**Universitatea Politehnică din București**

**Facultatea de Electronică, Telecomunicații și Tehnologia informației**

**Implementarea gestiunii memoriei**

Coordornator știintific

Ștefan Stăncescu

Studenti:

Chirețu Teodor-grupa 433A

Costea Cosmin-grupa 433A

Sănduleac Cristian-grupa 433A

**Cuprins:**

**1.Paginarea în Linux și Windows –Chirețu Teodor, grupa 433A**

* 1. **Introducere………………………………………………………1**
  2. **Paginarea Memoriei…………………………………………….2**
  3. **Paginarea Memoriei în sistemele de operare……………….....2**
     1. **Swaping & Paging................................………………....2**
     2. **Ferranti Atlas…………………………………………...3**
     3. **Windows 3.x și Windows 9.x…………………………..4**
     4. **Windows NT……………………………………………5**
     5. **Unix și sisteme de operare asemănătoare.....................5**
  4. **Comparație între Windows și Linux…………………………...6**

**2.Gestionarea memoriei fizice: algoritmul prietenului la Linux –Sănduleac Cristian, grupa 433A**

**2.1 Introducere……………………………………………………………..11**

**2.2 Gestiunea memoriei fizice în Linux…………………………………..12**

* 1. **Alocarea și dealocarea paginilor de memorie ………………………14**

**2.3.1 Alocatorul de zonă ( the zone allocator)……………………15**

**2.3.2 Algoritmul buddy (algoritmul „prietenului”)……………...16**

**2.3.3 Alocatorul slab………………………………………………17**

**3. Subsistemul de gestiune a memoriei fizice la Windows – Costea Cosmin, grupa 433A**

* 1. **Introducere…………………………………………………………18**
  2. **Utilizarea memoriei………………………………………………..19**
  3. **Rolul subsistemului de gestiune…………………………………..20**
  4. **Localitatea programelor………………………………………...…21**
  5. **spațiul de adresă al unui proces…………………………………..22**
  6. **Alocarea/Dealocarea memoriei……………………………………24**
  7. **Metode clasice de alocare a memoriei…………………………….25**
  8. **Memoria primara…………………………………………………..27**
  9. **Memoria de masa…………………………………...…………......27**
  10. **Considerații privind costul și performanta……………………….28**
  11. **Lucru cu memoria – Probleme………………………………....…29**

**4. Bibliografie……………………………………………………………….32**

**1.Paginarea în Linux și Windows –Chirețu Teodor , grupa 433A**

* 1. **Introducere:**

Prin memorie întelegem dispozitive folosite pentru a stoca informații într-un calculator. Termenul de memorie primară este utilizat pentru dispozitivele de stocare ce funcționeaza la viteză mare (spre exemplu memoria RAM). Memoria secundară reprezintă dispozitivul de stocare ce funcționeaza la o viteză de accesare mai lentă dar poate stoca cantitați mult mai mari de informații. Dacă se dorește, memoria primară poate fii stocată în memoria secundară printr-o tehnică de management numită “memorie virtuală”.

Pe scurt memoria reprezintă locul în care sunt stocate programe și date ce urmează să fie prelucrate de către procesor.

Prin “gestiune a memoriei” înțelegem procesul de control și coordonare a memoriei calculatorului, atribuirea unor porțiuni numite blocuri pentru diferite programe în scopul de a optimiza performanța generală a sistemului. Gestionarea memoriei se regăsește în hardware, în SO (sistemul de operare), în programe și aplicații.

Sistemul de gestiune a memoriei în cadrul unui sistem de operare este folosit și de alte subsisteme precum: sistemul de gestiune al fișierelor, sistemul de I/O, ceea ce îi oferă o importanță aparte în buna funcționare a unui sistem.

*1.*

* 1. **Paginarea Memoriei:**

În sistemele de operare aflate pe calculatoare, paginărea este una dintre schemele de gestiune a memoriei prin care un calculator stochează și retrage datele din memoria secundară pentru a fi utilizate în memoria principală. în schema memoriei de paginăre, sistemul de operare preia datele din memoria secundară în blocuri de aceași dimensiune numite pagini.

Principalul avantaj al paginării peste segmentarea memoriei este că permite spațiului de adrese al unui proces fizic să poată fi găsit nu numai în vecinătatea locației de memorie precedente.

Înainte ca schema de paginăre să fi intrat în folosință, sistemele au trebuit să “încadreze” programe întregi in locații de memorie vecine, fapt ce a cauzat diverse probleme de stocare și fragmentare.

Paginarea este o parte importantă a implementării memoriei virtuale în majoritatea sistemelor de operare de uz general contemporane, permițându-le să folosească memoria secundară pentru datele care nu se încadrează în memoria cu acces aleator (RAM).

* 1. **Paginarea memoriei în diverse sisteme de operare:**
     1. **Swapping & Paging**

**Fișierul Swap**

Menționat atât ca un fișier pagină sau un fișier de paginăre, un fișier de swap este un fișier stocat pe hard disk-ul calculatorului si este folosit ca o locație temporară pentru a stoca informații ce nu sunt utilizate la acel moment de timp de către memoria RAM a calculatorului.

Prin utilizarea unui fișier de swap, un calculator are capacitatea de a utiliza mai multă memorie decât cea instalată fizic. Cu toate acestea, utilizatorii ce nu dețin suficient spațiu pe hard disk pot observa cum calculatorul funcționează mai lent din cauza incapacității de a se extinde a fișierului de swap.

*2.*

Este perfect normal pentru fișierul de swap sau fișierul pagină să crească în dimensiuni, uneori acesta crește cu mai multe sute de Mb în decursul timpului.

In funcție de sistemul de operare, acest fișier poate avea mai multe nume, spre exmplu, pentru Windows NT/2000/XP/Vista/7 are numele PAGEFILE.SYS,

Pentru Windows 95/98/ME are numele WIN386.SWP iar pentru Windows 3.x are numele 386PART.PAR

**Swapping**

Sistemele de operare Unix folosesc memoria virtuală astfel încât zonăde disc (spațiu swap) este folosită ca o extensie a memoriei fizice pentru depozitare temporară atunci când sistemul de operare încearcă să țină evidența proceselor care necesită mai multă memorie fizică decât cea disponibilă. Când se întâmplă acest lucru spațiul de swap este folosit pentru swapping și paginăre.

**Paging**

Paginarea se folosește atunci când segmente individuale de memorie, sau pagini, sunt mutate în sau din zonăde swap. Când memoria nu este suficientă, mici porțiuni ale unui proces sunt mutate pentru a elibera spațiu de memorie. Segmentele sunt alese pentru a fi mutate în cazul în care nu au fost utilizate recent. Când procesul încearcă din nou să utilizeze acest segment apare o eroare de pagină și procesul se suspendă până când segmentul este returnat în memorie. O eroare de pagină este în mod normal returnata prima dată când un program este pornit, deoarece nu va fi în memorie. Este apoi „paginat” în sistemul de fișiere local sau la distanță.

