UNIVERSITATEA POLITEHNICA BUCUREȘTI

FACULTATEA DE ELECTRONICĂ, TELECOMUNICAȚII ȘI TEHNOLOGIA INFORMAȚIEI

SISTEME DE OPERARE

**IMPLEMENTAREA GESTIUNII MEMORIEI**

**433 A Petrescu Maria**

**Avram Tiberiu**

**Dimofte Cosmin**

**Pirea Radu**

**Cuprins**

**1.Introducere - Dimofte Cosmin**

* 1. Memoria virtuală

1.1.1 Definiție

1.1.2 Scopurile , utilizarile si efectele utilizarii memoriei virtuale

1.1.3 Tipuri de memorie

1.1.4 Memoria privată

1.1.5 Memoria partajată

1.1.6 Memoria anonimă

1.1.7 Fișier sprijinit și swap

1.2 Pagini

1.2.1 Introducere

1.2.2 Paginarea

**2.Algoritmi de gestiune a memoriei - Pirea Radu**

* 1. Algoritmul prietenului

2.1.1Cum funcționează

* 1. Implementare si eficienta
  2. Alocarea memoriei sub forma de stiva
  3. Garbage collection

2.4.1 Principiile Garbage collection

2.4.2 Avantaje

2.4.3 Dezavantaje

**3.Lucru cu memoria - Petrescu Maria**

3.1 Algoritmi de inlocuire a paginilor

3.1.1 Algoritmul NRU

3.1.2 Algoritmul FIFO

3.1.3 Algoritmul LRU

3.1.4 Algoritmul optim

3.1.5 Algoritmul Ceasului

3.2 Probleme in lucrul cu memoria

3.2.1 Tratare acces invalid

3.2.2 Tratare pierderi de spatii de memorie

3.2.2.1Valgrind

3.2.2.2 Mtrace

3.2.3 Dublă dealocare

**4.Implementarea gestiunii memoriei la Windows - Avram Tiberiu**

4.1. Memoria virtuala si memoria fizica

4.1.1 Memoria virtuala

4.1.2 Castigarea de spatiu pe HARD DISK prin modificarea memoriei virtuale

4.2 Alocarea memoriei in Windows

4.2.1 Nucleul: Alocarea memoriei

4.2.2 Nucleul: alocarea de spaţiu pe disc

4.2.3 Tipuri de alocatoare

4.2.3.1 Un alocator dinamic foarte simplu

4.2.3.2 Alocatorul cu hartă de resurse

4.2.3.3 Alocatorul cu puteri ale lui 2

4.2.3.4 Alocatorul ``slab''

1. **Introducere - Dimofte Cosmin**

**1.1 Memoria virtuala**

**1.1.1 Definitia memoriei virtuale**

Memoria virtuală este o caracteristică a unui sistem de operare (OS), care permite unui computer să compenseze deficitul de memorie fizică prin transferul temporar de pagini de date din memorie cu acces aleator (RAM) pentru stocare pe disc.

În cele din urmă, sistemul de operare va trebui pentru să preia datele mutate pe discul de stocare temporar - singurul motiv pentru care paginile au fost mutate din RAM in discul de stocare este deoarece se aglomera memoria RAM. Pentru a rezolva problema, sistemul de operare va trebui să se mute alte pagini pe hard disk, astfel există loc pentru a aduce înapoi în paginile de care are nevoie imediat de pe discul temporar. Acest proces este cunoscut sub numele de paginare sau pompare și spațiul de stocare temporară de pe hard disk este numit fișier de swap .

Pomparea care se întâmplă atât de repede încât utilizatorul final nu știe că se întâmplă, se efectuează de către unitatea de management al memoriei calculatorului (MMU). MMU-ul poate folosi unul sau mai multi algoritmi pentru a alege o pagina.

Într-un mediu de calcul virtualizat, administratorii pot folosi tehnici de management de memorie virtuale să aloce memorie suplimentară la unei mașini virtuale (VM), care a rămas fără resurse. Astfel de tactici de management de virtualizare pot îmbunătăți performanța VM și flexibilitate de management. [[2]](http://searchstorage.techtarget.com/definition/virtual-memory)

**1.1.2 Scopurile , utilizarile si efectele utilizarii memoriei virtuale**

Pe sistemele de operare moderne, fiecare proces trăiește în propriul spațiu de alocare de memorie. În loc de adrese de memorie de cartografiere direct la adresele hardware, sistemul de operare servește ca un strat de abstractizare hardware și creează un spațiu de memorie virtuală pentru fiecare proces. Maparea dintre adresa de memorie fizică și adresa virtuala se face de către CPU, folosind un tabel de pe proces menținut de kernel (de fiecare dată când kernel-ul schimbă procesul care rulează pe un miez CPU specific, se schimbă masa traducere care CPU ).

Memoria virtuală are mai multe scopuri. În primul rând, permite izolarea de proces. Un proces în userland pot exprima doar memoria accesele ca adresele în memoria virtuală. Ca urmare accesa doar date care au fost deja mapate in memorie.

Al doilea scop este captarea hardware. Kernel-ul este liber de a schimba adresa fizică la care este mapat cu o adresă virtuală. Se poate alege, de asemenea, nu să furnizeze orice memorie fizică pentru o adresă virtuală specifică până când devine de fapt nevoie. Mai mult, se poate schimba în memoria pe disc, atunci când acesta nu a fost folosit pentru o perioadă lungă de timp, iar sistemul nu mai are memorie fizică disponibila. Aceasta oferă la nivel global o mulțime de libertatea

kernel-ului, singura ei constrângere este că, atunci când programul citeste memoria este de fapt ceea ce consideră ca scris anterior acolo.

Al treilea scop este posibilitatea de a da adrese de lucru care nu sunt de fapt în memoria RAM. Acesta este principiul din spatele mmap și cartografiere fișierol. Puteți da o adresă de memorie virtuală într-un fișier, astfel încât să poată fi accesare ca și în cazul în care a fost un buffer de memorie. Aceasta este o abstracție foarte utila, care vă ajută menținerea codului destul de simplu și, din moment ce pe sistemele pe 64 de biți este un spatiu virtual imens, dacă se doreste, se poate mapa un hard disk întreg în memoria virtuală.

Al patrulea scop este schimbul. Deoarece kernel-ul știe ce proces este mapat în spațiul virtual al diferitelor procese de funcționare, se poate evita încărcarea de două ori în memorie și să facă adresele virtuale ale proceselor care utilizează aceleași resurse sa indice aceeași memorie fizică (chiar dacă adresa virtuala reala. [[2]](http://searchstorage.techtarget.com/definition/virtual-memory) [[4]](http://web.info.uvt.ro/~fortis/LICENTA/SO/Lectures/SistemeOperare2009_Memorie_1.pdf)

**1.1.3 Tipuri de memorie**

Nu toată memoria alocată în spațiul de memorie virtuală este aceeași. Putem clasifica prin intermediul a două axe: prima axă este dacă memoria este privată (specifică acestui proces) sau partajată, a doua axă este dacă memoria este sprijinită de fișier sau nu (în cazul în care se spune fie anonimă). Acest lucru creează o clasificare cu 4 clase de memorie.[[5]](https://techtalk.intersec.com/2013/07/memory-part-1-memory-types/)

**1.1.4 Memoria privata**

Memoria privată este, după cum spune și numele, memoria care este specifică pentru acest proces. Cele mai multe dintre memoriile care există intr-un program sunt de fapt memorii private.

Deoarece modificările aduse în memoria privată nu sunt vizibile altor procese, este supusă pentru copy-on-write. Ca un efect secundar, acest lucru înseamnă că, chiar dacă memoria este privată, mai multe procese s-ar putea să împărtășească aceeași memorie fizică pentru a stoca datele. În special, acesta este cazul pentru fișierele binare și biblioteci partajate. O afirmatie falsă comună este că KDE are nevoie de o multămemorie RAM, deoarece fiecare proces are sarcini Qt și kdelibs, cu toate acestea, datorită mecanismului de COW, toate procesele vor folosi aceeași memorie fizică exact pentru partile read-only ale acestor librarii.

În cazul în care memoria este privată - sprijinită pe fișier, modificările aduse de proces nu sunt scrise înapoi in dosarul de bază, însă modificările aduse la dosar pot sau nu fi puse la dispoziția procesului. [[5]](https://techtalk.intersec.com/2013/07/memory-part-1-memory-types/)

**1.1.5 Memorie partajată**

Memoria partajată este ceva conceput pentru comunicarea inter-proces. Ea poate fi creată doar cand se doreste în mod explicit utilizarea mmap right () apel sau un apel dedicat (SHM \*). Când un proces scrie într-o memorie partajată, modificarea este văzută de toate procesele care au aceeași hartă de memorie.

În cazul în care memoria este sprijinită de fisier, orice proces de cartografiere al dosarului va vedea modificările în fișier întrucât aceste modificări sunt propagate prin fișierul în sine. [[5]](https://techtalk.intersec.com/2013/07/memory-part-1-memory-types/)

**1.1.6 Memoria anonima**

Memoria anonimă se află pur în memoria RAM. Cu toate acestea, nucleul nu va mapa memoria la o adresă fizică înainte ca aceasta sa fie de fapt scrisă. Ca o consecință, memoria anonimă nu adaugă presiune asupra nucleului înainte de a fi efectiv utilizată. Acest lucru permite unui proces de a "rezerva" o mulțime de memorie în spațiul de adrese de memorie virtuală, fără a folosi RAM. Ca urmare, kernel-ul vă permite sa rezervați mai multă memorie decât este în ​​realitate disponibilă. Acest comportament este adesea referire ca supra-angajare (sau overcommitment memory). [[5]](https://techtalk.intersec.com/2013/07/memory-part-1-memory-types/)

**1.1.7 Fișier sprijinit și Swap**

Când o hartă de memorie este sprijinită de fișier, datele sunt încărcate de pe disc. De cele mai multe ori, acesta este încărcat la cerere, cu toate acestea, se pot da indicii kernel-ului, astfel încât să poată predescărca memoria înainte de citire. Acest lucru ajută păstrarea programului rapid , atunci când știți că aveți un anumit model de accesare . Pentru a evita folosirea a prea mult RAM,

atunci când sistemul are nevoie de memorie fizică disponibila, kernel-ul va încerca să deplaseze unele date din memoria RAM pe disc. Dacă memoria reprezintă fișierul sprijinit și partajat, acest lucru este destul de ușor. Deoarece fișierul este sursa de date, acesta este doar scos din RAM, atunci va fi citit data viitoare si va fi încărcat de la dosar.

