Gestiunea

memoriei

Cadîr Eren

Drossu Florin Stefan

Nichifor Ruxandra Lorena

431A

**Cuprins**

1. Alocarea memoriei*Nichifor Ruxandra Lorena*

1.1 Introducere

1.2 Tipuri de alocari

1.3 Alocarea dinamica a memoriei

1.3.1 Pointeri

1.3.2 Memoria Heap

1.4 Concluzii

2. Alocarea memoriei în Windows*Cadîr Eren*

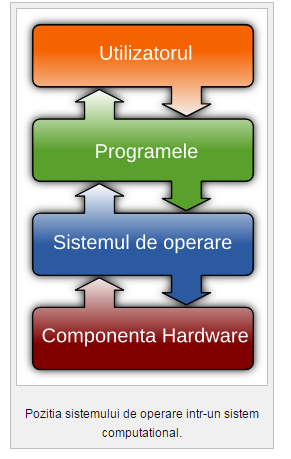
3. Probleme în lucrul cu memoria*Cadîr Eren*

4. Alocarea memoriei în Linux*Drossu Florin Ștefan*

5. Detecția și corecția erorilor în memorie*Cadîr Eren*

* 1. **Introducere**

Un sistem de operare este un ansamblu de programe care controleaza distributia resurselor unui calculator ( coordoneaza si controleaza executia programelor) si permite comunicarea utilizatorului cu sistemul de calcul.



Sistemul de operare este organizat pe două niveluri:

1. nivelul fizic include componeneta firmware a sistemului de calcul (programe care se executa pornirea sistemului de calcul si rutinele care mediaza utilizarea componentelor fizice ale sistemului de calcul-drivere fizice)
2. nivelul logic exploateaza sistemul de calcul cu ajutorul programelor sistemului de operare.

Functiile sistemului de operare sunt:

1. functia de instalare  automata a unui nou sistem de operare pe un sistem de calcul;

*(sursa:* [*http://cmc.site11.com/ase/Informatica/bti/tema\_finala/SO.html*](http://cmc.site11.com/ase/Informatica/bti/tema_finala/SO.html)*)*

1. functia de incarcare in memoria interna a sistemului de operare, la pornirea sistemului de calcul;
2. functia de configurare dinamica a sistemului de operare;
3. efectuarea operatiilor de intrare/iesire la nivel fizic;
4. interfata cu utilizatorul;
5. controlul executiei programelor;
6. gestionarea alocarii resurselor sistemului de calcul;
7. asigurarea protectiei intre utilizatori;
8. tratarea anumitor tipuri de erori-fizice(exemplu: erori de citire/scriere in memoria externa) sau logice(exemplu: operatiuni interzise)
9. functii auxiliare (exemplu: tinerea evidentelor erorilor).

*Gestiunea memoriei* sau gestionarea alocarii resurselor sistemului de calcul este una din funcţiile principale ale unui sistem de operare.

Subsistemul de gestiune a memoriei din cadrul unui sistem de operare este folosit de toate celelalte subsisteme, precum: scheduling, I/O, filesystem, gestiunea proceselor, networking.

Rolul subsistemului de *gestiune a memoriei* este de :

● de a aloca sau de a dealoca memoria;

● partajarea memoriei, oferind proceselor sau celorlalte subsisteme acces la memorie;

● a mapa paginile de memorie virtuală ale unui proces (pages) peste paginile fizice (frames).

*Alocarea si dealocarea*

Variabilele ocupa un spatiu in memoria calculatorului. Fiecare variabila poate avea un nume propriu, in concordanta cu scopul in care va fi folosita in codul sursa. Acestea nu vor avea un spatiu permanent pe care il vor folosi, mai ales datorita limbajelor moderne care le confera exact atat spatiu cat au nevoie, doar atunci cand au nevoie. Se numeste alocarea atunci cand unei variabile ii este dat un spatiu din memorie pentru a retine valoarea. Cand o variabila este alocata, poate opera ca o variabila obisnuita si sa retina o anumita valoare. O variabila este dealocata cand sistemul solicita inapoierea memoriei folosite de aceasta si nu ii mai ofera un alt spatiu pentru a-I retine valoarea.