* + 1. **Ferranti Atlas**

Primul computer care a sprijinit paginărea memoriei fost Atlas, dezvoltat în comun de către Ferranti, Universitatea din Manchester și Plessey. Masina a avut o memorie asociativa cu o intrare pentru fiecare pagină de 512 cuvinte .

*3.*



Figura 1 sistemul Ferranti Atlas

* + 1. **Windows 3.x și Windows 9.x**

Paginarea a fost o caracteristică Microsoft Windows înca de la Windows 3.0 în 1990. Windows 3.x creează un fișier ascuns numit 386SPART.PAR sau WIN386.SWP pentru utilizare ca fișier de swap. Acesta este în general găsit în directorul rădăcină, dar poate apărea și în altă parte (de obicei în directorul Windows). Dimensiunea acestui fișier depinde de cât de mult spațiu de swap are sistemul. Dacă utilizatorul mută sau șterge acest fișier, la următoarea pornire a Windows-ului va primi un mesajul de eroare "Fișierul de swap este permanent corupt". Utilizatorului i se va solicita să aleagă dacă să șteargă sau nu fișierul (daca există sau nu există).  
  
 Windows 95, Windows 98 și Windows Me folosesc un fișier asemănător, iar setările se află în panoul de control → Sistem → Performance → Virtual Memory. Windows setează automat dimensiunea fișierului pagină pentru a începe de la 1,5 × dimensiunea memoriei fizice și să se extindă până la 3 × dimensiunea memoriei fizice dacă este necesar. Dacă un utilizator execută aplicații care consumă o cantitate mare de memorie pe un sistem cu memorie fizică redusă, este de preferat ca acesta sa stabilească manual dimensiunea memoriei swap la o valoare mai mare decât cea implicita.

*4*.

* + 1. **Windows NT**

Fișierul folosit pentru paginăre în familia Windows NT este pagefile.sys. Locația implicită a fișierului pagină este în directorul rădăcină al partiției pe care este instalat Windows. Windows poate fi configurat pentru a utiliza spațiu liber de pe orice unități disponibile pentru swap. Cu toate acestea, este necesar ca partiția de sistem sa detină un pagefile pentru cazul în care sistemul este configurat să scrie fie kernel, fie erori de memorie după o fereastra albastra. Windows utilizează fișierul de paginăre ca depozit temporar pentru erorile de memorie. Când sistemul este repornit, Windows copiază eroarea de memorie din fișierul de paginăre într-un fișier separat și eliberează spațiul care a fost folosit în fișierul de paginăre.

* + 1. **Unix și sisteme de operare asemanatoare**

Sistemele Unix și alte sisteme de operare asemanatoare, folosesc termenul "swap" pentru a descrie atât actul de mutare al paginilor de memorie între RAM și hard disk, cât și regiunea de pe disc unde paginile sunt stocate . În unele dintre aceste sisteme se obisnuiește să se dedice o întreaga partiție a unui hard disk pentru swap. Aceste partiții sunt numite partiții de swap. Multe sisteme au un întreg hard disk dedicat swap-ului, separat de unitatea de disc care conține doar o partiție de swap. Un hard disk dedicat swapping este numit "unitate de swap" sau o "unitate zero" sau un "disc zero". Unele dintre aceste sisteme suporta numai swap-ul la o partiție de swap, altele susțin swapul de fișiere.

*5.*

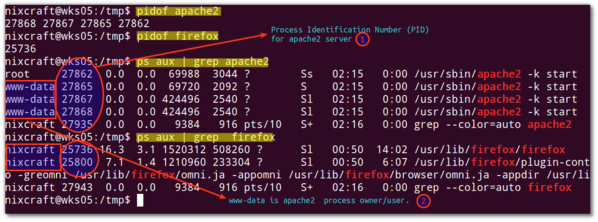
### Comparație Linux și Windows

**Process Control**

Un proces reprezintă un program în execuție, o instanța a unui program. Sistemul de operare alocă resurse utilizatorului prin procese.

In Linux se creează un context special. Acesta conține informații necesare sistemului de operare pentru rularea programuli ca și cum ar fi singurul program ce ruleaza în sistem.

Linux conține o tabelă de procese prin intermediul căreia se ține evidenta proceselor. În această tabelă găsim informații despre fiecare proces: procesul părinte (ppid), utilizatorul care a lansat procesul (uid), grupul care deține procesul.



**Figura 2. Găsirea PID-ului browserului Firefox**

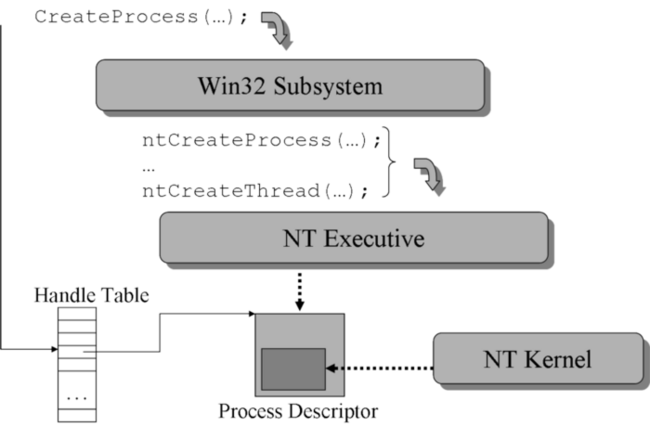
*6.*

Sistemul de operare pune la dispoziție apeluri pentru crearea unui nou process (exemplu: fork, exec), “exit” pentru terminarea unui proces, “wait” pentru un proces în asteptare și alte apeluri utile.

În cazul Windows, crearea unui proces se realizează apeland funcția API “CreateProcess”.

Funcția “CreateProcess” va returna o valoare nenulă dacă a fost executată cu succes, iar în caz contrar va returna 0.

În Windows sunt mai multe moduri de a termina un proces, dar singurul care trimite notificari modulelor atasate procesului este functia “ExitProcess”.



**Figura 3. Crearea unui process în windows**

În exemplul de mai sus se creează mai întai doua variabile, de tip 'STARTUPINFO', respectiv 'PROCESS\_INFORMATION' (aceasta din urmă va fi folosit pentru a obține informații despre procesul ce urmează sa fie pornit). Pentru a se asigura că aceste structuri conțin date valide, înainte de apelul funcției 'CreateProcess', se setează la zero întreaga zonă de memorie asociată structurilor, mai putin câmpul 'cb' din variabila de tip 'STARTUPINFO', care trebuie sa indice dimensiunea în octeți a structurii.