Kernel-ul poate alege, de asemenea, să elimine memorie anonimă privata din RAM. Caz în care datele sunt scrise într-un loc specific pe disc.

Datorită utilizării unui spațiu de adresare virtual, schimbarea de pagini este complet transparenta. [[4]](http://web.info.uvt.ro/~fortis/LICENTA/SO/Lectures/SistemeOperare2009_Memorie_1.pdf) [[5]](https://techtalk.intersec.com/2013/07/memory-part-1-memory-types/)

**1.2. Pagini**

**1.2.1 Introducere**

Memoria virtuală este împărțită în pagini.O dimensiune de pagină este impusă de CPU și este, de obicei, 4KB. Ce înseamnă acest lucru este că managementul memoriei în kernel se face cu o granularitate de o pagină. Când este nevoie de memorie, kernel-ul va oferi una sau mai multe pagini, atunci când se eliberează memorie, se eliberează una sau mai multe pagini. Fiecare API fine-grained (de exemplu, malloc) este pus în aplicare în terenul de utilizare. [[2]](http://searchstorage.techtarget.com/definition/virtual-memory)

Pentru fiecare pagină alocată, nucleul pastrează un set de permisiuni: pagina poate fi ușor de citit, scris și / sau executat (de reținut că nu toate combinațiile sunt posibile). Aceste permisiuni sunt setate fie în timp ce are loc cartografierea de memorie sau folosind mprotect () apelul de sistem, după aceea. Paginile care nu au fost alocate încă, nu sunt accesibile. Atunci când încercați să efectuați o acțiune interzisă pe o pagină (de exemplu, citirea datelor de la o pagină fără permisiunea citire), veți declanșa (pe Linux) o eroare de segmentare. Ca o notă separată, puteți vedea că, deoarece are o granularitate de o pagină, puteți efectua out-of-tampon acces care nu conduc la o segfault. Cazul de utilizare mai cunoscut de COW este fork (). Pe sistemele Unix-like, fork () este apelul de sistem care creează un proces de duplicare a celui actual. Când fork () execută, ambele procese trebuie să continue exact din același punct, cu aceleași fișiere deschise și aceeași memorie. Datorită COW, fork () nu va duplica în memorie un proces atunci când îl detine, numai datele care sunt modificate de către oricare părinte al copilului se duplica în memoria RAM. Deoarece cele mai multe utilizări ale fork () sunt urmate imediat de un apel la exec (), care invalidează întreaga memorie virtuală, mecanismul COW evită o copie inutilă a memoriei procesului părinte.

Un alt efect secundar, este că fork () creează o imagine de memorie (privată) a unui proces la un cost mic. Dacă doriți să efectuați o operație în memoria unui proces fără a lua riscul de a fi modificată, și nu doriți să se adăuge un mecanism de blocare costisitoare și predispus la erori, doar fork() comunică rezultatul de calcul înapoi la procesul de părinte (prin cod retur, dosar, memorie partajata, pipe, ...).

Acest lucru va funcționa extrem de bine, atâta timp cât calculul este suficient de rapid, astfel încât o mare parte a memoriei rămâne împărțită între atât procesele mamă cat și procesele copil. Acest lucru ajută, de asemenea, la păstrarea codului simplu, complexitatea este ascunsă în codul memoriei virtuale a kernel-ului. [[2]](http://searchstorage.techtarget.com/definition/virtual-memory)

**1.2.2 Paginarea**

In sistemele de operare pe calculator, paginarea este una dintre schemele de gestiune a memoriei prin care un computer inamagazineaza și preia datele din stocarea secundara pentru utilizarea în memoria principală. In schema de management de memorie de paginare, sistemul de operare preia datele din depozitul secundar în blocuri de același dimensiune numite pagini. Principalul avantaj al paginarii peste segmentarea memoriei este că permite spațiul de adrese al unui proces fizic de a fi necontinu. Înainte ca paginarea sa intre în folosință, sistemele trebuiau să se potrivească programe întregi în depozitare adiacenta, care a cauzat diverse probleme de depozitare și de fragmentare. [[1]](http://en.wikipedia.org/wiki/Paging)

Paginarea este o parte importantă a implementării memorie virtuale în majoritatea sistemelor de operare de uz general , permițându-le să folosească depozitare secundare pentru datele care nu se încadrează în memoria cu acces aleator.

Principalele funcții ale paginarii sunt efectuate atunci când un program încearcă să acceseze paginile care nu sunt mapate în prezent în memoria fizică (RAM). Această situație este cunoscută ca un defect de pagină. Sistemul de operare trebuie apoi să preia controlul și se ocupe de pagină, într-un mod invizibil pentru program. Prin urmare, sistemul de operare:

-Determina locația datelor în depozitul secundar.

-Obține un cadru de pagină goală în RAM pentru a-l utiliza ca un container pentru datel.

-Încarca datele solicitate în cadrul paginii disponibile

-Actualizeaza tabelul de pagină pentru a se referi la noul cadru de pagina.

-Revine controlul programuluo, transparent va reîncerca instrucțiunea care a cauzat eroarea de pagina.

-Dacă nu există suficientă memorie RAM disponibilă când se obține un cadru pagină goală, un algoritm de înlocuire pagină se utilizează pentru a alege un cadru pagină existentă pentru evacuare. În cazul în care cadrul pagină evacuat a fost alocat dinamic în timpul execuției unui program, sau dacă face parte din segmentul de date a unui program și a fost modificat de când a fost citit în memoria RAM (cu alte cuvinte, în cazul în care acesta a devenit "murdare"), el trebuie să fie scris într-o locație de stocare secundar înainte de a fi eliberat. În caz contrar, conținutul cadrului paginii în memoria RAM este același ca și conținutul paginii în depozitarea secundar, deci nu trebuie să fie scris la depozitarea secundara. Dacă, într-o etapă ulterioară, se face o trimitere la pagina de memorie, un alt defect de pagină va avea loc și un alt frame din pagina goala trebuie să fie obținut, astfel încât conținutul paginii în depozitul secundar sa poata fi din nou citit în memoria RAM. [[1]](http://en.wikipedia.org/wiki/Paging)

Sistemele eficiente de paging trebuie să stabilească cadrul pagină goală alegând unul care este cel mai puțin probabil să fie folosit intr-un timp apropiat. Există diversi algoritmi de înlocuire pagina care încearcă să facă acest lucru. Cele mai multe sisteme de operare utiliza o aproximare a algoritmului (LRU) înlocuirea paginii utilizate cel mai recent (ÎFP în sine nu poate fi pusă în aplicare pe hardware-ul actual) sau un algoritm bazat pe set de lucru.

Pentru a spori și mai mult capacitatea de reacție, sistemele de paging pot utiliza diverse strategii pentru a anticipa care pagini vor fi necesare în curând. Astfel de sisteme vor încerca să încarce pagini în memoria principală preventiv, înainte ca un program sa le refere. [[1]](http://en.wikipedia.org/wiki/Paging)

.

BIBLIOGRAFIE

[1]<http://en.wikipedia.org/wiki/Paging>

[2]<http://searchstorage.techtarget.com/definition/virtual-memory>

[3]<http://www.techopedia.com/definition/2280/secondary-memory>

[4]<http://web.info.uvt.ro/~fortis/LICENTA/SO/Lectures/SistemeOperare2009_Memorie_1.pdf>

[5] https://techtalk.intersec.com/2013/07/memory-part-1-memory-types/

**2.Algoritmi de gestiune a memoriei- Pirea Radu**

# 2.1 Algoritmul prietenului

Tehnica de alocare de memorie "prietenul" este un algoritm de alocare de memorie care împarte memoria în partiții pentru a încerca satisfacerea unei cereri de alocare de memorie corespunzător posibil. Aceasta divizeaza memoria în două părți pentru a încerca să dea o potrivre perfecta. În conformitate cu Donald Knuth, sistemul "prietenul" fost inventat în 1963 de Harry Markowitz, care a castigat Premiul Nobel 1990 Memorial în economie, și a fost descris pentru prima data de Kenneth C. Knowlton (publicat 1965). Alocarea de memorie "prieten" este relativ ușor de implementat. Aceasta susține divizarea și coalescența de blocuri de memorie limitata, dar eficienta. [[1]](https://en.wikipedia.org/wiki/Buddy_memory_allocation)

## 2.1.1Cum funcționează

Există diferite forme ale acestui sistem, în care fiecare bloc este împărțit în 2 blocuri mai mici, cea mai simplă și cea mai comună varietate. Fiecare bloc de memorie în acest sistem are o ordine, unde comanda este un întreg variind între 0 până la o limită superioară specificată. Marimea unui bloc de ordin n este proporțională cu 2^n, astfel încât blocurile sunt exact doua ori mai mari decât blocurile care sunt de ordin inferior. Dacă blocurile au dimensiunea puterilor lui 2, ele fac calculul adresei simplu, pentru că toți prietenii sunt aliniati pe limitele de adresa de memorie care sunt puteri de lui 2. Când un bloc mai mare este împărțit, acesta este împărțit în două blocuri mai mici, iar fiecare bloc mic devine un prieten unic al altuia. Un bloc împărțitt se poate uni numai cu blocul său "prietenul" unic, care apoi reformeaza blocul mai mare din care au fost impartite.