Timpul de la alocare la dealocare se numeste timp de viata.

**1.2Tipuri de alocari**

1.Limbaje care nu pot aloca dinamic memorie. Din această categorie fac parte primele limbaje: Cobol, Fortran.

2.Limbaje cu alocare şi dealocare explicită. Limbaje ca Pascal, C şi C++ îi permit utilizatorului să ceară pe parcursul execuţiei noi zone de memorie şi să returneze memoria folosită.

3. Limbaje cu colectoare de gunoaie (garbage collection). Lisp şi Java folosesc un mecanism extrem de interesant

**1.3 Alocarea dinamica a memoriei**

Fiecarui program i se aloca trei zone distincte in memoria interna:

* segment de date;
* segment de stiva;
* heap.

Prin alocare dinamicaamemoriei vom intelege alocarea unor variabile in Heap,alocare care se face in timpul executarii programului si nu de la inceput.

Alocarea dinamica foloseste pointeri si variabile de tip pointer.

*1.3.1 Pointeri*

Un pointer detine o referinta a unei valori, practic este adresa unei variabile.

Variabila de tip pointer este o variabila care poate retine adresele altor variabile.

Deferentul urmareste referinta unui pointer pentru a gasi valoarea pointata (referita). Unui pointer trebuie sa ii fie atribuita o adresa inainte ca operatia de deferentiere sa aiba loc.

Pointerul nul este un pointer special care face referire la faptul ca pointerul nu are o valoare pointata.

Poinerii rezolva 2 probleme fundamentale:

* asigura unor sectiuni diferite de cod posibilitatea sa-si imparta informatia usor ( o alta solutie ar fi fost sa se copieze informatia) ;
* premit apelarea unor date complexe apartinand unor structuri sau arbori.

*Implementarea pointerilor*

Fiecare parte a unei memorii are o adresa unica numerica. Un pointer la o adresa a memoriei este un integer care stocheaza adresa acelei celule a memoriei.

Deferentul cauta adresa referita pentru a prelua informatia stocata la acea locatie.

Atribuirea doar copiaza adresa numerica de la un pointer la altul.

*1.3.2. Memoria Heap*

Memoria Heap ( Memoria dinamica) este o alternativa a stivei aflata in memoria locala.

Memoria locala este alocata( la apelarea functiei “call”) si dealocata automat.Memoria Heap este alocata si dealocata explicit de catre programator.

*Alocarea memoriei Heap*

Memoria Heap este o memorie care poate fi folosita de program, prin cererea unui anumit bloc, de o anumita dimensiune, pe care ulterior o va folosi in interiorul memoriei heap. Pentru a aloca un bloc de o anumita dimensiune, programul face o cerere explicita cu ajutorul functiei “allocation”. Aceasta functie rezerva un bloc de dimensiunea specificata in Heap si returneaza un pointer la aceasta.

Fiecare cerere de alocare rezerva o zona continua de dimensiunea ceruta si returneaza un pointer catre noul bloc in program. Deoarece fiecare bloc este intotdeauna referit de un pointer, blocul va juca rolul de pointat/referit. Programul va dirija blocurile din memoria Heap numai prin pointeri. Pointerul unui bloc este pointerul adresei celei mai mici al blocului.

Managerul memoriei Heap poate aloca in Heap blocuri, atat timp cat nu se suprapun si cat respecta dimensiunea ceruta. Pot exista atat zone libere, cat si zone alocate. Managerul memoriei Heap are propria lui structura de date pentru a inregistra zonele folosite la un moment dat. Managerul aloca spatiul in functie de cereri si le inregistreaza pentru a determina zonele folosite.

*Dealocarea*

Cand programul nu mai are nevoie de un anumit bloc de memorie, face o cerere de dealocare specifica pentru a indica pentru a indica managerului ca programul nu mai foloseste acea zona de memorie. Managerul isi reface structura privata cu date, astfel incat zona aceea de memorie va fi considerata libera si va putea fi refolosita.Uneori codul va seta pointerul la zero, pentru a specifica lipsa validitatii.