*7.*

Se verifică apoi dacă programul a primit argumente de intrare dupa care urmeaza apelul funcției “CreateProcess” , care creează un nou proces. Acest apel este inclus într-o instrucțiune if, pentru a se verifica dacă operația de pornire a unui nou proces a eșuat. Semnificația argumentelor primite de functia “CreateProcess” este comentată și în interiorul codului sursă: spre exemplu primul argument reprezintă comanda ce va fi rulată. În cazul în care s-a produs o eroare, functia “GetLastError()” este folosită pentru a obtine codul de eroare corespunzator. Urmeaza apelul funcției “WaitForSingleObject”, unde este folosită structura de tip PROCESS\_INFORMATION, pentru a obține handle-ul procesului ce a fost creat mai devreme. Această funcție blochează execuția programului până când se încheie procesul specificat. În final, sunt închise resursele asociate procesului fiu creat în interiorul programului cu ajutorul funcției “CloseHandle”.

**Linux**

În sistemul Linux un proces se creează cu ajutorului apelului fork(). Prin acest apel se creeaza o copie a procesului apelant și ambele procese, cel nou creat și cel apelant, își vor conținua executia cu urmatoarea instrucțiune ce urmează după apelul funcției fork.

Un proces nou se creează de către un proces deja existent (exceptie făcând procesul 0 care este creat de către sistemul de operare).

În Linux fiecare proces are un identificator de proces (PID). Acesta reprezintă un număr pozitiv și este atribuit de sistemul de operare. Prin apelul sistem

“getpid” un proces poate sa afle PID-ul părintelui. PID-ul părintelui nu se poate modifica, adică un proces nu își poate schimba el însuși părintele. Acest lucru se poate întampla totuși o singura data, dar este realizat de către sistemul de operare.

*8.*

In Windows există un planificator de procese numit ProcessScheduler . Planificatorul de porcese decide ce proces urmează să fie executat.

In Windows procesele și thread-urile conțin niște valori numite prioritati de planificare. Acestea pot lua valori intre 0, ceea ce reprezintă cea mai slaba prioritate și 31ce reprezintă cea mai puternica prioritate.

Pentru protecția memoriei Windows-ul blocheaza atacuri ale anumitor procese asupra altora.

In Windows există un proces de prevenire a atacurilor numit Address Space Layout Randomization(ASLR). Acesta realizeaza o radomizare a adreselor unde obiectele sunt scrise în spațiul de memeorie virtual al unui proces. Astfel unui atacator i-ar fi mai dificil sa gaseasca adresa de memorie.

**File Management**

Windows-ul gestionează fișierele prin obiecte de fișiere(file objects), pointeri de fișiere(files pointers) și file handlere.

In Windows avem stream-uri care conțin informații despre fișiere. Fiecare stream este asociat unui fișier. Odata cu crearea unui fișier se creeaza și un stream nedenumit (este creat default) care stochează toate informațiile scrise în fișier, atata timp cât acesta este deschis. Se pot crea și stream-uri adiționale.

*9.*

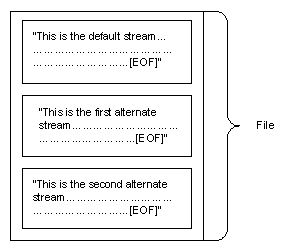


Figura 4. Un fișier cu stream-ul creat default și alte două stream-uri create adițional

În Linux, toate fișierele sunt accesate ca facând parte dintr-un arbore. Numele fișierelor și directoarelor sunt fie absolute (daca incep cu “/”) sau relative (nu incep cu “/”).

In Linux putem crea, muta și șterge fișierele cu ajutorul comenzilor~~:~~ cp, mv și rm.

Sistemul de fișiere din Linux permite crearea legăturilor la fișiere. Legatura (link) este vazută ca un fișier cu nume propriu, dar de fapt referă la un alt fișier existent pe disc. Astfel orice operatie care se execută asupra link-ului va avea efect și asupra fișierului indicat de acesta (mai putin ștergerea; în cazul ștergerii efectul depinde de tipul legăturii respective).

**Device Management**

Driverele nu sunt la fel în Linux și Windows.

In Linux majoritatea device driver sunt extensii ale kenel-ului. Aici putem gasii informații despre drivere folosind comenzi ca lspci și lsusb.

*10.*

In Windows, Device Manager este o extensie a Microsoft Management Console care asigură o organizare a tuturor dispozitivelor hardware instalate în calculator.

Device Manager-ul în Windows poate fi accesat prin diferite metode cum ar fi din Control Panel sau din Command Prompt.

**2.Gestiunea memoriei în Linux –Sănduleac Cristian-grupa 433A**

* 1. **Introducere:**

Memoria internă (primara) este o parte importantă a calculatoarelor și are scopul de a menține datele și programele aferente procesului în curs de execuție. Memoria executabilă a unui calculator este compusă din memoria internă, regiștrii CPU și memoria cache. Atunci când se dorește să se prelucreze date ,ele sunt încarcate de unitatea aritmetică și logică din memoria internă în registrii CPU și când prelucrarea a luat sfarșit rezultatele sunt stocate în memoria internă. De pe alta parte memoria externă (secudară) este folosită pentru a stoca datele un timp îndelungat.

Memoria este un vector de octeți, iar fiecare dintre aceștia are o adresă proprie. în memoria principală sunt stocate codul și datele. Trebuie tinut cont că memoria este una dintre cele mai importante resurse ale unui sistem de calcul și trebuie sa fie gestionată cât mai bine. Din aceast motiv sistemul de gestiune al memoriei este una dintre cele mai importante parți ale unui sistem de operare.

Termenul de gestiune a memoriei se referă la mecanismele implementate de un sistem de operare pentru a asigura aplicațiilor resursele de memorie de care au nevoie. Aceste resurse pot fi resurse de memorie virtuală(utilizând un hard disk sau orice mediu de stocare non-RAM adițional), memorie protejată (accesul exclusiv al unui program la o regiune de memorie) și memorie partajată (mai multe procese au acces la aceeasi zonă de memorie).

*11.*

Serviciile de gestiune a memoriei în Linux sunt construite pe o baza de programare care include un dispozitiv periferic numit MMU (Memory Management Unit). MMU traduce adrese fizice de memorie în adrese liniare care sunt folosite de sistemul de operare, iar acesta solicită o intrerupere de pagină atunci când CPU (unitatea centrala de prelucrare) încearcă sa acceseze o zonă de memorie la care nu are dreptul să ajungă.

Unitatea de memorie de bază a Linux-ului este pagină care este o zonă de memorie continuă fară suprapuneri. Toată memoria fizică este organizată în pagini , iar marimea unei pagini depinde de arhitectura procesorului. În mod normal o pagină utilizată de Linux are marimea de 4096 de octeți.