Dimensiunea celui mai mic bloc este determinata. Daca nu exista nicio limita inferioara (de exemplu, alocări de biți de dimensiuni ce au fost posibile), nu ar exista o multime de memorii de calcul pentru sistem pentru a urmări care parti ale memoriei sunt alocate și nealocate. Totuși, o limită destul de scăzută poate fi astfel încât resturile de memorie medie pe alocare (privind alocările) sunt minimizate. De obicei limita inferioară ar fi suficienta pentru a minimiza spațiul mediu irosit pe alocare mica. Cea mai mică dimensiune a blocului este dimensiunea unui ordin 0, astfel încât toate comenzile mai mari sunt exprimate ca puterea lui 2.

Programatorul trebuie apoi să decidă cu privire la cel mai inalt ordin posibil care poate încăpea în spațiul disponibil rămas memoriei. Deoarece dimensiunea totală de memorie disponibilă într-un sistem informatic dat, nu poate fi un multiplu de puteri ale lui 2, cea mai mare dimensiune de bloc nu acopera întreaga memorie a sistemului. De exemplu, în cazul în care sistemul a avut 2000K de memorie fizică și ordin 0, mărimea blocului a fost 4K, limita superioară de ordinul ar fi de 8, deoarece un bloc de ordine 8 (256 ordin 0 blocuri, 1024K) este cel mai mare bloc care se va potrivi în memorie. Prin urmare, este imposibil să se aloce întreaga memorie fizică într-o singură bucată; 976K rămasi ai memoriei ar trebui să fie alocati în blocuri mai mici. [[1]](https://en.wikipedia.org/wiki/Buddy_memory_allocation)

## 2.2 Implementare si eficienta

Sistemul de memorie are puțină fragmentare externă, și permite compactarea memoriei cu puțin deasupra limitei. Metoda prietenului de a elibera memoria este rapidă, cu numărul maxim de tasări necesare egală log2 (cea mai mare comanda). De obicei sistemul „prieten” de alocare a memoriei este implementat cu ajutorul unui arbore binar pentru a reprezenta blocurile de memorie divizate utilizate sau neutilizate. „Prietenul" fiecarui bloc poate fi găsit exclusiv cu adresa blocului și dimensiunea blocului.

Cu toate acestea, există încă problema fragmentării interne - memoria pierdută pentru că memoria solicitată este un pic mai mare decât un bloc mic, dar mult mai mic decât un bloc mare. Din cauza modului în care tehnica de alocare de memorie prietene funcționează, un program care cere 66K de memorie va aloca 128K, ceea ce duce la o pierdere de 62K de memorie. Această problemă poate fi rezolvată prin alocarea lespede, care poate fi stratificata pe partea de sus a repartitorului amic mai aspru pentru a oferi alocare cu granulație mai fină.

O versiune a algoritmului de alocare prieten a fost descrisă în detaliu de Donald Knuth în volum 1 de Arta Computer Programming. Kernel-ul Linux folosește, de asemenea, sistemul „prieten”, cu modificarile ulterioare, pentru a minimiza fragmentarea externa, împreună cu diverse alte repartitoare pentru a gestiona memoria în blocuri. [[1]](https://en.wikipedia.org/wiki/Buddy_memory_allocation)

# 2.3 Alocarea memoriei sub forma de stiva

Stivele în arhitecturile de calcul sunt regiuni de memorie în care se adaugă date sau sunt eliminate în modul „ultimul venit, primul servit” (LIFO).

În cele mai multe sisteme informatice moderne, fiecare fir are o zonă rezervată de memorie denumite stiva sa. Când o funcție se executa, se pot adăuga unele dintre datele sale de stat la partea de sus a stivei; când funcția iese este responsabila pentru eliminarea de date din stivă. La un nivel minim, stivă unui fir este utilizata pentru a stoca locația de apeluri de funcții pentru a permite declarațiile RETURN pentru a reveni la locația corectă, dar programatorii pot alege în continuare să utilizeze în mod explicit stiva. Dacă o regiune de memorie se află pe stiva firului, memoria se spune că a fost alocata pe stiva.  
 Deoarece se adaugă și îndepărtează datele în modul ultimul-intrat-primul-iesit, alocarea de memorie pe baza de stivă este foarte simplă și este mai rapidă decât în ​​modul obișnuit al alocării de memorie heap-based (de asemenea, cunoscut sub numele de alocare dinamică a memoriei). O altă caracteristică este că memoria pe stiva este automată, și foarte eficientă, recuperată atunci când ieșirile sunt funcționale, care pot fi convenabile pentru programator, dacă nu mai sunt necesare datele. Dacă totuși, datele trebuie să fie păstrate într-o formă, atunci acestea trebuie să fie copiate de pe stivă înainte de ieșirile de funcții. Prin urmare, alocarea pe baza de stiva este potrivita pentru datele temporare sau date care nu mai sunt necesare după ce functia de incheie.

Alocarea de mai multă memorie pe stiva decat este disponibilă poate duce la un accident din cauza overflow-ului.  
 Unele familii de procesoare, cum ar fi x86, au instrucțiuni speciale pentru manipularea stivei firului de executare în prezent. Alte familii de procesoare, inclusiv PowerPC și MIPS, nu au suport stack explicit, ci se bazează pe convenții și managementul stivei la interfața binara, aplicat sistemului de operare (ABI). [[1]](https://en.wikipedia.org/wiki/Buddy_memory_allocation)

# 2.4 Garbage collection

Colectorul de resturi de memorie sau pur și simplu colector, încearcă să revendice memorii ocupate de obiecte care nu mai sunt utilizate de program. Colectorul a fost inventat de John McCarthy în jurul anului 1959 pentru a rezolva problemele în Lisp.

Algoritmul de colectare a resturilor este adesea descris ca opusul de management al memoriei, care necesită ca programatorul sa specifice obiectele de dealocare și de revenire la sistemul de memorie. Cu toate acestea, multe sisteme folosesc o combinație de abordări, inclusiv alte tehnici, cum ar fi alocarea stack. Ca și alte tehnici de management de memorie, colectorul poate lua o proporție semnificativă din timpul total de prelucrare într-un program și poate avea, astfel, o influență semnificativă asupra performanței.

Altele decât memoria de resurse, cum ar fi prize de rețea, baze de date, ferestre de interacțiune catre utilizator, și fișiere și dispozitive descriptorii, nu sunt, de obicei, gestionate de colector. Metodele utilizate pentru a gestiona astfel de resurse, în special destructorii, pot fi suficiente pentru a gestiona memoria, și nu este nevoie de GC. Unele sisteme GC permit acestor alte resurse de a fi asociate cu o regiune de memorie care, atunci când a colectat, provoacă alte resurse să fie regenerate; aceasta se numește finalizare. [[2]](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))

## 2.4.1 Principiile „garbage collection”

Principiile de bază ale colectarii resturilor de memorie sunt de a găsi obiecte de date dintr-un program care nu pot fi accesate în viitor, precum și pentru a recupera resursele folosite de aceste obiecte.  
  
 Multe limbaje de programare necesită colectarea resturilor, fie ca parte a caietului de sarcini a limbajului (de exemplu, Java, C #, limbajul D și cele mai multe limbaje de scripting) sau efectiv pentru punerea în aplicare (de exemplu, calculul lambda). Alte limbaje au fost concepute pentru a fi utilizate cu managementul memoriei manual, dar au implementari de colectoare disponibile (de exemplu, C și C ++). Unele limbaje, cum ar fi Ada, Modula-3, și C ++ / CLI, permit atât colectarea resturilor de memorie cat si un management al memoriei manual pentru a co-exista în aceeași cerere, cu ajutorul grămezii separate pentru obiectele colectate și gestionate manual; alții, cum ar fi D, dispun de colectoare de resturi de memorie, dar permit utilizatorului să ștearga manual obiecte și de asemenea si dezactivarea colectoarelor atunci cand este necesara viteza.   
  
 Pe langă integrarea de colectare a resturilor de memorie în compilator și execuția limbajului, sistemul permite o gamă mult mai largă de metode; există sisteme GC post-hoc, cum ar fi ARC, inclusiv unele care nu necesită recompilarea. (Post-hoc GC este uneori distins ca colector) Colectorul de resturi de memorie va fi aproape întotdeauna strâns integrat cu alocarea de memorie. [[2]](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))

### Avantaje

Colectarea de resturi de memorie eliberează programatorul de a o face manual, cu dealocare de memorie. Ca urmare, anumite categorii de bug-uri sunt eliminate sau reduse substanțial:  
  
 -Bug-uri pointer marionete, care apar atunci când o bucată de memorie este eliberata în timp ce există încă indicii la ea, iar unul dintre acesti indici este dereferentierea. Până atunci memoria poate fi mutata la o altă utilizare, cu rezultate imprevizibile.  
 -Bug-uri libere duble, care apar atunci când programul încearcă să elibereze o regiune de memorie care a fost deja eliberata, și, probabil, a fost deja alocată din nou.  
 -Anumite tipuri de pierderi de memorie, prin care un program nu reușește sa memoreze obiectele ocupate care au devenit de neatins, fapt care poate duce la epuizarea memoriei eliberate. (Colectarea de obicei nu se ocupă cu acumularea nelimitata de date, care este accesibilă, dar care nu va fi de fapt utilizata de program).Unele dintre bug-urile vizate de colectarea resturilor de memorie pot avea implicații de securitate. [[2]](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))

* + 1. **Dezavantaje**

În mod obișnuit, colectarea de resturi de memorie are anumite dezavantaje, inclusiv consumarea de resurse suplimentare, efecte de performanță, posibilele probleme în executarea programului, și incompatibilitatea cu managementul resurselor manual.