*Proprietatile memoriei Heap:*

* Memoria Heap este o zona de memorie in care se pot aloca blocuri de memorie pentr un anumit program;
* Managementul memoriei Heap este inclus intr-o librarie, de exemplu: malloc(), free(), and realloc() in C;
* Managerul Heap foloseste propriile structuri de date pentru a tine o evidenta a blocurilor;
* Memoria Heap poate avea o dimensiune fixa sau poate parea ca e de dimeniune fixa, dar foarte mare sustinuta de memoria virtuala. Daca se depaseste aceasta valoare, se va returna un pointer nul sau se va enunta o exceptie;
* Functia de alocare cere un anumit bloc, de o anumita dimensiune. Managerul selecteaza aria din memorie care satisface cererea si o marcheaza ca fiind in folosinta si returneaza un pointer in blocul Heap. Apelantul poate folosi acum memorie, prin deferentul pointerului. Blocul este rezervat apelantului. Blocul nu isi va schimba locatia sau dimensiunea pe toata durata alocarii lui de catre manager. Continutul blocului este necunoscut, asa ca programatorul i va seta valoarea;
* Dealocarea este ulterioara alocarii. Programul face o singura cerere de dealocare pentru a intoarce un bloc din memoria heap si de a o elibera pentru o utilizare ulteriora. Fiecare bloc ar trebui dealocat doar o data. Functia de dealocare are ca argument un pointer catre un bloc creat anterior. Pointerul trebuie sa fie exact acelasi pointer returnat anterior de catre functia de alocare. Dupa dealocare programul trebuie sa nu mai faca referinta la acel pointer.

*Alocarea dinamica in diferite limbaje de nivel inalt*

Programatorul în C are la dispoziţie funcţiile malloc (calloc/realloc ) pentru a aloca memorie, şi funcţia *free* pentru a o elibera. Prototipurile acestor funcţii se află în fişierul header <stdlib.h> din biblioteca standard C, prezentă pe orice sistem.

Funcţia malloc are un singur argument: numărul de octeţi care trebuie alocaţi. Rezultatul funcţiei este un pointer generic (void\*) spre zona alocată. Dacă malloc nu găseşte suficienta memorie, atunci rezultatul apelului ei este pointerul cu valoarea 0 (NULL).

Funcţia free() are tot un singur argument, şi nici un rezultat. Argumentul ei este un pointer obţinut de la un malloc anterior; efectul executării ei este eliberarea zonei de memorie aflată în acel loc.

Biblioteca în care sunt implementate malloc şi free detine o harta a zonelor libere si a celor ocupate.

Pentru sistemul de operare Unix, functiile bibliotecii care detin harta se apeleaza cu linia de cod “sbrk”, cu un singur parametru pozitiv sau negativ, urmand alocarea unui spatiu egal cu parametrul la sfarsitul segmentului de date al programului rulat.

**1.4Concluziile alocarii dinamice.**

*Avantajele alocarii dinamice:*

-utilizarea memoriei doar pe perioada utilizarii respectivei structuri, oferindu-i un timp de viata determinat;

-in C++ memoria din segmentul de date nu este intotdeauna suficienta, dar apeland la Heap, creste memoria disponibil, putand utiliza exact dimensiunea data de programator.

*Dezavantajele alocarii dinamice sunt:*

-alocarea memoriei Heap presupune ca programatorul sa creeze un cod foarte explicit;

-aparitia unor erori in cod, care conduc la zone deja dealocate.

*Pierderile de memorie*

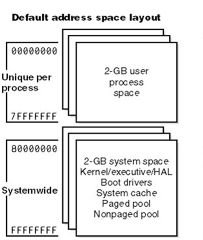
Daca un program care ruleaza pentru o perioada nedeterminata la terminarea sa nu returneaza spatiul memoriei Heap, aceasta in timp se va umple si nu va onora viitoarele cereri de alocare, astfel programul va ingheta.

