UNIVERSITATEA POLITEHNICĂ BUCUREȘTI
Facultatea de Electronică,Telecomunicații și Tehnologia Informației

SISTEME DE OPERARE

Gestiunea memoriei

Prof. Ștefan Stăncescu

-2012-

Ghiga Alexandru,431A
Nica Mădălin,432A
Radu Ana-Maria,432A
Zamfir Oana-Liliana,431A

CUPRINS

1.INTRODUCERE

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

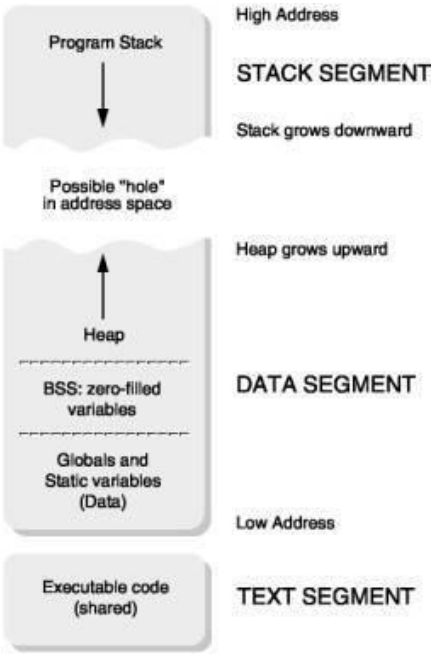
Rolul subsistemului de gestiune a memoriei este de a evidenția zonele de memorie fizică ocupate sau libere, de a oferi proceselor sau celorlalte subsisteme acces la memorie și de a mapa paginile de memorie virtuală ale unui proces (pages) peste paginile fizice (frames).

Nucleul sistemului de operare ofera un set de interfețe (apeluri de sistem) care permit alocarea/dezalocarea de memorie, maparea unor regiuni de memorie virtuală peste fișiere, partajarea zonelor de memorie.Din păcate, nivelul limitat de înțelegere a acestor interfețe și a acțiunilor ce se petrec în spate conduc la o serie de probleme foarte des întâlnite în aplicaițile software: memory leak-uri, accese invalide, suprascrieri, buffer overflow, corupere de zone de memorie.

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

2.CONCEPTE FUNDAMENTALE

A.Spațiul de adresă al unui proces



Spațiul de adrese al unui proces, spațiul virtual de adresă al unui proces reprezintă zona de memorie virtuală utilizabilă de un proces. Fiecare proces are un spațiu de adresă propriu. Chiar în situațiile în care doua procese partajează o zona de memorie, spațiul virtual este distinct, dar se mapează peste aceeași zonă de memorie fizică.

În figura alaturată este prezentat un spațiu de adresă tipic pentru un proces. În sistemele de operare moderne, în spațiul virtual al fiecarui proces se mapează memoria nucleului.Aceasta poate fi mapată fie la începutul,fie la sfârșitul spațiului de adresă.

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

B. Segmentul de cod

Segmentul de cod (denumit și 'text segment') reprezintă instrucțiunile programului. Registrul de tip 'instruction pointer' va referi adrese din zona de cod. Se citește instrucțiunea indicată, se decodifică și se interpretează, dupa care se incrementează contorul programului și se trece la următoarea instrucțiune.Segemntul de cod este, de obicei,un segment read-only.

C.Segementul de date

Segmentul de date conține variabilele globale definite într-un program şi variabilele de tipul read-only.

În funcție de tipul de date există mai multe sub-tipuri de zone de date :

.data

Zona .data conține variabilele globale inițializate la valori nenule ale unui program.

De exemplu:
static int a = 3;
char b = 'a';

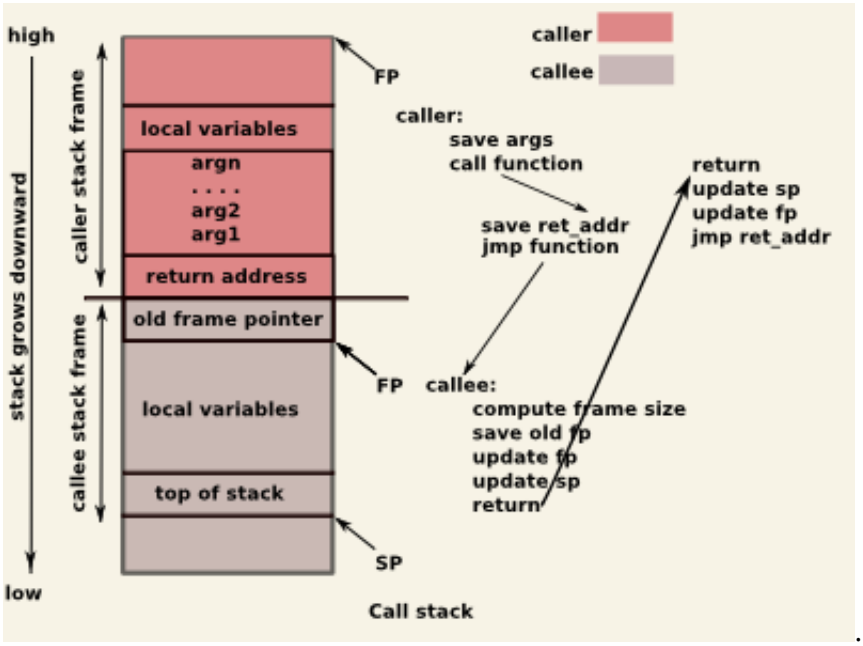
.bss
Zona .bss conține variabilele globale neinițializate sau inițializate la zero ale unui program.