Pentru colectare se consumă resurse de calcul pentru a decide care memoria e libera, chiar dacă programatorul poate sa cunoasca deja aceste informații. In urma unui document revizuit s-a ajuns la concluzia că GC are nevoie de cinci ori mai multă memorie pentru a compensa și pentru a efectua cât mai repede gestionarea memoriei explicite. Interacțiunea cu efectele ierarhice de memorie poate face acest lucru intolerabil în circumstanțele care sunt greu de prezis sau detectat în testarea de rutina. Impactul asupra performanței a fost dat de Apple ca un motiv pentru a nu adopta colectarea resturilor în iOS, în ciuda faptului ca este caracteristica cea mai dorită.

Momentele în care resturile de memorie sunt de fapt colectate pot fi imprevizibile, rezultând în boxe împrăștiate pe parcursul unei sesiuni. Boxele imprevizibile pot fi inacceptabile în medii în timp real, în procesarea tranzacțiilor, sau în programe interactive. Colectorii de resturi elementari, concurenti, și în timp real ar aborda aceste probleme, cu diferite compromisuri.  
  
GC non-deterministic este incompatibil cu managementul RAII bazat pe resursele non-Gced. Ca urmare, nevoia managementului manual de resurse (eliberare / inchidere) pentru resursele non-GCed devine tranzitivă. Aceasta este: într-un sistem GC non-deterministic, în cazul în care o resursă sau un obiect asemănător de resurse necesită un management manual al resurselor (eliberare / închidere), iar acest obiect este utilizat ca "o parte din" alt obiect, atunci obiectul compus va deveni, de asemenea, un obiect resursă, care presupune managementul manual al resurselor (eliberare / close). [[2]](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))

BIBLIOGRAFIE

[1] <https://en.wikipedia.org/wiki/Buddy_memory_allocation>

[2]<https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)>

**3. Lucru cu memoria - Petrescu Maria**

**3.1 Algoritmi de inlocuire a paginilor**

La aparitia unei erori de paginare, sistemul de operare este determinat sa determine pagina care va fi eliminata din memorie pentru a face loc unei noi pagini.

Pot fi abordate diferite politici: alegerea aleatoare; alegerea paginilor cu utilizare scazuta; alegerea paginilor intr-o ordine prestabilita, etc.

**3.1.1Algoritmul NRU (Nefolosit recent)**

Acest algoritm se bazeaza pe doi biti: R (referinta catre pagina) si M (pagina modificata). Cei doi biti sunt actualizati la orice referire catre memorie.

La inceperea unui proces, toate intrarile sunt marcate ca nefiind in memorie: primul acces va genera o eroare de paginare. Ca urmare, bitul R este setat si intrarea din tabela de pagini modificata pentru a referi pagina corecta. Cand pagina este modificata este generata o eroare de paginare pentru a permite sistemului de operare sa seteze bitul M. [[1]](http://stst.elia.pub.ro/news/SO_2008/SO_09_memem/Implementarea%20gestiunii%20memoriei(2).pdf)

Algoritmul este urmatorul:

-La inceperea unui proces pentru fiecare pagina bitii R si M sunt resetati.

-Periodic bitul R este resetat pentru a face diferenta intre pagini recent referite si pagini nereferite.

-La o eroare de paginare, toate paginile sunt incadrate in categoriile:

0: pagini nereferite si nemodificate

1: pagini nereferite dar modificate

2: pagini referite dar nemodificate

3: pagini referite si modificate

Sistemul de operare alege aleator o pagina din clasa cea mai mica.

**3.1.2 Algoritmul FIFO (Primul venit, primul iesit)**

Ideea algoritmului este simpla: este aleasa pagina cu cea mai mare vechime in sistem. Pentru aceasta sistemul de operare va mentine o lista a tuturor paginilor din sistem, prima pagina fiind cea mai veche.

La aparitia unei erori de paginare este aleasa prima pagina din lista.[[1]](http://stst.elia.pub.ro/news/SO_2008/SO_09_memem/Implementarea%20gestiunii%20memoriei(2).pdf)

**3.1.3 Algoritmul LRU (Cel mai puțin recent folosit)**

Acest algoritm se bazeaza pe ideea ca paginile intens utilizate sa aiba o utilizare intensa si in viitor. La aparitia unei erori de paginare va fi eliminata una dintre paginile neutilizate in ultima perioada de timp.

Algoritmul obtinut ar putea fi insa costisitor: paginile aflate in memorie sunt pastrate intr-o lista care va fi modificata permanent in functie de utilizarea paginilor.

Acest algoritm ar putea fi utilizat prin utilizarea unui dispozitiv continand nxn biti, initial nesetati. La o referire catre paina cadru k, bitii de pe linia k vor fi setati iar cei de pe coloana k resetati. Una dintre liniile cu cea mai mica valoare binara va fi aleasa in aceasta situatie.

[[5]](http://web.info.uvt.ro/~fortis/LICENTA/SO/Lectures/SistemeOperare2009_Memorie_1.pdf)

Algoritmul LRU, folosind o matrice, cand paginile sunt referite in ordinea 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

[[5]](http://web.info.uvt.ro/~fortis/LICENTA/SO/Lectures/SistemeOperare2009_Memorie_1.pdf)

Algoritmul de imbatranire simuleaza LRU.Sunt reprezentate 6 pagini, pentru 5 batai de ceas((a) --(e)).

**3.1.4 Algoritmul optim**

Acest algoritm se bazeaza pe urmatoarea idee: la momentul unei erori de paginare o singura pagina va fi referita pentru urmatoarea instructiune.

Ideea este de a eticheta paginile cu numerele instructiunilor care urmeaza sa fie executate, in ordine: pagina eliminata va fi “cea mai departata” (cu eticheta cea mai mare)

Algoritmul este dificil de utilizat datorita faptului ca sistemul de operare nu este capabil sa prezica momentele la care va folosi diferitele pagini de memorie, insa simularea lui poate fi utila pentru a modela functionarea algoritmilor “reali”. [[1]](http://stst.elia.pub.ro/news/SO_2008/SO_09_memem/Implementarea%20gestiunii%20memoriei(2).pdf)

**3.1.5 Algoritmul Ceasului**

Algoritmul precedent presupune o activitate intensa de gestiune a listei de pagini.

Algoritmul ceasului presupune utilizarea unei liste circulare, cu un indicator pentru cea mai veche pagina.

De data aceasta, o eroare de paginare este urmata de inlocuirea paginii indicate daca bitul R este nesetat sau inaintarea indicatorului si resetarea bitului R.

[[5]](http://web.info.uvt.ro/~fortis/LICENTA/SO/Lectures/SistemeOperare2009_Memorie_1.pdf)

**3.2 Probleme in lucrul cu memoria**

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.

Ambele probleme și utilitarele care pot fi folosite pentru combaterea acestora vor fi prezentate în continuare. [[2]](http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-04)

**3.2.1 Tratare acces invalid**

De obicei, accesarea unei zone de memorie nevalide rezultă într-o eroare de pagină (page fault) și terminarea procesului (în Unix înseamnă trimiterea semnalului SIGSEGV → afișarea mesajului 'Segmentation fault'). Totuși, 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 exista situații în care să fie folosită doar jumătate din pagină. Deși 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. După cum se va vedea în continuare, există utilitare care ajută la detectarea acestor situații.

Un tip special de acces nevalid este [buffer overflow](http://en.wikipedia.org/wiki/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 [[2]](http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-04)

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

1

#include <stdio.h>

#include <stdlib.h>

static int fibonacci(int no)

{

if (1 == no || 2 == no)

return 1;

return fibonacci(no-1) + fibonacci(no-2);

}

int main(void)

{

short int numar, baza=10;

char sir[1];

scanf("%s", sir);

numar=strtol(sir, NULL, baza);

printf("fibonacci(%d)=%d\n", numar, fibonacci(numar));

return 0;

}

Pe exemplul de mai sus, vom demonstra utilitatea comenzii backtrace:

so@spook$ gcc -Wall exemplul-7.c -g

so@spook$ gdb a.out

(gdb) break 8

Breakpoint 1 at 0x8048482: file exemplul-7.c, line 8.

(gdb) run

Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out

7

Breakpoint 1, fibonacci (no=2) at exemplul-7.c:8

8 return 1;

(gdb) bt

#0 fibonacci (no=2) at exemplul-7.c:8

#1 0x0804849d in fibonacci (no=3) at exemplul-7.c:9

#2 0x0804849d in fibonacci (no=4) at exemplul-7.c:9

#3 0x0804849d in fibonacci (no=5) at exemplul-7.c:9

#4 0x0804849d in fibonacci (no=6) at exemplul-7.c:9

#5 0x0804849d in fibonacci (no=7) at exemplul-7.c:9

#6 0x0804851c in main () at exemplul-7.c:20

#7 0x4003d280 in \_\_libc\_start\_main () from /lib/libc.so.6

(gdb)

Se observă că la afișarea apelurilor de funcții se listează și parametrii cu care a fost apelată funcția. Acest lucru este posibil datorită faptului că atât variabilele locale, cât și parametrii acesteia sunt păstrați pe stivă până la ieșirea din funcție.

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. După cum s-a observat, exemplul anterior are un bug ce se manifestă atunci când numărul introdus de la tastatură depășește dimensiunea buffer-ului alocat (static). Acest tip de eroare poartă denumirea de *buffer overflow* și este extrem de gravă. Cele mai multe atacuri de la distanță pe un sistem sunt cauzate de acest tip de erori. Din păcate, acest tip de eroare nu este ușor de detectat, pentru că în procesul de buffer overrun se pot suprascrie alte variabile, ceea ce duce la detectarea erorii nu imediat când s-a făcut suprascrierea, ci mai târziu, când se va folosi variabila afectatã.

so@spook$ gdb a.out

(gdb) run

Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out

10

Program received signal SIGSEGV, Segmentation fault.