De cele mai multe ori detectia erorilor si corectia codului pentru blocajul memoriei Heap nu e testata corect, deoarece verificarea programului se face pentru perioade scurte de timp si nu se ajunge la o utilizare continua cu valori ale unei utilizari reale. Majoritatea compilatoarelor au un debugger pentru memoria heap care te poate ajuta sa gasesti eventualele dealocari lipsa.

**2.Alocarea memoriei în Windows**

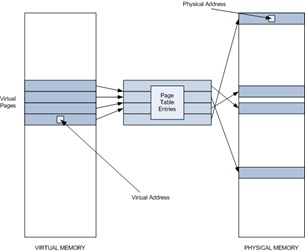
La ora actuală se folosesc două versiuni de Windows : pe 32 și pe 64 de biți. În versiunile pe 32 biți, procesele au acces la un spațiu de 2^32 adrese distincte, deci programele rulate pe 32 de biți pot accesa maximum 4 GB de memorie. Pe de altă parte, în versiunile pe 64 de biți, procesele au la dispoziție un spațiu de memorie de 2^64 adrese distincte.

Windows împarte spațiul de adrese din memorie în două: spațiu pentru folosirea in **modul kernel** , respectiv spațiu pentru folosirea în **modul user.** Spațiul de sistem, sau spațiul kernel, este porțiunea din spațiul adreselor în care se află sistemul de operare si driverele. Acest spațiu poate fi accesat numai de modul kernel. Firele de execuție din modul user nu pot accesa direct datele din alt proces sau spațiul de adrese alocat sistemului.



Urmărind imaginea de mai sus se poate vedea cum este impărțită o memorie de 4GB. Windows alocă jumătatea de jos a spațiului de adrese ( de la 0x00000000 la 0x7FFFFFFF) ca spațiu pentru diverse procese rulate in user-mode și cealaltă jumătate ( de la 0x80000000 la 0xFFFFFFFF) pentru sistemul de operare.

Spațiul de adresare virtuală alocat fiecărui proces poate fi mai mare sau mai mic decât memoria fizică disponibilă. Fiecare pagină din memoria virtuală este indexată intr-un tabel cu care se face asocierea cu pagina fizică corespunzătoare.



Primul alocator se ocupă de gestionarea spațiului virtual de adrese. Cu ajutorul acestuia se efectueaza operații la nivel de pagină. De exemplu, rezervarea unui interval din spațiul virtual de adrese. O rezervare nu va ocupa pagini fizice din memoria calculatorului, dar va preveni folosirea blocului respectiv pentru alte operații până la eliberarea acestuia. Rezervarea nu influențează cu nimic alocările pentru alte procese. O rezervare îi permite unui proces să păstreze un interval de adrese pentru o structură de date cu dimensiune variabilă, care se poate mări în funcție de necesități. Rezervarea unui interval se face cu funcția **VirtualAlloc** prin transmiterea valorii **MEM\_RESERVE.**

O altă operație importantă este maparea unui set de pagini. Prin mapare se înțelege asocierea unui interval de adrese rezervat anterior cu un set de pagini fizice din memorie. Dacă în urma mapării nu se ajunge la o alocare efectivă de memorie, se consumă pagini din memoria sistemului. Operația se realizează tot cu **VirtualAlloc,** prin transmiterea valorii **MEM\_COMMIT.**  Pentru a rezerva și a mapa paginile într-un singur pas prin combinarea celor două valori, se apelează **VirtualAlloc** cu **MEM\_COMMIT | MEM\_RESERVE**. La mapare paginile primesc atribute de protecție care specifică daca paginile respective sunt accesibile pentru execuție, citire sau scriere. Atributele de protecție pot fi modificate cu ajutorul funcției **VirtualProtect.**

**Sintaxa C++:**

LPVOID WINAPI VirtualAlloc(

\_In\_opt\_ LPVOID lpAddress, //adresa de început a regiunii de alocat

\_In\_     SIZE\_T dwSize, //dimensiunea spațiului in octeți

\_In\_     DWORD  flAllocationType, /\*tipul alocării: MEM\_COMMIT, MEM\_RESERVE, MEM\_RESET sau MEM\_RESET\_UNDO \*/

