul de swap sau alte spaţii de stocare ce nu sunt adresabile prin pathname-uri. Nu include nici toate sistemele de fişiere de reţea.
- Spaţiul de Swap – Memoria discurilor ce stochează data care nu e în memoria reală sau a file system-ului. Spaţiul swap este cel mai eficient când e pe un disc separat sau partiţie separată, dar cateodată este doar un fişier mare din sistemul de fişiere.

Memoria Sistemului de operare foloseşte:
- Kernel – Spaţiul de memorie privat al Sistemului de operare. Acesta se afla tot timpul in memoria principală.
- Cache – Memoria care e folosită pentru a ţine elemente ale sistemului de fişiere şi alte operaţii de intrare/ieşire. A nu fi confundat cu cache-ul procesorului sau al hard disk-urilor, ce nu fac parte din memoria principală.
- Memorie Virtuală – Totalitatea spaţiului de memorie adresabilă a tuturor proceselor ce rulează pe maşină. Locaţia fizică se poate afla pe oricare dintre cele trei tipuri de memorie.

Memoria Proceselor foloseşte:
- Data – Memoria alocata şi folosită de program.
- Stiva – Stiva de execuţie a programului (gestionată de sistemul de operare).
- Memoria mappată – Conţinutul fişierului adresabil in spaţiul de memorie al procesului.

8.3.Rezumat:

1. Concepte fundamentale
Modelul de memorie la Unix a suferit puţine schimbări odată cu trecerea timpului.
Fiecare proces Unix are un spaţiu de adrese ce conţine trei segmente: segmentul de text, segmentul de date şi segmentul de stivă.
Segmentul de text conţine instrucţiunile maşinii ce formează codul executabil al programului. Este read-only.
Segmentul de date conţine variabilele programului. La rândul său acesta conţine datele iniţializate şi datele neiniţializate. Acest segment se poate schimba.
Segmentul de stivă, la pornirea programului conţine parametrii acestuia. Stiva creşte în jos.
Segmentele de text pot fi partajate.

2. Implementarea gestiunii de memorie în Unix

2.1 Swappingul
La început, distrubuţiile de Unix foloseau numai swappingul. Acessta presupune mutarea (swappingul) unui întreg program în memorie sau pe disc.
Swappingul este facut de un scheduler numit swapper.

2.2 Paginarea
Odată cu trecrea timpului, în sistemele Unix a fost introdusă şi paginarea, deoarece procesele creşteau din ce în ce mai mult şi ocupau tot mai mult spaţiu din memorie.
Ideea paginării este ca un proces nu trebue adus cu totul în memorie pentru a putea fi rulat.
Paginarea este efectuala parţial de kernel şi parţial de daemonul de pagini.

2.3. Algoritmi de înlocuire a paginilor

Algoritmul de înlocuire a paginilor este executat de daemon-ul de pagini care are grijă să fie cel puţin lotsfree pagini libere în memorie.
La început s-a folosit algoritmul ceasului, un pointer verifica lista harţii memoriei circular, iar dacă la a doua verificare pagina nu fusese activată atunci ea era scoasă din memorie. O pagină este rescrisă pe disc numai dacă a fost schimbată între timp, de când a fost adusă în memorie.
Pentru că parcurgerea întregii memoriei dureaza mult, s-a introdus algoritmul ceasului cu două limbi. Prima limbă setează progina ca nefolosită, iar a doua verifică dacă între timp a fost folosită, dacă nu atunci este scoasă din memorie.

Bibliografie:

1) **Andrew Tanenbaum, Operating Szstems, Design and Implementation**
2) http://en.wikipedia.org
3) http://www.cpushack.net
4) http://www.freebsd.org
5) notite curs SO cu St.St