0x08048497 in fibonacci (no=-299522) at exemplul-7.c:9

9 return fibonacci(no-1) + fibonacci(no-2);

(gdb) bt -5

#299520 0x0804849d in fibonacci (no=-2) at exemplul-7.c:9

#299521 0x0804849d in fibonacci (no=-1) at exemplul-7.c:9

#299522 0x0804849d in fibonacci (no=0) at exemplul-7.c:9

#299523 0x0804851c in main () at exemplul-7.c:20

#299524 0x4003e280 in \_\_libc\_start\_main () from /lib/libc.so.6

Din analiza de mai sus se observă că funcția fibonacci a fost apelată cu valoarea 0. Cum funcția nu testează ca parametrul să fie valid, se va apela recursiv de un număr suficient de ori pentru a cauza umplerea stivei programului. Se pune problema cum s-a apelat funcția cu valoarea 0, când trebuia apelată cu valoarea 10.

so@spook$ gdb a.out

(gdb) run

Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out

10

Program received signal SIGSEGV, Segmentation fault.

0x08048497 in fibonacci (no=-299515) at exemplul-7.c:9

9 return fibonacci(no-1) + fibonacci(no-2);

(gdb) bt -2

#299516 0x0804851c in main () at exemplul-7.c:20

#299517 0x4003d280 in \_\_libc\_start\_main () from /lib/libc.so.6

(gdb) fr 299516

#299516 0x0804851c in main () at exemplul-7.c:20

20 printf("fibonacci(%d)=%d\n", numar, fibonacci(numar));

(gdb) print numar

$1 = 0

(gdb) print baza

$2 = 48

(gdb)

Se observă că problema este cauzată de faptul că variabila baza a fost alterată. Pentru a determina când s-a întâmplat acest lucru, se poate folosi comanda watch. Această comandă primește ca parametru o expresie și va opri execuția programului de fiecare dată când valoarea expresiei se schimbă.

(gdb) quit

so@spook$ gdb a.out

(gdb) break main

Breakpoint 1 at 0x80484d6: file exemplul-7.c, line 15.

(gdb) run

Starting program: /home/tavi/cursuri/so/lab/draft/intro/a.out

Breakpoint 1, main () at exemplul-7.c:15

15 short int numar, baza=10;

(gdb) n

18 scanf("%s", sir);

(gdb) watch baza

Hardware watchpoint 2: baza

(gdb) continue

Continuing.

10

Hardware watchpoint 2: baza

Old value = 10

New value = 48

0x40086b41 in \_IO\_vfscanf () from /lib/libc.so.6

(gdb) bt

#0 0x40086b41 in \_IO\_vfscanf () from /lib/libc.so.6

#1 0x40087259 in scanf () from /lib/libc.so.6

#2 0x080484ed in main () at exemplul-7.c:18

#3 0x4003d280 in \_\_libc\_start\_main () from /lib/libc.so.6

(gdb)

Din analiza de mai sus se observă că valoarea variabilei este modificată în funcția \_IO\_vfscanf, care la rândul ei este apelată de către functia scanf. Dacă se analizează apoi parametrii pasați functiei scanf, se observă imediat cauza erorii.

Pentru mai multe informații despre GDB consultați [documentația online](http://sourceware.org/gdb/download/onlinedocs/) (alternativ pagina info - info gdb) sau folosiți comanda help din cadrul GDB. [[2]](http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-04)

mcheck - verificarea consistenței heap-ului

glibc permite verificarea consistenței heap-ului prin intermediul apelului [mcheck](http://www.gnu.org/software/libc/manual/html_node/Heap-Consistency-Checking.html#Heap-Consistency-Checking) definit în mcheck.h. Apelul [mcheck](http://www.gnu.org/software/libc/manual/html_node/Heap-Consistency-Checking.html#Heap-Consistency-Checking) 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.

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).

În continuare, este prezentat un exemplu de cod cu probleme în alocarea și folosirea heap-ului:

mcheck\_test.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main(void)

{

int \*v1;

v1 = malloc(5 \* sizeof(\*v1));

if (NULL == v1) {

perror("malloc");

exit (EXIT\_FAILURE);

}

/\* overflow \*/

v1[6] = 100;

free(v1);

/\* write after free \*/

v1[6] = 100;

/\* reallocate v1 \*/

v1 = malloc(10 \* sizeof(int));

if (NULL == v1) {

perror("malloc");

exit (EXIT\_FAILURE);

}

return 0;

}

Mai jos se poate vedea cum programul este compilat și rulat. Mai întâi este rulat fără opțiuni de mcheck, după care se definește variabila de mediu MALLOC\_CHECK\_ la rularea programului. Se observă că deși se depășește spațiul alocat pentru vectorul v1 și se referă vectorul **după** eliberarea spațiului, o rulare simplă nu rezultă în afișarea nici unei erori.

Totuși, dacă definim variabila de mediu MALLOC\_CHECK\_, se detectează cele două erori. De observat că o eroare este detectată doar în momentul unui nou apel de memorie interceptat de mcheck.

so@spook$ make

cc -Wall -g mcheck\_test.c -o mcheck\_test

so@spook$ ./mcheck\_test

so@spook$ MALLOC\_CHECK\_=1 ./mcheck\_test

malloc: using debugging hooks

\*\*\* glibc detected \*\*\* ./mcheck\_test: free(): nevalid pointer: 0x0000000000601010 \*\*\*

\*\*\* glibc detected \*\*\* ./mcheck\_test: malloc: top chunk is corrupt: 0x0000000000601020 \*\*\*

mcheck nu este o soluție completă și nu detectează toate erorile ce pot apărea în lucrul cu memoria. Detectează, totuși, un număr important de erori și reprezintă o facilitate importantă a glibc.

**3.2.2 Tratare pierderi de spatii de memorie**

Un [leak de memorie](http://en.wikipedia.org/wiki/Memory_leak) apare în două situații:

* un program omite să elibereze o zonă de memorie
* un program pierde referința la o zonă de memorie alocată și, 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.[[3]](https://msdn.microsoft.com/en-us/library/windows/desktop/dd744766%28v=vs.85%29.aspx)

Ca și în cazul problemei accesului nevalid la memorie, [utilitarul Valgrind](http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-04#valgrind) este foarte util în detectarea leak-urilor de memorie ale unui program.

**3.2.2.1Valgrind**

Valgrind reprezintă o suită de utilitare folosite pentru operații de debugging și profiling. Cel mai popular este [Memcheck](http://valgrind.org/docs/manual/mc-manual.html), 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 sauHelgrind, util pentru depanarea programelor multithreaded. [[4]](http://valgrind.org/docs/manual/mc-manual.html)

În continuare, ne vom referi doar la utilitarul [Memcheck](http://valgrind.org/docs/manual/mc-manual.html) de detectare a erorilor de lucru cu memoria. Mai precis, acest utilitar detectează următoarele tipuri de erori:

* folosirea de memorie neinițializată
* citirea/scrierea din/în memorie după ce regiunea respectivă a fost eliberată
* citirea/scrierea dincolo de sfârșitul zonei alocate
* citirea/scrierea pe stivă în zone necorespunzătoare
* memory leak-uri
* folosirea necorespunzătore de apeluri malloc/new și free/delete

Valgrind nu necesită adaptarea codului unui program, ci folosește direct executabilul (binarul) asociat unui program. La o rulare obișnuită Valgrind va primi argumentul –tool pentru a preciza utilitarul folosit și programul care va fi verificat de erori de lucru cu memoria.

S-a folosit utilitarul [Memcheck](http://valgrind.org/docs/manual/mc-manual.html) pentru obținerea informațiilor de acces la memorie. [[2]](http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-04)

Exemplul următor reprezintă un program cu o gamă variată de erori de alocare a memoriei:

1

#include <stdlib.h>

#include <string.h>

int main(void)

{

char buf[10];

char \*p;

/\* no init \*/

strcat(buf, "al");

/\* overflow \*/

buf[11] = 'a';

p = malloc(70);

p[10] = 5;

free(p);

/\* write after free \*/

p[1] = 'a';

p = malloc(10);

/\* memory leak \*/

p = malloc(10);

/\* underrun \*/

p--;

\*p = 'a';

return 0;

}

La o rulare obișnuită, programul nu generează nici un fel de eroare. Totuși, la rularea cu Valgrind, apar erori.

În plus, există leak-uri de memorie datorită noului apel malloc care asociază o nouă valoare lui p.

Valgrind este un utilitar de bază în depanarea programelor. Este facil de folosit (nu este intrusiv, nu necesită modificarea surselor) și permite detectarea unui număr important de erori de programare apărute ca urmare a gestiunii defectuoase a memoriei. [[4]](http://valgrind.org/docs/manual/mc-manual.html)

**3.2.2.2 Mtrace**

Un alt utilitar care poate fi folosit la depanarea erorilor de lucru cu memoria este [mtrace](http://en.wikipedia.org/wiki/Mtrace). Acest utilitar ajută la identificarea leak-urilor de memorie ale unui program. [[2]](http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-04)

Utilitarul [mtrace](http://www.gnu.org/software/libc/manual/html_node/Tracing-malloc.html#Tracing-malloc) sefolosește cu apelurile [mtrace](http://www.gnu.org/software/libc/manual/html_node/Tracing-malloc.html#Tracing-malloc) și [muntrace](http://www.gnu.org/software/libc/manual/html_node/Tracing-malloc.html#Tracing-malloc) implementate în biblioteca standard C:

void mtrace(void);

void muntrace(void);

Utilitarul [mtrace](http://www.gnu.org/software/libc/manual/html_node/Tracing-malloc.html#Tracing-malloc) introduce handlere pentru apelurile de biblioteca pentru lucrul cu memoria (malloc,realloc, free). Apelurile [mtrace](http://www.gnu.org/software/libc/manual/html_node/Tracing-malloc.html#Tracing-malloc) și [muntrace](http://www.gnu.org/software/libc/manual/html_node/Tracing-malloc.html#Tracing-malloc) activează, respectiv dezactivează monitorizarea apelurilor de bibliotecă de lucru cu memoria.