\_In\_     DWORD  flProtect //tipul de protecție implementată pentru spatial alocat

);

Paginile rezervate și/sau mapate pot fi eliberate cu ajutorul funcției **VirtualFree,** folosind valoarea **MEM\_RELEASE**, respectiv **MEM\_DECOMMIT.** La ștergerea rezervării trebuie specificat intervalul de adrese alocat anterior cu VirtualAlloc, dar nu contează dacă acesta conține doar pagini rezervate, doar pagini mapate sau un amestec de cele două.

Un alt alocator important din Windows oferă funcții de heap. Funcțiile pentru heap se folosesc pentru alocarea unor blocuri de dimensiuni relativ reduse de memorie. Fiecare proces dispune de un heap accesibil prin functia **GetProcessHeap.** Aplicațiile pot crea heapuri cu funcția **HeapCreate.** Alocarea de memorie se face prin intermediul funcției **HeapAlloc**. Alte funcții sunt **HeapReAlloc** (utilizată pentru redimensionare), **HeapFree** și **HeapDestroy** (ambele sunt folosite pentru eliberare).

**Sintaxa C++:**

**HANDLE WINAPI HeapCreate(**

\_**In\_ DWORD  flOptions**, /\* opțiuni de alocare; poate avea valoarea 0 sau HEAP\_CREATE\_ENABLE\_EXECUTE , HEAP\_GENERATE\_EXCEPTIONS, HEAP\_NO\_SERIALIZE\*/

\_**In\_ SIZE\_T dwInitialSize,** /\*dimensiunea inițială a heapului în octeți, trebuie să fie mai mica decât dwMaximumSize\*/

**\_In\_ SIZE\_T dwMaximumSize** /dimensiunea maxima a heapului in octeți

**);**

**LPVOID WINAPI HeapAlloc(**

**\_In\_ HANDLE hHeap,**

**\_In\_ DWORD  dwFlags,**

**\_In\_ SIZE\_T dwBytes**/\*numărul de octeți destinați alocării; dacă parametrul hHeap specifică un heap de dimensiune constantă, dwBytes trebuie să fie mai mic decât 0x7FFF8\*/

**);**

**BOOL WINAPI HeapDestroy(**

**\_In\_ HANDLE hHeap**

**);**

BOOL WINAPI HeapFree(

\_In\_ HANDLE hHeap,

\_In\_ DWORD  dwFlags,

\_In\_ LPVOID lpMem

);

**3. Probleme în lucrul cu memoria**

Cele mai des întâlnite probleme în lucrul cu memoria sunt:

* leak-urile de memorie;
* accesul invalid la memorie.

Cauzele principale ale aparițiilor acestor problem sunt lucrul cu heapul, lucrul cu pointerii, folosirea unor apeluri de sistem pentru alocarea sau dealocarea memoriei.

**3.1 Accesul invalid**

Accesul invalid presupune accesarea unor porțiuni de memorie care nu au fost incă alocate sau au fost eliberate. În general accesarea unei zone invalide de memorie rezultă într-o eroare și terminarea procesului. În cazul în care eroarea apare într-o pagină validă dar la o adresă invalidă, sistemul nu va putea sesiza acțiunea ca fiind invalidă, deoarece alocarea memoriei se face la nivelul paginii. Astfel de accese pot duce la pierderea consistenței memoriei alocate și la coruperea heapului.

Acest tip de problemă poate fi prevenită prin folosirea unor utilitare precum **GDB** ( ajută la detecția zonei de acces invalid), **Valgrind**(o suită de utilitare ce include **Memcheck** pentru detectarea erorilor de lucru cu memoria) și **mcheck** (verifică consistența heapului)

**3.2 Leak-urile de memorie**

Un leak (scurgere) de memorie apare dacă un program omite să elibereze o zonă alocată de memorie sau dacă un program nu poate dealoca un spațiu de memorie. Efectul cel mai devastator al scurgerilor de memorie este reducerea cantității de memorie din sistem. În situații extreme se poate ajunge chiar la consumarea întregii memorii si imposibilitatea de funcționare a aplicațiilor.