Sistemul de gestiune al memoriei asigură:

* Alocarea de spațiu de memorie internă proceselor
* Protecție: fiecare proces din sistem are propriul spațiu de adrese virtual. Aceste spații de adresă sunt complet separate unul de celelalt și în timp ce un proces ruleaza o aplicație acesta nu trebuie sa influențeze celelalte procese. De asemenea putem avea zone de memorie protejate, în care nu se poate scrie.
* Minimizarea timpului de acces la locațiile de memorie
* Memorie viruala partajată: deși memoria virtuală permite proceselor sa aibă spații de adrese separate, totusi pot există și situații când procesele trebuie sa partajeze memoria. Un exemplu în care se execută acelasi cod între mai multe procese sunt bibliotecile dinamce.
  1. **Gestiunea memoriei fizice în Linux:**

Memoria fizică este divizată în unitați discrete numite pagini. Mărimea unei pagini depinde de arhitectura procesorului, dar sistemele actuale utilizează de obicei pagini de 4096 de octeți. Constanta PAGE\_SIZE (definită în <asm/page.h> ) oferă dimensiunea paginii pentru orice arhitectură.

Paginile sunt împarțite în patru segmente:

* User Code
* User Data
* Kernel Code
* Kernel Data

*12.*

În modul utilizator (**user mode**) se accesează doar User Code și User Data.

În modul Kernel se accesează de asemenea și User Data.

În Linux pot există doua posibilități:

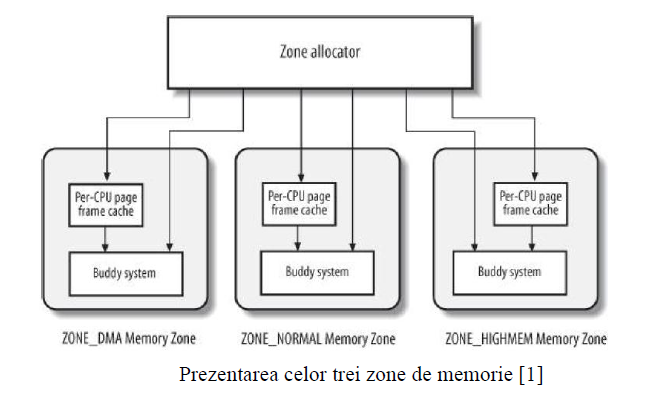
* În Kernel: poti aloca unul dintre cele mai mici obiecte kernel utilizând alocatorul slab. Se poate aloca un bloc de memorie cu kmalloc, dar acesta va aloca doar blocul apropiat cel mai mare pe care il are.
* În modul utilizator (user mode): se poate aloca orice cantitate de memorie utilizând funcțiile de management heap (heap-ul este o zonă de memorie dedicată alocării dinamice a memoriei) implemetate în bibliotecile C standard.

Fiecare pagină fizică are asociat un descriptor de pagină care conține: un contor de utilizare al paginii, flag-uri, poziția în swap sau în fișier, buferele conținute de pagină, poziția în „page cahe”.

Zonele fizice de memorie sunt [1]:

* Zonă DMA: cu pagini de memorie de 0-16 MB, este folosită pentru operatii directe cu memoria;
* Zonă Normal (LowMem): conține pagini de memorie intre 16MB și 896 MB și este folosită de kernel pentru structurile de date interne;
* Zonă HighMem: 896 MB – 4GB/64 GB, include toata memoria folosită pentru alocările de sistem;

*13.*



Prezentarea celor trei zone de memorie [1]

Memoria fizică este împărțită între mai multe noduri, iar fiecare nod are propriul procesor și are zone de memorie proprii: DMA, NORMAL, HIGHMEM

.

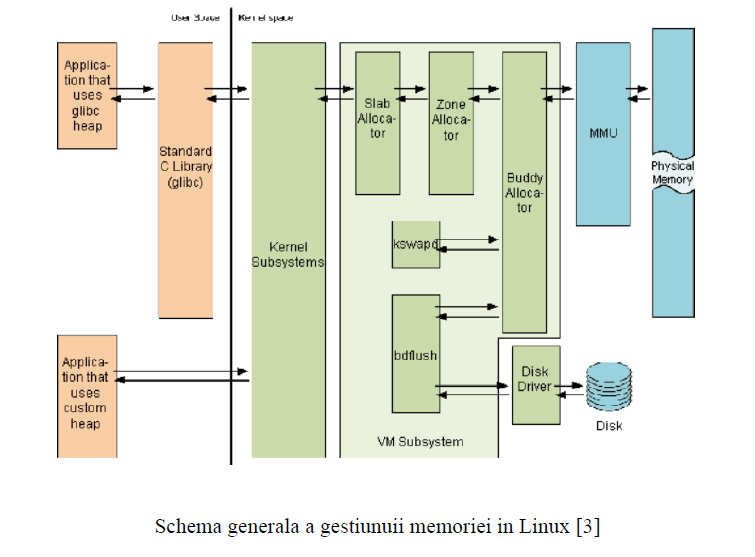
* 1. **Alocarea și dealocarea paginilor de memorie :**

Linux folosește principiul de a încărca în memoria fizică doar informația necesară și cu care se lucrează în momentul curent, pentru a lasa loc în memorie oricarui alt program care are de executat o sarcină anume. Gestiunea memorie în Linux este realizată de kernel impreună cu CPU (unitatea centrală de prelucrare). Acestea sunt responsabile de buna desfașurare a lucrurilor și trebuie sa realizeze schimbul între memoria fizică și cea de pe disc și trebuie sa depună în memoria fizică toate informațiile (date, instructiuni, operanzi) de care programele care se execută au nevoie la un moment dat.

În Linux alocarea și dealocarea de pagini de memorie se realizeaza utilizând unul din algoritmii [3]:

* Alocatorul slab
* Algoritmul (alocatorul) buddy
* Alocatorul de zonă(de sector)

*14.*



Schema generala a gestiunuii memoriei în Linux [3]

Se poate vedea în această schemă că alocarea memoriei presupune înlănțuirea celor trei alocatori enumerați mai sus și paginile nou alocate sunt adăugate la spațiul de adrese al memoriei.

**2.3.1 Alocatorul de zonă( the zone allocator):**

După cum am văzut mai sus memoria este împărțită în trei zone și pentru a aloca /dealoca (elibera) pagini de momorie în/din fiecare zonă de memorie se folosește alocatorul de zonă. O caracteristică particulară a acestui alocator este că el lucrează numai cu pagini de memorie. Memoria este alocată/ dealocată de alocatorul de zonă folosind algoritmul buddy (algoritmul „prietenului”):