Jurnalizarea operațiilor efectuate se realizează în fișierul definit de variabila de mediu MALLOC\_TRACE. După ce apelurile au fost înregistrate în fișierul specificat, utilizatorul poate să folosească utilitarul mtrace pentru analiza acestora.

În exemplul de mai jos este prezentată o situație în care se alocă memorie fără a fi eliberată:

mtrace\_test.c

#include <stdlib.h>

#include <mcheck.h>

int main(void)

{

/\* start memcall monitoring \*/

mtrace();

malloc(10);

malloc(20);

malloc(30);

/\* stop memcall monitoring \*/

muntrace();

return 0

}

În secvența de comenzi ce urmează se compilează fișierul de mai sus, se stabilește fișierul de jurnalizare și se rulează comanda mtrace pentru a detecta problemele din codul de mai sus.

so@spook$ gcc -Wall -g mtrace\_test.c -o mtrace\_test

so@spook$ export MALLOC\_TRACE=./mtrace.log

so@spook$ ./mtrace\_test

so@spook$ cat mtrace.log

= Start

@ ./mtrace\_test:[0x40054b] + 0x601460 0xa

@ ./mtrace\_test:[0x400555] + 0x601480 0x14

@ ./mtrace\_test:[0x40055f] + 0x6014a0 0x1e

= End

so@spook$ mtrace mtrace\_test mtrace.log

Memory not freed:

-----------------

Address Size Caller

0x0000000000601460 0xa at /home/razvan/school/so/labs/lab4/samples/mtrace.c:11

0x0000000000601480 0x14 at /home/razvan/school/so/labs/lab4/samples/mtrace.c:12

0x00000000006014a0 0x1e at /home/razvan/school/so/labs/lab4/samples/mtrace.c:15

**3.2.3 Dublă dealocare**

Denumirea de “dublă dealocare” oferă o bună intuiție asupra cauzei: eliberarea de două ori a aceluiași 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. Fie exemplul de mai jos: [[2]](http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-04)

dubla\_dealocare.c

#include <stdlib.h>

int main(void)

{

char \*p;

p = malloc(10);

free(p);

free(p);

return 0;

}

Rularea executabilului obținut din programul de mai sus duce la afișarea unui mesaj specific al glibc de eliberare dublă a unei regiuni de memorie și terminare a programului:

Situațiile de dublă dealocare sunt, de asemenea, detectate de Valgrind.

BIBLIOGRAFIE

[1]<http://stst.elia.pub.ro/news/SO_2008/SO_09_memem/Implementarea%20gestiunii%20memoriei(2).pdf>

[2] <http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-04>

[3] <https://msdn.microsoft.com/en-us/library/windows/desktop/dd744766%28v=vs.85%29.aspx>

[4] <http://valgrind.org/docs/manual/mc-manual.html>

[5]<http://web.info.uvt.ro/~fortis/LICENTA/SO/Lectures/SistemeOperare2009_Memorie_1.pdf>

[6]<http://en.wikipedia.org/wiki/Page_replacement_algorithm>

**4 .Implementarea gestiunii memoriei la Windows - Avram Tiberiu**

**4.1. Memoria virtuala si memoria fizica**

**4.1.1 Memoria virtuala**

Dezvoltarea software a dus la aparitia unor aplicatii care depasesc memoria fizica disponibila.Chiar daca tehnicile bazate pe swapping sunt extrem de utile intr-un sistem cu multiprogramare, acestea nu pot oferi posibilitatea depasirii limitelor fizice ale memoriei.

Una dintre solutiile de inceput pentru depasirea limitarilor fizice impuse de memorie presupune impartirea programelor in mai multe bucati, overlays.

Executia unui program incepe, in aceasta situatie, cu prima bucata disponibila. Odata incheiata executia unei bucati, este incarcata de pe disc o alta bucata.Sistemul de operare permitea pastrarea in memorie a unui numar mare de bucati-program (overlays), astfel incat sarcinile de gestiune si design ale acestora sunt extrem de dificil de realizat.Sistemul de operare ofera suportul pentru aducerea si eliminarea din memorie, intr-o maniera dinamica, a segmentelor de program.Impartirea programelor in segmente este realizata insa de catre programator.Prin automatizarea sarcinilor de impartire a programelor in bucati (propusa initial de Fotheringham), au fost propuse primele tehnici de memorie virtuala.In aceasta situatie, sistemul de operare va determina singur daca o aplicatie urmeaza sa fie impartita in bucati si va determina modul in care se poate realiza aceasta impartire.

Intr-un sistem cu multiprogramare, un proces care asteapta ca un segment sa fie adus in memorie este considerat ca fiind in asteptarea unei operatii de intrare/iesire.

Cantitatea de memorie virtuala si fizica de care dispune fiecare calculator ce foloseste un sistem de operate Microsoft Windows este determinata de configuratia hardware si de editia platformei de Windows folosita. Pe un hardware de 32 de biti spatiul adresei virtuale este de 4 GB si catitatea maxima de memorie fizica este cuprinsa intre 2 si 128 GB. Pe un hardware de 64 de biti spatiul adresei virtuale este de 16 TB si catintatea maxima de memorie fizica este cuprinsa intre 64 GB si 1 TB.

Urmatorul tabel prezinta cantitatea de memorie virtuala si cantitatea maxima de memorie fizica pentru fiecare editie de Windows.[[2]](http://web.info.uvt.ro/~fortis/LICENTA/SO/Lectures/SistemeOperare2009_Memorie_1.pdf)



[[2]](http://web.info.uvt.ro/~fortis/LICENTA/SO/Lectures/SistemeOperare2009_Memorie_1.pdf)

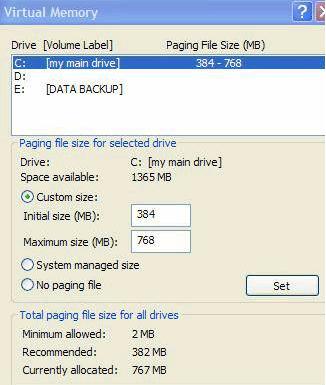
**4.1.2 Castigarea de spatiu pe HARD DISK prin modificarea memoriei virtuale**

Fisierul de paginare este rezervat de windows pentru a fi folosit ca memorie virtuala. Memoria virtuala este acesata de windows ca memorie fizica, aceasta fiind mai lenta decat memoria RAM. De fapt, Windows XP utilizeaza acest fisier de memorie virtuala in mod continuu, indiferent de cantitatea de memorie RAM disponibila. Asadar optimizarea acestui fisier are un impact asupra performantei calculatorului dumneavoastra.

Pentru a optimiza acest fisier exista cateva optiuni pe care ar trebui sa le luati in seama si anume: schimbarea locatiei fisierului de paginatie.

Deoarece fisierele scrise in acest fisier trebuiesc citite si scrise in mod continuu, punerea acestuia in aceeasi unitate cu sistemul de operare poate compromite performanta acestuia. Desigur daca sistemul dumneavoastra are o singura unitate de stocare acest fisier de paginatie nu poate fi mutat. Daca sistemul contine mai multe unitati de stocare luati in considerare mutarea “memoriei virtuale” pe o alta unitate decat cea pe care se afla instalata sistemul de operare.

[[2]](http://web.info.uvt.ro/~fortis/LICENTA/SO/Lectures/SistemeOperare2009_Memorie_1.pdf)



**4.2 Alocarea memoriei in Windows**

In Windows, un proces poate să-şi creeze mai multe obiecte Heap pe langă Heap-ul implicit al procesulului. Acest lucru este foarte util in momentul in care o aplicaţie alocă şi dealocă foarte multe zone de memorie cu cateva dimensiuni fixe.

Aplicaţia poate să-şi creeze cate un Heap pentru fiecare dimensiune şi, in cadrul fiecărui Heap, să aloce zone de aceeaşi dimensiune reducand astfel la maxim fragmentarea heap-ului.

Pentru crearea, respectiv distrugerea unui Heap se vor folosi funcţiile HeapCreate şi HeapDestroy:

HANDLE HeapCreate(

DWORD flOptions,

SIZE\_T dwInitialSize,

SIZE\_T dwMaximumSize

);

BOOL HeapDestroy(

HANDLE hHeap

);

Pentru a obţine un descriptor al heap-ului implicit al procesului (in cazul in care nu dorim crearea altor heapuri) se va apela funcţia GetProcessHeap. Pentru a obţine descriptorii tuturor heap-urilor procesului se va apela GetProcessHeaps.

Există, de asemenea, funcţii care enumeră toate blocurile alocate intr-un heap, validează unul sau toate blocurile alocate intr-un heap sau intorc dimensiunea unui bloc pe baza descriptorului de heap şi a adresei blocului: HeapWalk, HeapValidate, HeapSize.