Utilitare care pot ajuta la detecția scurgerilor de memorie sunt **mtrace** si **Valgrind.**

**4. Alocarea memoriei în Linux**

Există trei funcţii de bibliotecă care se pot folosi în Linux pentru a aloca memorie: **malloc(), calloc() si realloc().** Dealocarea ei se face prin intermediul funcției **free()**. Aceste funcţii reprezintă apeluri de bibliotecă şi rezolvă cererile de alocare şi dezalocare de memorie, pe cât posibil, în user space.

**#include <**stdlib.h**>**

**void \*malloc(size\_t***size***);**

**void free(void***\*ptr***);**

**void \*calloc(size\_t***nmemb***, size\_t***size***);**

**void \*realloc(void***\*ptr***, size\_t***size***);**

Funcția **malloc()** este cea mai simplă dintre cele trei. Această funcţie alocă mărimea octeţilor si returnează un pointer memoriei alocate. Memoria nu este iniţializată! Daca *size* este 0, atunci **malloc()**returnează fie NULL, sau o valoare unică a pointerului care poate fi trimisă mai departe către **free().**

**Exemplu:**

*struct coord {/ \* Coordonatele 3D \* /*

*int x, y, z;*

*} coordonatele \*;*

*unsigned count int; / \* Câte avem nevoie \* /*

*size\_t scor; / \* Valoare totală de memorie \* /*

*/ \* ... Determina scorul ... \* /*

*scor = count \* sizeof (struct coord); / \* Câti bytes să aloce \* /*

*coordonatele = (struct coord \*) malloc (valoare); / \* Obține spațiul \* /*

*dacă (coordonatele == NULL) {*

*/ \* Raport de eroare, recuperarea sau renunţare \* /*

*}*

*/ \* ... Utilizare coordonatele ... \* /*

Funcţia **calloc()**alocă memorie pentru un vector al elementelor *nmemb*, cu *size*măsurat în octeţi pentru fiecare element şi returnează un pointer memoriei alocate. Memoria este setată zero. Daca *nmemb* sau *size* este 0, atunci **calloc()**returneaza fie NULL, fie o valoare unică a pointerului care poate sa fie transmisă mai tarziu către **free().**

O posibilă implementare a functiei **calloc()** este prezentată mai jos:

*void \* calloc (size\_t nmemb, size\_t dimensiune)*

*{*

*void \* p;*

*size\_t total;*

*total = nmemb \*dimensiune; /\*Calculează dimensiune\*/*

*p = malloc (total); /\* Alocă memorie\*/*

*if (p! = NULL) /\*În cazul în care functionează. \*/*

*memset (p, '\ 0', total); /\*Se umple cu zerouri\*/*

*return p; /\*Valoarea returnată este NULL sau pointer\*/*

*}*

Funcţia **realloc()**schimbă marimea blocului de memorie care arata către *ptr* cu lungimea în octeti a lui *size.* Continutul va fi neschimbat in intervalul de la începutul regiunii pană la minimul dintre dimensiunile vechi si noi. Dacă noua dimensiune este mai mare decat cea veche, memoria adaugată nu va fi iniţializată. Daca *ptr* este NULL, atunci apelul este echivalent cu *malloc(size),* pentru toate valorile ale *size.* Cand size este egal cu zero, si *ptr* nu este NULL, atunci apelul este echivalent cu *free(ptr).* Cu excepţia cazului cand *ptr* este NULL, trebuie să fi fost returnat de un apel anterior la *malloc (), calloc ()* sau *realloc ().*

Se poate folosi *realloc()*,şi pentru a reduce mărimea unui bloc de memorie dandu-i o mărime mai mica decat cea alocată iniţial. Continuând cu exemplul *coordonatelor,* un exemplu de cod ar fi următorul:

*int noua\_numarare;*

*size\_t noua\_suma;*

*struct coord \*newcoords;*

*/\* Seteaăa noua\_numarare, de exemplu \*/*

*noua\_numarare = count \* 2; /\* Dublează spaţiul\*/*

*noua\_suma = noua\_numarare \* sizeof(struct coord);*

*noi\_coordonate = (struct coord \*) realloc(coordonatele, noua\_suma);*

*if (noi\_coordonate == NULL) {*

*/\* Raportează eroare, recuperare sau renunţă\*/*

*}*

*coordonatele= noi\_coordonate;*

*/\* Continuă folosirea coordonatelor \*/*

Funcţia free() are prototipul: **void free (void\* ptr);**

Aceasta este folosită pentru a dealoca memorie care a fost alocată inainte în una din cele 3 metode care a fost menţionată precedent. Primeşte un pointer ce refera memoria alocată. Un bloc de memorie poate fi eliberat si cu metoda **realloc()** daca se apelează pentru marimea 0. Dupa ce un bloc a fost eliberat, acesta nu mai trebuie folosit. Daca memoria este folosită după ce a fost eliberată, este probabil ca programul sa nu cedeze pe moment, ci mai târziu, atunci cand efectele acestui uz cauzează altă problemă. Aceste tipuri de buguri sunt mai greu de depistat.

Singurul argument este un pointer obtinut inainte de la una din celelalte rutine de alocare. Este mai sigur, desi deloc necesar să creeăm un pointer NULL catre free():

*free(coordonatele);*

*coordinates = NULL; /\* Nu este necesar, dar este o idee bună sa îl lăsăm aşa \*/*

Odată ce *free(coordonatele)* este apelată, memoria indicată prin *coordonatele* depăşeşte limitele. ‘Aparţine’ acum subrutinelor de alocare, si sunt libere sa-l gestioneze cum cred de cuviinţă. Ele pot schimba conţinutul memoriei sau chiar să o elibereze din spaţiul de adrese al proceselor. Există, astfel, mai multe erori comune la care trebuie să fim atenți pentru funcția **free ():**

* **Accesarea memoriei eliberate**

Dacă rămân neschimbate, coordonatele continuă să arate către memorie care nu mai face parte din aplicaţie. Aceasta se numește un pointer marionetă. În multe sisteme, se poate continua accesarea memoriei, cel puțin până cand mai multă memorie este alocată sau eliberată. În multe altele, un astfel de acces nu va funcționa.

În concluzie, accesarea memoriei eliberate este o idee rea: Nu este portabilă sau de încredere, și standardele de codificare GNU nu o permit. Din acest motiv, este o idee bună pentru a seta imediat variabila pointer programului sa fie NULL. Dacă se incearcă apoi accidental pentru a accesa memoria eliberată, programul va eșua imediat cu o eroare de segmentare.

* **Eliberarea aceluiași pointer de două ori**

Acest lucru duce la "comportament nedefinit". Odată ce memoria a fost cedata rutinelor de alocare, acestea pot fuziona blocul liber cu alt spatiu care îl avem la dispozitie. Eliberarea a ceva ce a fost deja eliberat poate duce la confuzii sau cedează în cel mai bun caz , și pot duce la probleme de securitate.

* **Depășiri tampon și subdepășiri**

Accesarea memoriei în afara unei bucăti alocate, duce de asemenea la un comportament nedefinit, din nou, pentru că acest lucru este probabil sa contabilizeze informații sau, eventual, memorie care nu e nici măcar în spațiul de adrese. Scrierea în astfel de memorie este mult mai problematică, deoarece este posibil să distrugă datele contabilizate.

* **Imposibilitatea de a elibera memorie**

Orice memorie dinamică, care nu este nevoie să fie eliberată. În special, memorie care este alocată în interiorul unei bucle sau apeluri de funcții recursive sau profund imbricate care ar trebui gestionate și eliberate cu grijă. Imposibilitatea de a avea grijă de acestea duce la scurgeri de memorie, în care memoria procesului poate crește fără limite; în cele din urmă, procesul se inchide din cauza lipsei de memorie.