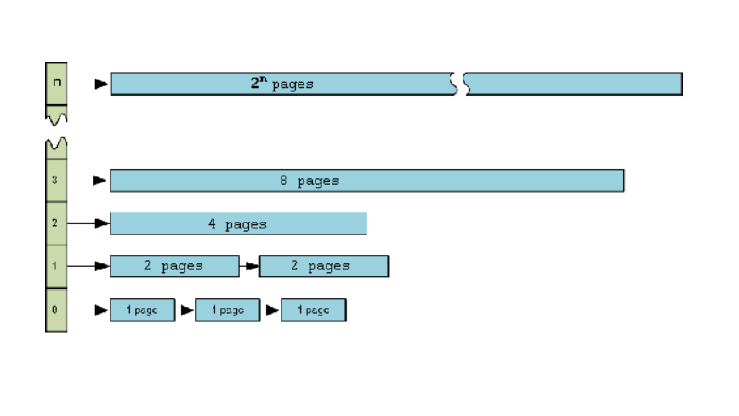
*15.*

**2.3.2 Algoritmul buddy (algoritmul „prietenului”):**

Algoritmul buddy este un algoritm de alocare a memoriei destul de simplu de implementat. Principalul scop al acestui algoritm este de a reduce fragmentarea externă și în acelasi timp de a realiza operațiile de alocare și dealocare a memorie cu o viteză cât mai mare, rezultând asftel un timp de execuție mai mic. Pentru a reduce fragmentarea externă paginile libere de memorie sunt grupate în blocuri de dimensiune variabilă, dar cu condiția ca blocurile sa aibă o dimensiune putere a lui 2. Cu aceste blocuri de pagini se alcătuiesc liste, iar o lista trebuie sa aibă numai blocuri de aceeași dimensiune ( de exemplu o listă de blocuri de câte 2 pagini sau o listă de blocuri de câte 4 pagini și așa mai departe). De exemplu dacă are loc o cerere de 4 pagini conținute se verifică dacă există un bloc liber de 4 pagini și în cazul în care este găsit blocul se alocă imediat. Sau dacă un bloc de 8 pagini este disponibil se împarte acest bloc în două blocuri de câte 4 pagini și un bloc se aloca unitații apelante, iar celalalt este plasat în lista blocurilor de 4 pagini.

În cazul dealocarii dacă există blocuri adiacente în memoria fizică, fiecare bloc având dimensiunea N, iar primul bloc este aliniat la 2N atunci cele 2 blocuri se unesc într-un bloc de dimensiune 2N. Se incearcă să se unească cât mai multe blocuri pentru a reduce fragmentarea externă.

În figura următoare [3] sunt prezentate listele cu blocuri de pagini utilizate în algoritmul buddy:



*16.*

**2.3.3 Alocatorul slab:**

Acest alocator are zone de memorie diferite pentru obiecte diferite, zone pe care le putem asemăna cu niște plăci de mozaic și de aici ii vine și numele de „slab” care în limba engleză înseamna „lespede”.

Alocatorul slab încearcă sa rezolve următoarele probleme:

* Încearcă să nu interfereze prea mult cu alte zone de memorie atunci când caută o zonă nouă de memorie, se spune că are o „urmă” mica (small footprint);
* Încearcă să nu pună două obiecte diferite în acceași linie din cache-ul de date.
* Incearcă să reducă numarul de operații de inițializare asupra noilor obiecte alocate.

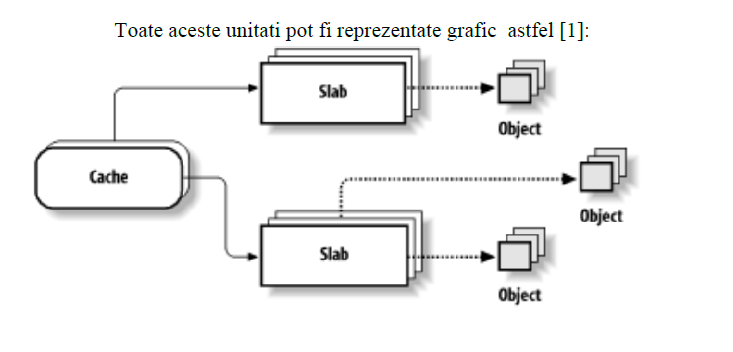
Există multe situații când trebuie sa alocăm obiecte care au o dimensiune mai mică decât cea a unei pagini și trebuie să găsim soluții să alocăm părți mai mici de memorie. De aceea a fost introdus alocatorul slab care încearca să rezolve aceste situații. [3]

În cazul alocatorului slab zonele de memorie sunt văzute ca niște obiecte și fiecare obiect are un constructor și un destructor propriu. Obiectele dealocate sunt păstrate într-un cache astfel ca la urmatoarea folosire a lor nu mai trebuie reinițializate și nu mai este nevoie să se apeleze algoritmul buddy. [3]

Alocatorul slab este format din :

* Cache: în cache se stochează obiectele care au fost recent utilizate, obiecte care trebuie sa fie de acelasi tip. Cache-ul reprezintă unitatea cea mai mare de stocare în cadrul alocatorului slab.
* Slab: este un container pentru obiecte și este alcătuit din una sau mai multe pagini. Mai multe slab-uri la un loc se stochează într-un cache.
* Obiect: obiectul este unitatea cea mai mică care este utilizată în cazul alocatorului slab. Mai multe obiecte puse la un loc alcatuiesc un slab. [2]

*17.*



**3.Subsistemul de gestiune a memoriei fizice la Windows**

**–Costea Cosmin - grupa 433A**

## 3.1 Introducere

Creșterea puterii de calcul și a spațiului adreselor la microprocesoarele moderne a condus la extinderea gamei de aplicații, incluzând câmpuri de tipul sistemelor multiutilizator cu funcționare în timp real, pe timpuri rezervate doar calculatoarelor mari sau minicalculatoarelor.

*18.*

Ideile ce stau la baza implementarii mecanismelor de gestiune a memoriei nu sunt noi, ci sunt împrumutate din soluțiile folosite în calculatoare mari sau minicalculatoare. Totuși, aceste idei sunt noi în lumea microprocesoarelor, fapt ce a determinat descrierea celor mai frecvent utilizate metode pentru utilizarea eficientă a memoriei, cât și pentru protecția împotriva accesului neautorizat, voit sau eronat.

Subsistemul de gestiune al memoriei din cadrul unui sistem de operare este folosit de toate celelalte subsisteme: planificator, I/O, sistemul de fișiere, gestiunea proceselor, networking. Memoria este o resursă importantă, de aceea sunt necesari algoritmi eficienți de utilizare și gestiune a acesteia.

Alocarea memoriei nu este statică, întrucât, în cele din urmă, sistemul va prelucra toate informațiile memorate. Deci, datele și programele vor fi mutate dintr-o zonă de memorie în alta. Fluxul acestor informații poate fi controlat de către programator, sistemul de operare, hardware sau o combinație a tuturor acestor factori. Fiecare din ei poate gestiona un anumit subsistem al memoriei.

## 3.2 Utilizarea memoriei

Utilizarea rațională a memoriei se realizează prin luarea în considerare a următoarelor :

- există mai multe tipuri de memorie, cu diverse viteze și costuri pe bit (cum sunt cele cu semiconductoare și cele magnetice). În general, capacitatea memoriilor rapide este limitată de cost sau posibilități tehnologice, în timp ce memoriile ieftine au timpuri lungi de acces;

*19.*

- într-un sistem complex, așa cum este calculatorul multiprogramat funcționând prin divizarea timpului, nu este necesar a păstra simultan toate programele în memoria primară, deoarece numai o submulțime a acestora este gata de rulare, în timp ce altele pot aștepta terminarea unor operații I/O sau pot fi temporar blocate.