Pentru alocarea, dealocarea, redimensionarea unui bloc de memorie din Heap, Windows pune la dispoziţia programatorului funcţiile HeapAlloc, HeapFree, respectiv HeapReAlloc, cu signaturile de mai jos:

LPVOID HeapAlloc(

HANDLE hHeap,

DWORD dwFlags,

SIZE\_T dwBytes

);

BOOL HeapFree(

HANDLE hHeap,

DWORD dwFlags,

LPVOID lpMem

);

LPVOID HeapReAlloc(

HANDLE hHeap,

DWORD dwFlags,

LPVOID lpMem,

SIZE\_T dwBytes

);

In continuare, este prezentat un exemplu de folosire al acestor funcţii:

#include <windows.h>

#include "utils.h"

/\* Example of matrix allocation \*/

int main(void)

{

HANDLE processHeap;

DWORD \*\*mat;

DWORD i, j, m = 10, n = 10;

processHeap = GetProcessHeap();

DIE (processHeap == NULL, "GetProcessHeap");

mat = HeapAlloc(processHeap, 0, m \* sizeof(\*mat));

DIE (mat == NULL, "HeapAlloc");

for (i = 0; i < n; i++) {

mat[i] = HeapAlloc(processHeap, 0, n \* sizeof(\*\*mat));

if (mat[i] == NULL) {

PrintLastError("HeapAlloc failed");

goto freeMem; /\* free previously allocated memory \*/

}

}

/\* do work \*/

freeMem:

for (j = 0; j < i; j++)

HeapFree(processHeap, 0, mat[j]);

HeapFree(processHeap, 0, mat);

return 0;

}

Pe sistemele Windows se pot folosi şi funcţiile bibliotecii standard C pentru gestiunea memoriei: *malloc, realloc, calloc, free,* dar apelurile de sistem specifice Windows oferă funcţionalităţi suplimentare şi nu implică legarea bibliotecii standard C in executabil.[[3]](http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-04)

**4.2.1 Nucleul: Alocarea memoriei**

Un alt alocator se află în nucleul sistemului de operare. Aşa cum am spus deja, acest alocator gestionează atît spaţiul folosit în structurile de date interne nucleului, cît şi spaţiu necesar proceselor care se execută. Interacţiunea dintre variatele alocatoare este înfăţişată în figura [2](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html#interactiune).

|  |
| --- |
| [\begin{figure}\begin{verbatim}Memoria RAM ^ \vert ___v_____________ \ver... ...uffere pentru retea, inoduri, tabela de procese, etc.)\end{verbatim}\end{figure}](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html) |
| [**[1]**](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html)  **Figure 2:** Interacţiunea dintre feluritele alocatoare de memorie din nucleu. Alocatorul de pagini gestionează întreaga memorie fizică a maşinii. El generează spaţiu atît pentru procese (unitatea de bază fiind pagina), cît şi memorie pentru alocatorul intern al nucleului. Săgeţile indică schimb de spaţiu de memorie între componente. |

Subiectul acestui articol este cutiuţa etichetată ``alocatorul intern al nucleului'', deşi multe din principiile indicate se aplică şi celorlalte entităţi. Aceste alocatoare sunt mult mai constrînse decît alocatoarele din spaţiul utilizatorului; în particular trebuie să aibă timpi de răspuns foarte mici, pentru că pot fi chemate de părţi critice din nucleu.

Cele trei alocatoare interacţionează permanent: cînd nucleul nu mai are destule zone pentru propriile lui date cere noi pagini de la alocatorul de pagini. Cînd procesele utilizatorilor mor, paginile lor sunt preluate de alocatorul de pagini. Cînd alocatorul de pagini are puţine resurse disponibile, el poate cere alocatorului intern să returneze din memoria pe care nu o foloseşte în acel moment, etc..

**4.2.2 Nucleul: alocarea de spaţiu pe disc**

Un ultim mare subsistem al nucleului care se ocupă de alocarea spaţiului este sistemul care alocă spaţiu pe disc, atît pentru fişiere, cît şi pentru memoria virtuală (memoria proceselor care sunt oprite din execuţie este uneori salvată pe disc pentru a permite executarea altora; această tehnologie se numeşte ``memorie virtuală'').

Alocatoarele de spaţiu pe disc au alte atribute foarte specifice; de pildă alocatoarele care trebuie să parcurgă o listă întreagă pentru a găsi spaţiul necesar sunt complet nepractice, pentru că accesele la disc durează extrem de mult (milisecunde, ceea ce înseamnă milioane de cicli de procesor). Despre funcţionarea acestor alocatoare am vorbit pe scurt în alte articole din PC Report, consacrate sistemelor de fişiere.

**4.2.3 Tipuri de alocatoare**

**4.2.3.1 Un alocator dinamic foarte simplu**

Vom ilustra aici în scop pur didactic implementarea unui alocator extrem de ineficient şi risipitor, dar totodată complet. Aceasta este schema de bază de funcţionare a tuturor alocatoarelor din nucleu; variaţiunile constau în îmbunătăţiri. Acest alocator implementează funcţia free() fără a face nimic! Memoria eliberată este pur şi simplu pierdută. Alocatorul extrage zonele de memorie dintr-o memorie mare alocată static iniţial, care are 16 megaocteţi. [[1]](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html)

/\* implementarea unui alocator dinamic foarte simplu; functia

free() nu face absolut nimic. \*/

typedef unsigned int size\_t;

#define MARIME\_MEMORIE 16 \* 1024 \* 1024

char memorie[MARIME\_MEMORIE];

int memorie\_consumata = 0;

void \* malloc(size\_t marime)

{

if (MARIME\_MEMORIE < marime + memorie\_consumata)

return 0;

memorie\_consumata += marime;

return (void\*) &memorie[memorie\_consumata - marime];

}

void free(void \*)

{}

Acest alocator este extrem de rapid; probabil că nici nu poate fi scris un alocator mai rapid. Preţul plătit este o enormă risipă de memorie: tot ce este dealocat se pierde complet.

Oricum, aceste funcţii ilustrează cărămizile de bază cu care un alocator din nucleu trebuie să opereze. Alocatorul de pagini manipulează memoria fizică a maşinii ca un vector enorm de octeţi, din care extrage porţiuni aşa cum facem noi cu vectorul numit memorie.

**4.2.3.2 Alocatorul cu hartă de resurse**

Acesta este probabil cel mai simplu tip de alocator care implementează şi funcţia free. Acest alocator menţine un vector de structuri care descriu fiecare bloc liber. Figura [3](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html#harta) arată o posibilă reprezentare a situaţiei la un moment dat:

|  |
| --- |
| \begin{figure}\centerline{\epsfxsize=8.5cm\epsffile{harta.eps}}\end{figure} |
| [[1]](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html)  **Figure 3:** Alocatorul cu hartă de resurse are o ``hartă'' care indică fiecare regiune liberă şi ocupată. |

Avem o mică problemă, pentru că alocatorul are nevoie el însuşi de spaţiu pentru a reprezenta tabela cu resurse. Putem face însă nişte mici trucuri pentru a compacta reprezentarea:

Cînd *malloc* găseşte un bloc, foloseşte primii 4 octeţi[5](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html" \l "foot175) pentru a memora lungimea sa şi returnează adresa următorului octet. Această lungime este folosită de free pentru a şti cît de mult trebuie dealocat.

Forma unui bloc de memorie ocupat: fiecare bloc este precedat de 4 octe'ti care con'tin lungimea blocului.

--------------------------------------------

| lungimea | |

--------------------------------------------

^

\ adresa returnat'a de malloc

Putem memora informaţia despre blocurile libere în chiar spaţiul ocupat de blocurile libere! Astfel, putem ţine blocurile libere într-o listă înlănţuită, ca în figura [4](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html#lista).

|  |
| --- |
| \begin{figure}\centerline{\epsfxsize=8cm\epsffile{lista.eps}}\end{figure} |
| [[1]](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html)  **Figure 4:** Putem implementa alocatorul cu hartă de memorie menţinînd în fiecare bloc liber informaţie despre lungimea proprie (L) şi adresa următorului bloc liber (N). |

Cînd alocatorul vrea să găsească un bloc liber pleacă de la adresa primului bloc liber şi parcurge lista de blocuri libere pînă găseşte unul de dimensiune corespunzătoare.

Acest alocator este relativ simplu, dar foarte ineficient. Din cauza fragmentării complexitatea operaţiilor este mare (după o vreme lista de blocuri libere devine populată cu o sumedenie de blocuri mici, a căror traversare este costisitoare şi inutilă). Schema este de asemenea uşor risipitoare: 4 octeţi sunt memoraţi în plus pentru fiecare bloc ocupat, iar faptul că un bloc liber trebuie să conţină două valori impune o lungime minimă de 8 octeţi pentru un bloc.

Eliberarea unui bloc îl inserează la începutul listei de blocuri libere. Dacă vrem să reducem fragmentarea, la eliberare putem să comasăm un bloc cu vecinul următor, în caz că acesta este liber şi el. Dacă vrem să putem comasa şi cu vecinul anterior, atunci în fiecare bloc memorăm şi în ultimii 4 octeţi lungimea sa; în acest fel, atunci cînd eliberăm un bloc, putem şti exact unde începe cel de dinainte. În orice caz, eliberarea cere un număr constant de instrucţiuni (care nu depinde de numărul de zone deja alocate).

[[1]](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html)

**4.2.3.3 Alocatorul cu puteri ale lui 2**

Problema alocatorului de mai sus constă în faptul că trebuie să caute un bloc de dimensiune potrivită. Pentru a remedia acest lucru putem folosi o altă reprezentare: păstrăm o serie de liste de blocuri, toate blocurile de pe fiecare listă avînd o anumită dimensiune. Atunci cînd vrem un anumit bloc, returnăm un bloc de dimensiunea imediat superioară. Cele mai folosite dimensiuni sunt puteri ale lui 2, ca în figura [5](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html#put2).

|  |
| --- |
| \begin{figure}\centerline{\epsfxsize=9.5cm\epsffile{put2.eps}}\end{figure} |
| [[1]](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html)  **Figure 5:** Structurile de date ale alocatorului cu puteri ale lui 2: cîte o listă de blocuri libere pentru fiecare dimensiune de bloc. Fiecare bloc ocupat are informaţii despre propria sa lungime. |

Acest alocator suferă de fragmentare internă, pentru că alocă întotdeauna mai mult decît i se cere.

Alocarea şi dealocarea cer timp constant pe operaţiune (calculul listei plus extragerea primului bloc din listă). Putem reprezenta listele de blocuri libere exact ca în schema cu harta de resurse; comasarea este practic imposibil a (am putea comasa numai vecini de dimensiuni egale, ca să obţinem rezultate de mărime tot puteri ale lui doi). Pentru toate blocurile mai mici de o anumită limită şi mai mari de o alta, putem avea nişte liste separate, în care facem căutare exhaustivă. Dacă vrem să avem liste cu toate dimensiunile posibile de blocuri sunt suficiente 32 de liste, ceea ce e o valoare rezonabilă.