Această situație poate fi deosebit de dăunătoare dacă memoria este alocată pe înregistrare de intrare sau ca o altă funcție de intrare: scurgerea de memorie nu va fi observată atunci când rula pe intrări mici, dar poate deveni brusc evidentă.Această eroare este chiar mai rea pentru sistemele care trebuie să se execute în mod continuu, cum ar fi sistemele de comutare telefonică. O scurgere de memorie care blochează un astfel de sistem poate duce la deteriorarea monetară sau alta mai semnificativă.

Chiar dacă programul nu se distruge din cauza lipsei de memorie, programul care creste in mod constant suferă in materie de performanță, deoarece sistemul de operare trebuie să gestioneze menținerea datelor care utilizează în memoria fizică. În cel mai rău caz, acest lucru poate duce la un comportament în care sistemul de operare este atât de ocupat sa deplaseze conținutul spațiului de adrese în și din memoria fizică încât nu se realizeaza nimic în mod concret.

**5. Detecția și corecția erorilor**

Tehnicile de detecție si corecție a erorilor sunt metode care permit livrarea sigură a datelor. Ideea generală din spatele acestor metode este adăugarea de date redunante pe care receptorul le poate folosi pentru a verifica consistența datelor primite.

Cea mai simplă metodă de detecție este **bitul de paritate**. Aceasta poate detecta erorile simple, dar nu și erorile multiple, și nici nu poate corecta erorile detectate. Se adaugă câte un bit la fiecare cuvânt din memorie. Bitul suplimentare reține dacă biții fiecărui cuvânt conțin un număr par sau impar de cifre 1și este memorat sau trimis odată cu cuvântul. La recepția sau citirea cuvântului se recalculează paritatea și se compara cu bitul citit sau recepționat. Dacâ cele doua rezultate nu coincid, înseamnâ că s-a produs o eroare.

O altă metodă este **Cererea automată de repetare (engleză : Automatic repeat request – ARQ)** . Această metodă constă în utilizarea confirmărilor trimise de receptor și timpii de întrerupere (perioade fixe de timp în care se așteaptă confirmarea). Dacă transmițătorul nu primește o confirmare înainte de expirarea timpului de întrerupere, informația este retransmisă până când se primește confirmarea sau este atins un număr predefinit de retransmiii. Tipurile de ARQ includ : **stop-and-wait ARQ, go-back-n ARQ** și **selective repeat ARQ.**

**Bibliografie**

<http://www.biblioteca-digitala.ase.ro/biblioteca/pagina2.asp?id=cap3>   
<http://rgyorodi.webhost.uoradea.ro/modules/OS/OS2/files/OS213.pdf>

<http://andrei.clubcisco.ro/cursuri/3so/labs/v2/Laborator_04_-_Gestiunea_memoriei.pdf>   
<http://www.scribd.com/doc/58327708/Alocarea-Dinamica-a-Memoriei#scribd>   
<http://gribblelab.org/CBootcamp/7_Memory_Stack_vs_Heap.html>   
<https://msdn.microsoft.com/en-us/library/ms810603.aspx>

<http://blogs.technet.com/b/askperf/archive/2007/02/23/memory-management-101.aspx>

<http://so-usb.blogspot.ro/p/gestiunea-memoriei.html>

<https://msdn.microsoft.com/en-us/library/windows/desktop/aa366599>

<http://cursuri.cs.pub.ro/~so/Laboratoare/04.%20Gestiunea%20memoriei.pdf>

<http://www.competentedigitale.ro/it/it6.html>

<http://cmc.site11.com/ase/Informatica/bti/tema_finala/SO.html>

<http://andrei.clubcisco.ro/cursuri/3so/labs/04.%20Gestiunea%20memoriei.pdf>

<http://andrei.clubcisco.ro/cursuri/f/f-sym/1uso/2011-2012/11_USO_curs_01.pdf>

<http://www.informit.com/articles/article.aspx?p=173438&seqNum=2>

<http://linux.die.net/man/3/malloc>