## 3.3 Rolul subsistemului de gestiune

Rolul subsistemului de gestiune a memoriei este de :

* a ține evidența zonelor de memorie fizică (ocupate sau libere)
* a oferi proceselor sau celorlalte subsisteme acces la memorie
* a mapa paginile de memorie virtuală ale unui proces (pages) peste paginile fizice (frames).

În general este convenabil a considera sistemul de memorie drept constând din mai multe subsisteme, cu capacități diferite și timpi de acces diferiți. Într-o astfel de configurare mecanismele de gestiune a memoriei vor distribui informația între diversele tipuri de memorie, încercând a păstra datele și programele la care accesul este mai frecvent în memoriile mai rapide, restul informațiilor fiind plasate în memoriile mai lente.

Nucleul sistemului de operare oferă un set de interfețe (apeluri de sistem) care permit alocarea/dealocarea de memorie, maparea unor regiuni de memorie virtuală peste fișiere, partajarea zonelor de memorie.

Din păcate, nivelul limitat de înțelegere a acestor interfețe și a acțiunilor ce se petrec în spate conduc la o serie de probleme foarte des întâlnite în aplicațiile software: memory leak-uri, accese nevalide, suprascrieri, buffer overflow, corupere de zone de memorie.

Este, în consecință, fundamentală cunoașterea contextului în care acționează subsistemul de gestiune a memoriei și înțelegerea interfeței pusă la dispoziție de sistemul de operare programatorului.

*20.*

## 3.4 Localitatea programelor

Un exemplu de alocare a informației sub comanda explicită a programului este constituit de gestiunea registrelor efectuată de către un programator în limbajul de așamblare. Registrele CPU reprezintă cea mai rapidă formă de memorie din sistem, dar capacitatea lor nu este suficient de mare pentru a memora toate datele manipulate de catre un program. Deci, un programator experimentat foloseste registrele pentru a memora datele la care frecventa de acces este cea mai mare. Mai mult, intrucât mulțimea acestor variabile se poate schimba pe durata execuției unui program, programatorul va modifica modul de utilizare a registrelor astfel încât variabilele care nu mai sunt intens utilizate de către secțiunea urmatoare de cod să fie mutate înapoi în memoria primară, de unde se aduc în registre cele frecvent apelate.

În cadrul acestui exemplu, gestiunea unei zone de memorie unică și rapidă se bazează pe cunoașterea structurii programului. Deci, o astfel de tehnică nu este adecvata pentru gestionarea memoriei prin mecanisme hardware sau de către sistemul de operare, pentru că lor le lipsesc cunostintele despre program. Un numar de experimente și masurari efectuate asupra sistemelor a evidentiat existenta unui principiu, numit principiul localitatii programelor (program locality). El se poate enunța astfel:

Dacă la momentul t se face acces la adresa de memorie x, există o probabilitate ridicată ca, la momentul t+dt,să se efectueze accesul la adresa de memorie x+dx, unde dt și dx sunt valori mici ale lui t și x.

Mecanismele de gestiune automată a spațiului de memorie pot folosi localitatea programelor pentru a păstra în memoria rapidă subspațiul de memorie cu cele mai frecvente accese în cadrul programului curent. Un element esențial constă în detectarea tranziției între faze și identificarea locațiilor de memorie utilizate de către fiecare fază. În cele ce urmează se vor ilustra soluțiile cele mai des folosite ale acestor probleme.

*21.*

În final, combinarea gestionării automate a memoriei cu cea efectuata de catre utilizator presupune oferirea de catre programator a unor informații privind structura programului, ceea ce permite mecanismului automat sa mute informațiile între diferitele subsisteme de memorie corespunzător informațiilor furnizate de către programator.

## 3.5 Spațiul de adresă al unui proces

Spațiul de adrese al unui proces, sau, mai bine spus, spațiul virtual de adresă al unui proces reprezintă zonăde memorie virtuală utilizabilă de un proces. Fiecare proces are un spațiu de adresă propriu. Chiar în situațiile în care două procese partajează o zonă de memorie, spațiul virtual este distinct, dar se mapează peste aceeasi zonă de memorie fizică.

Cele 4 zone importante din spațiul de adresă al unui proces sunt zonăde date, zonăde cod, stiva și heap-ul. După cum se observă și din figură, stiva și heap-ul sunt zonele care pot crește. De fapt, aceste două zone sunt dinamice și au sens doar în contextul unui proces. De partea cealaltă, informațiile din zonăde date și din zonăde cod sunt descrise în executabil.

**Zona de cod**

Segmentul de cod (denumit și text segment) reprezintă instrucțiunile în limbaj masină ale programului. Registrul de tip instruction pointer (IP) va referi adrese din zonăde cod.

**Zone de date**

Zonele de date conțin variabilele globale definite într-un program și variabilele de tipul read-only. În funcție de tipul de date există mai multe subtipuri de zone de date.

*22.*

**Stiva**

Stiva este o regiune dinamică în cadrul unui proces, fiind gestionată automat de compilator.

Stiva este folosită pentru a stoca “stack frame-uri”. Pentru fiecare apel de funcție se va crea un nou “stack frame”.

Un “stack frame” conține:

-variabile locale

-argumentele funcției

-adresa de retur

Pe marea majoritate a arhitecturilor moderne stiva crește în jos și heap-ul crește în sus. Stiva crește la fiecare apel de funcție și scade la fiecare revenire din funcție.

**Heap-ul**

Heap-ul este zonăde memorie dedicată alocării dinamice a memoriei. Heap-ul este folosit pentru alocarea de regiuni de memorie a căror dimensiune este determinată la runtime.

La fel ca și stiva, heap-ul este o regiune dinamică care îsi modifică dimensiunea. Spre deosebire de stivă, însă, heap-ul nu este gestionat de compilator. Este de datoria programatorului să știe câtă memorie trebuie să aloce și să rețină cât a alocat și când trebuie să dealoce. Problemele frecvente în majoritatea programelor țin de pierderea referințelor la zonele alocate (memory leaks) sau referirea de zone nealocate sau insuficient alocate (accese nevalide).

În limbaje precum Java, Lisp etc. unde nu există “pointer freedom”, eliberarea spațiului alocat se face automat prin intermediul unui garbage collector. Pe aceste sisteme se previne problema pierderii referințelor, dar încă rămâne activă problema referirii zonelor nealocate.

*23.*

## 

## 3.6 Alocarea/Dealocarea memoriei

**Alocarea memoriei** este realizată static de compilator sau dinamic, în timpul execuției. Alocarea statică e realizată în segmentele de date pentru variabilele globale sau pentru literali.

În timpul execuției, variabilele se alocă pe stivă sau în heap. Alocarea pe stivă se realizează automat de compilator pentru variabilele locale unei funcții (mai puțin variabilele locale prefixate de identificatorul **static**).