În momentul în care o listă dorită este complet depopulată, putem scoate un bloc mai mare de pe lista imediat superioară, pe care îl putem sparge în două bucăţi mai mici.

În interiorul nucleului există multe structuri de date care au ca dimensiuni exact puteri ale lui 2; pentru astfel de cereri se iroseşte o cantitate enormă de memorie, pentru că fiecare bloc alocat trebuie să conţină şi cei 4 octeţi suplimentari. Astfel, dacă vrem să alocam 256 de octeţi, trebuie de fapt 260, care înseamnă ca folosim un bloc de 512; risipa este de aproape 100%!

Alte dezavantaje ale metodei: în general blocurile mici nu se pot grupa laolaltă pentru a forma blocuri mari, şi sistemul de paginare nu poate obţine pagini nefolosite înapoi în caz de nevoie. [[1]](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html)

Alocatorul Karels-McKusick

În 1988 Kirk McKusick şi Michael Karels au construit o variantă îmbunătăţită a alocatorului cu puteri ale lui 2, cere este folosită acum în 4.4BSD Unix şi Digital Unix. Metoda lor elimină risipa pentru cazul blocurilor care au dimensiuni exact puteri ale lui 2.

Listele de blocuri libere sunt înlănţuite exact ca în alocatorul cu puteri ale lui 2. Diferenţa principală constă în modul în care blocurile ocupate îşi reprezintă lungimea; trucul folosit este foarte ingenios: alocatorul menţine un vector mare de numere, cîte unul pentru fiecare pagină. Elementul 5 din vector reprezintă mărimea blocurilor din pagina 5. Figura [6](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html#karels) arată un exemplu de structuri de date ale alocatorului.

|  |
| --- |
| \begin{figure}\centerline{\epsfxsize=7cm\epsffile{karels.eps}}\end{figure} |
| [[1]](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html)  **Figure 6:** Vectorul care reprezintă mărimea blocurilor pentru fiecare pagină în alocatorul Karels-McKusick. Pagina 0 este divizată în blocuri de cîte 32 de octeţi, pagina 1 în blocuri de cîte 512, etc. Paginile încă nefolosite sunt înlănţuite într-o lista. |

O altă optimizare făcută de alocatorul acesta este modul în care se calculează rotunjirea în sus a unei valori la cea mai mică putere a lui 2; calculul este făcut cu un macro care conţine o expresie de genul:

#define LISTA(marime) \

(marime) > 128 \

? (marime) > 256 ? 4 : 3 \

: (marime) > 64 \

? 2 \

: (marime) > 32 ? 1 : 0

Avantajul unei astfel de expresii în locul unei bucle este că, atunci cînd mărimea este cunoscută la compilare, întreaga expresie devine o constantă care poate fi calculată de compilator, deci codul executat devine mult mai rapid. Acest alocator este mai eficient decît cel cu puteri ale lui 2, şi risipeşte mult mai puţină memorie, pentru că toate cererile egale cu o putere a lui doi sunt satisfăcute cu un buffer de mărime potrivită. Celelalte dezavantaje rămîn însă prezente.

**4.2.3.4 Alocatorul ``slab''**

O schemă mult mai sofisticată de alocare, inspirată din limbajele orientate pe obiecte a fost introdusă de Sun în sistemul de operare Solaris 2.4 şi este acum implementată şi în Linux. ``Slab'' înseamnă în engleză ``lespede''; acest alocator are zone de memorie diferite pentru obiecte diferite, pe care le putem vedea ca nişte mozaicuri de forme diferite; de aici şi numele. Acest alocator tratează cu multă atenţie o serie de aspecte importante pentru performanţă care sunt complet ignorate de alte alocatoare:

Alocatorul încearcă atunci cînd caută noi zone să nu atingă prea multe locaţii de memorie, pentru a nu polua cache-ul microprocesorului cu date care nu sunt utile; se spune că alocatorul are o ``urmă'' mică (small footprint);

Alocatorul încearcă să aloce obiectele în memorie în aşa fel încît două obiecte diferite să nu pice în aceeaşi linie din cache-ul de date; asta maximizează utilizarea cache-ului cînd obiectele sunt folosite simultan. De exemplu obiectul care reprezintă în nucleu un fişier are cîteva cîmpuri care sunt foarte des folosite, aflate să zicem la începutul structurii, şi alte cîteva cîmpuri care sunt extrem de rar folosite. Să presupunem că obiectul ``fişier'' are 256 de octeţi; în acest caz toate cîmpurile des folosite din toate fişierele se vor afla la adrese multiplu de 256. Din cauza asta ele se vor lupta pentru un set restrîns de linii din cache, în vreme ce celelalte linii vor fi prost utilizate. Alocatorul slab va încerca să aloce fiecare obiect la o adresă care nu e multiplu de aceeaşi valoare.

Alocatorul încearcă să reducă numărul de operaţii de iniţializare asupra noilor obiecte alocate. De exemplu multe obiecte din nucleu care pot fi folosite de mai multe procese au un cîmp care indică numărul de utilizatori (de pildă un fişier memorează numărul de procese care au deschis fişierul). Cînd acest număr ajunge 0, înseamnă că obiectul nu mai este utilizat, şi el poate fi dealocat. Asta înseamnă că avem certitudinea că orice fişier dealocat va avea valoarea 0 în acel cîmp. Atunci cînd alocăm un nou fişier, dacă folosim vechea zonă de memorie, ştim deci că nu mai trebuie să iniţializăm acest cîmp, pentru că are deja valoarea corectă! Practic nucleul menţine un cache cu obiecte de curînd dealocate, pe care le foloseşte atunci cînd i se cer noi alocări.

Alocatorul slab constă de fapt într-o rutină centrală (numită kmem\_cache\_create) care crează alocatoare pentru fiecare obiect. Rutina asta primeşte ca parametri numele obiectului, mărimea unui obiect, constrîngerile de aliniere (de ex. toate obiectele trebuie să fie la o adresă multiplu de 16) şi pointeri spre o funcţie constructor şi o funcţie destructor.

Iată un exemplu de utilizare:

alocator\_de\_fisiere = kmem\_cache\_create("fisier", sizeof(struct fisier),

8, constructor\_fisier,

destructor\_fisier);

alocator\_de\_inoduri = kmem\_cache\_create("inod", sizeof(struct inod),

4, constructor\_inod,

destructor\_inod);

struct fisier \* fs = kmem\_cache\_alloc(alocator\_de\_fisiere, FLAGS);

struct inod \* in = kmem\_cache\_alloc(alocator\_de\_inoduri, FLAGS);

Funcţia *kmem\_cache\_create* crează deci alocatoare pentru felurite obiecte. Fiecare alocator este apoi folosit pentru a construi obiecte de acel tip. Fiecare alocator are propria lui zonă de memorie, în care menţine numai obiecte de acel tip; va exista astfel o zonă de pagini în care se alocă numai fişiere, o zonă în care se alocă numai inoduri, etc. În acest fel toate obiectele dintr-o zonă au aceeaşi dimensiune!

Fiecare alocator menţine de asemenea o listă cu obiectele care au fost de curînd dealocate, şi le refoloseşte atunci cînd i se cer noi obiecte. Pentru că obiectele au fost dealocate, ele nu mai trebuie sa fie iniţializate din nou.

Atunci cînd un alocator epuizează toată memoria pe care o are la dispoziţie, el cere de la alocatorul de pagini o nouă pagină pe care o umple cu obiecte noi. Pentru că aceste obiecte nu au fost niciodată iniţializate, alocatorul cheamă funcţia constructor care a fost indicată pentru fiecare nou obiect. Observaţi că acest constructor este chemat numai prima oară cînd pagina este obţinută; după ce un obiect a fost dealocat constructorul nu mai este chemat din nou.

|  |
| --- |
| \begin{figure}\centerline{\epsfxsize=11cm\epsffile{slab.eps}}\end{figure} |
| [[1]](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html)  **Figure 7:** Alocatorul ``slab''. Fiecare obiect are propriul lui alocator, care menţine în pagini obiectele alocate şi libere. În fiecare pagină primul obiect alocat începe la altă adresă, pentru a încărca uniform liniile din cache-ul microprocesorului. Fiecare pagină mai posedă anumite structuri de date folosite la întreţinere (cum ar fi o listă dublu înlănţuită a tuturor paginilor pentru un anumit obiect). |

Fiecare alocator foloseşte propriile lui pagini într-un mod similar cu alocatorul cu puteri ale lui doi. La sfîrşitul fiecărei pagini alocatorul rezervă o zonă pentru o structură de date care descrie ocupaţia acelei pagini. Primul obiect în pagină este plasat la o distanţă aleatoare de marginea pagini; acest plasament are efectul de a pune obiecte din pagini diferite la adrese diferite în cache, exact cum am indicat mai sus.

Alocatorul slab poate returna sistemului de paginare paginile în care toate obiectele sunt nefolosite. Alocatorul are o ``urmă'' mică, pentru că majoritatea cererilor accesează o singură pagină. Acest tip de alocator risipeşte ceva resurse datorită modului de plasare în pagină şi pentru că are o zonă diferită pentru fiecare tip de obiect. Performanţa lui este însă excelentă, şi rămîne unul din cele mai puternice alocatoare implementate. [[1]](http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html)

BIBLIOGRAFIE

[1]<http://www.cs.cmu.edu/~mihaib/articles/mem/mem-html.html>

[2]<http://web.info.uvt.ro/~fortis/LICENTA/SO/Lectures/SistemeOperare2009_Memorie_1.pdf>

[3]<http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-04>

[4]<http://stst.elia.pub.ro/news/SO_2008/SO_09_memem/Implementarea%20gestiunii%20memoriei(2).pdf>

[5]<http://en.wikipedia.org/wiki/Virtual_memory>