Alocarea **dinamică** se realizează în heap. Alocarea dinamică are loc atunci când nu se știe, în momentul compilării, câtă memorie va fi necesară pentru o variabilă, o structură, un vector. Dacă se știe din momentul compilării cât spațiu va ocupa o variabilă, se recomandă alocarea ei statică, pentru a preveni erorile frecvent apărute în contextul alocării dinamice.

Pentru a fragmenta cât mai puțin spațiul de adrese al procesului, ca urmare a alocărilor și dealocărilor unor zone de dimensiuni variate, alocatorul de memorie va organiza segmentul de date alocate dinamic sub formă de heap, de unde și numele segmentului.

**Dealocarea memoriei** înseamnă eliberarea zonei de memorie (este marcată ca fiind liberă) alocate dinamic anterior.

Dacă se omite dealocarea unei zone de memorie, aceasta va rămâne alocată pe întreaga durata de rulare a procesului. Ori de câte ori nu mai este nevoie de o zonă de memorie, aceasta trebuie dealocată pentru eficiența utilizării spațiului de memorie.

*24.*

Pot apărea probleme și dacă se încearcă dealocarea aceleiasi regiuni de memorie de două sau mai multe ori și se corup datele interne de management al zonelor alocate dintr-un heap.

## 3.7 Metode clasice de alocare a memoriei

La sistemele monoutilizator este disponibil întreg spaţiul de memorie. Gestiunea acestui spațiu cadeexclusiv în sarcina utilizatorului. El are la dispoziţie tehnici de suprapunere (overlay) pentru a-si putea rula programele mari. Ideea de bază este de a păstra în memoria internă numai aceleinstrucţiuni și date de care este nevoie permanent. Celelalte grupuri de instrucţiuni sunt încărcate înmemoria internă numai atunci când este nevoie de ele, după care sunt evacuate. În figura 3.1 esteilustrat acest mod de lucru.

Porţiunea dintre adresele 0 și a-1 este rezervată nucleului SO, carerămâne acolo de la încărcare și până la oprirea sistemului. Între adresele c și m-1 (dacă memoria arecapacitatea de m locaţii) este spațiul nefolosit de către programul utilizator activ. Evident, adresa cvariază de la un program utilizator la altul.

**Alocarea cu partiţii fixe**

(MFT-Memory Fix Tasks sau alocare statică). Se presupune că memoria este împărţită în N zone de lungime fixă numite partiţii. Presupunem că o partiţie i este de lungime Ni și este alocată unui proces pe toată durata execuției lui, indiferent dacă o ocupă complet sau nu. Editorul de legături pregăteşte programele pentru a fi rulate într-o zonă de memorie prestabilită.

De obicei, partiţiile au lungimi diferite, astefel încât procesele solicită partiţii în conformitate cudimensiunile lor. Dacă un proces are nevoie de **n**k unităţi de memorie ele poate fi încărcat în oricare dintre partiţiile i, pentru care Ni ≥ **n**k. În timpul execuţiei procesului, un spațiu de dimensiune Ni – **n**k rămâne neutilizat.

*25.*

Acest fenomen se numeşte **fragmentare internă**. Problema care se pune este să se aleagă partiţia astfel încât porţiunea de memorie nefolosită să aibă o dimensiune căt maimică, adică să se minimizeze diferenţele de forma Ni - **n**k.

Dacă un proces nu încape în nici una dintre partiţiile existente, el nu poate fi executat. Una dintre problemele cele mai dificile este fixarea acestor dimensiuni. Alegerea unor dimensiuni mai mari scade probabilitatea ca unele procese să nu poată fi executate, dar scade și numărul proceselor active din sistem. În cazul în care există job-uri în sistem care aşteaptă să fie executate, dar toate partiţiile libere existente la momentul respectiv sunt prea mici, apare fenomenul de **fragmentare externă a memoriei**.

Alocarea cu partiţii variabile

(alocare dinamică sau alocare MVT – Memory Variable Task),reprezintă o extensie a alocării cu partiţii fixe, care permite o exploatare mai eficientă a memorieiSC. În cazul multiprogramării cu partiţii fixe, problema cea mai dificilă este optimizarea dimensiuniipartiţiilor, astfel încît să se minimizeze fragmentarea memoriei. De asemenea, se presupune că joburile au o dimensiune cunoscută, ipoteză care nu este în general adevărată.Aceste inconveniente pot fi rezolvate dacă se admite modificarea dinamică a dimensiunii partiţiilor, în funcţie de solicitările adresate sistemului și de capacitatea de memorie încă disponibilă la un moment dat. Prin folosirea acestei metode, numărul și dimensiunea partiţiilor se modifică în timp.

*26.*

## 3.8 Memoria primară

Memoria primară constituie cel mai mare subsistem de memorie adresabil direct de către CPU. Toate programele ce se ruleaza (cod și date) trebuie memorate în memoria primară, ca și informațiile necesare gestionării memoriei. Totusi, nu toate programele sunt gata de a rula în paralel si, chiar când un program este executat, el folosește doar un subset al spațiului sau virtual de adresare pe durata unei anumite faze.

Tehnicile de gestiune a conținutului memoriei primare necesită un anumit suport hardware, dar algoritmii sunt implementati prin software. De aici decurge o varietate de soluții posibile.

Trebuie remarcat ca memoria primară și registrele sunt două niveluri ierarhice prezente întotdeauna într-un sistem, în timp ce celelalte niveluri s-ar putea sa lipsească.

## 3.9 Memoria de masă

Informația care nu este imediat necesară CPU (programe care nu sunt gata de rulare sau subspații de adresare care nu sunt necesare în faza curentă) este memorată în dispozitive cu mediu magnetic de stocare, care ofera un mare volum de memorare la costuri unitare mici, dar cu timpi de acces cu cel puțin trei ordine de mărime mai mari decât cei ai memoriei primare.

*27.*

## Consideratii privind costul și performanța

Succesul organizării ierarhizate a memoriei este datorat bunelor performanțe și costuri implicate, deoarece costul pe bit este apropiat de cel al celei mai lente memorii din sistem, în timp ce performanțele obținute sunt comparabile cu cele ale celor mai rapide memorii din ierarhie.

În tabelul de mai jos sunt indicate caracteristicile memoriilor folosite la diferitele niveluri ale ierarhiei; ele permit evaluarea costurilor și performanțelor unui sistem de memorie.

|  |  |  |
| --- | --- | --- |
| NUME | DIMENSIUNE | TIMP ACCES |
|  |  |  |
| Registre | 16 -32 | 0.1 ns |
| Cache pe circuit | 16 kB - 256 kB | 1 - 2 ns |
| Cache pe plachetă | 265 kB - 1 MB | 8 - 10 ns |
| Memorie principală | 128 MB - 1 GB | 50 - 70 ns |
| Memorie secundară | 8 GB - 128 GB | 10 ms |

*28.*

## 3.11 Lucru cu memoria – Probleme

Lucrul cu heap-ul este una dintre cauzele principale ale aparițiilor problemelor de programare. Lucrul cu pointerii, necesitatea folosirii unor apeluri de sistem/bibliotecă pentru alocare/dealocare, pot conduce la o serie de probleme care afectează (de multe ori fatal) funcționarea unui program.

Problemele cele mai des întâlnite în lucrul cu memoria sunt:

- accesul nevalid la memorie - ce prespune accesarea unor zone care nu au fost alocate sau au fost eliberate.

- leak-urile de memorie - situațiile în care se pierde referința la o zonă alocată anterior. Acea zonă va rămâne ocupată până la încheierea procesului.

**Acces nevalid**

De obicei, accesarea unei zone de memorie nevalide rezultă într-o eroare de pagină (page fault) și terminarea procesului. Totusi, dacă eroarea apare la o adresă nevalidă, dar într-o pagină validă, hardware-ul și sistemul de operare nu vor putea sesiza acțiunea ca fiind nevalidă. Acest lucru este din cauza faptului că alocarea memoriei se face la nivel de pagină. Spre exemplu, pot există situații în care să fie folosită doar jumătate din pagină. Desi cealaltă jumătate conține adrese nevalide, sistemul de operare nu va putea detecta accesele nevalide la acea zonă.

Asemenea accese pot duce la coruperea heap-ului și la pierderea consistenței memoriei alocate.

*29.*

Un tip special de acces nevalid este **buffer overflow**. Acest tip de atac presupune referirea unor regiuni valide din spațiul de adresă al unui proces prin intermediul unei variabile care nu ar trebui să poată referenția aceste adrese. De obicei, un atac de tip buffer overflow rezultă în rularea de cod nesigur. Protejarea împotriva atacurilor de tip buffer overflow se realizează prin verificarea limitelor unui buffer/vector fie la compilare, fie la rulare.

**GDB - Detectarea zonei de acces nevalid de tip page fault**

O comandă foarte utilă atunci când se depanează programe complexe este backtrace. Această comandă afișează toate apelurile de funcții în curs de execuție.

Fiecare funcție are alocată pe stivă un frame, în care sunt plasate variabilele locale funcției, parametrii funcției și adresa de revenire din funcție. În momentul în care o funcție este apelată, se creează un nou frame prin alocarea de spațiu pe stivă de către funcția apelată. Astfel, dacă avem apeluri de funcții imbricate, atunci stiva va conține toate frame-urile tuturor funcțiilor apelate imbricat.

GDB dă posibilitatea utilizatorului să examineze frame-urile prezente în stivă. Astfel, utilizatorul poate alege oricare din frame-urile prezente folosind comanda frame.

**mcheck - verificarea consistenței heap-ului**

**glibc** permite verificarea consistenței heap-ului prin intermediul apelului mcheck definit în mcheck.h. Apelul mcheck forțează malloc să execute diverse verificări de consistență precum scrierea peste un bloc alocat cu malloc. Alternativ, se poate folosi opțiunea -lmcheck la legarea programului fără a afecta sursa acestuia.

*30.*

Varianta cea mai simplă este folosirea variabilei de mediu MALLOC\_CHECK\_. Dacă un program va fi lansat în execuție cu variabila MALLOC\_CHECK\_ configurată, atunci vor fi afișate mesaje de eroare (eventual programul va fi terminat forțat - aborted).

**Leak-uri de memorie**

Un leak de memorie apare în două situații:

-un program omite să elibereze o zonă de memorie

-un program pierde referința la o zonă de memorie alocată si, drept consecință, nu o poate elibera

Memory leak-urile au ca efect reducerea cantității de memorie existentă în sistem. Se poate ajunge, în situații extreme, la consumarea întregii memorii a sistemului și la imposibilitatea de funcționare a diverselor aplicații ale acestuia.

Ca și în cazul problemei accesului nevalid la memorie, utilitarul Valgrind este foarte util în detectarea leak-urilor de memorie ale unui program.

**Valgrind**

Valgrind reprezintă o suită de utilitare folosite pentru operații de debugging și profiling. Cel mai popular este Memcheck, un utilitar care permite detectarea de erori de lucru cu memoria (accese nevalide, memory leak-uri etc.). Alte utilitare din suita Valgrind sunt Cachegrind, Callgrind utile pentru profiling sau Helgrind, util pentru depanarea programelor multithreaded.

**Dublă dealocare**

Denumirea de “dublă dealocare” oferă o bună intuiție asupra cauzei: eliberarea de două ori a aceluiasi spațiu de memorie. Dubla dealocare poate avea efecte negative deoarece afectează structurile interne folosite pentru a gestiona memoria ocupată.

În ultimele versiuni ale bibliotecii standard C, se detectează automat cazurile de dublă dealocare.

*31.*

## 

1. **Bibliografie:**

-http://en.wikipedia.org/wiki/Paging

- https://www.cs.duke.edu/csl/docs/sysadmin\_course/sysadm-92.html

- http://www.computerhope.com/jargon/s/swapfile.htm

-http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-04

-http://www.scribd.com/doc/13597643/Sisteme-de-Operare-

-http://www.scritube.com/stiinta/informatica/GESTIUNEA-MEMORIEI32136.php

-Arpaci-Dusseau, Remzi H.; Arpaci-Dusseau, Andrea C. (2014), *Operating Systems: Three Easy Pieces [Chapter: Paging]*

-Belzer, Jack; Holzman, Albert G.; Kent, Allen, eds. (1981). "Virtual memory systems".*Encyclopedia of computer science and technology*

-Figura 1. Sistemul Ferranti Atlas primul supercalculator din regatul unit. Imaginea a fost descarcate de la adresa -[www.ourcomputerheritage.org](http://www.ourcomputerheritage.org/F5%20part%20of%20the%20Ferranti%20Atlas%20at%20Manchester%20University.htm)

-Figura 2 Gasirea PID-ului browserului Firefox *-* [www.cyberciti.biz](http://www.cyberciti.biz/faq/kill-process-in-linux-or-terminate-a-process-in-unix-or-linux-systems/)

-Figura 3 Crearea unui process nou în Windows -[faculty.salina.k-state.edu](http://faculty.salina.k-state.edu/tim/ossg/Process/p_create.html)

-Figura 4. Fișier cu stream-ul creat default și inca alte doua streamuri create additional - http://msdn.microsoft.com/enus/library/windows/

*32.*

[1] “Understanding the Linux Kernel, 3rd Edition” de Daniel P. Bovet, Marco Cesati, editura O’Reilly, noiembrie 2005

[2] “Memory Management în Linux - Desktop Companion to the Linux Source Code”, de Abhishek Nayani, Mel Gorman & Rodrigo S. de Castro, 25 mai 2002

[3] http://www.codeproject.com/Articles/131862/Special-Features-of-Linux-Memory-Management-Mechan

[4] “Outline of the Linux Memory Management System” de Joe Knapka

*33.*