De exemplu:
static int a;
char b;

În general acestea nu vor fi prealocate în executabil,ci în momentul creării precesului. Alocare zonei .bss se face peste pagini fizice zero (zeroed frames).

.rodata
Zona .rodata conține informație care poate fi doar citită, nu și modificată.
Aici sunt stocate constantele:
const int a;
*const char *ptr;*
i literalii:
"Hello, World!"

Stiva
Stiva este o regiune dinamică în cadrul unui proces. Stiva este folosită pentru a reține "stack frame-urile" (link) în cazul apelurilor de funcții și pentru a stoca variabilele locale. Pe marea majoritate a arhitecturilor moderne stiva crește în jos și heap-ul crește în sus. Stiva este gestionată automat de compilator. La fiecare revenire din funcție stiva este golită.



.

Heap-ul

Heap-ul este zona de memorie dedicată alocarii dinamice a memoriei. Heap-ul este folosit pentru alocarea de regiuni de memorie a caror dimensiune se află doar la runtime.

Spre deosebire de stivă, însă, heap-ul nu este gestionat de compilator. Este de datoria programatorului să știe câtă memorie trebuie sa aloce și să rețină cât a alocat și când trebuie să dezaloce. Problemele frecvente în majoritatea programelor sunt de pierderea referințelor la zonele alocate (memory leaks) sau referirea de zone nealocate sau insuficient alocate(accese invalide).

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

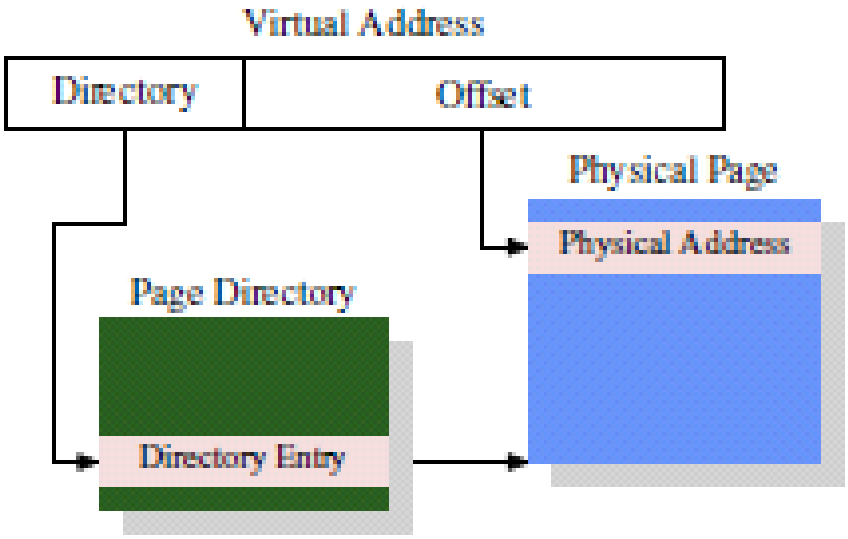
MEMORIA VIRTUALĂ

A.Introducere

Memoria cache asigură accesul rapid la informaţiile cel mai recent utilizate, astfel că memoria principală poate acţiona ca un cache pentru memoria secundară. Această tehnică se numeşte memorie virtuală. Memoria virtuală este un mecanism folosit pentru a extinde limitele memoriei fizice. Într-un sistem cu memorie virtuală, aceasta apare utilizatorului integral din spaţiul logic de adrese,fiind disponibil pentru memorare. In orice moment, numai câteva pagini din spaţiul logic de adrese sunt mapate peste spaţiul fizic. Alte pagini nu sunt prezente în memoria principală. În schimb, informaţia din aceste pagini este memorată temporar într-o memorie secundară de tip disc (al cărui cost pe bit este mult mai scăzut). Ori de câte ori este accesată o pagină care lipseşte, sistemul de operare încarcă pagina respectivă de pe disc şi memorează pe disc o altă pagină, care nu a fost recent referită. Prin acest mecanism, utilizatorul are impresia unei memorii fizice uriaşe, dar mai lente.

B.Translatarea adresei

O parte interesantă este translatarea unei adrese virtuale la o adresă fizică.Unitatea de management a memoriei (UMM) poate sa remapeze adrese pagină cu pagină. La fel ca atunci când abordează linii de cache, adresa virtuala este împărţită în părţi distincte.Aceste componente sunt folosite ca index în numeroase tabele, care ulterior sunt utilizate în construcţia adresei fizice finale.Pentru cel mai simplu model, avem doar un singur nivel de tabele.



Pentru un exemplu concret figura de mai sus este tiparul folosit pentru 4MB pagini pentru sistemul de calcul x86.Partea de offset a adresei virtuale are dimensiunea de 22 de biți care este suficientă pentru a aborda fiecare octet din cei 4MB ai paginei. Restul de 10 de biţi de adrese virtuale selecteaza unul dintre cele 1024 de intrări în directorul pagina. Fiecare intrare conţine o adresă de 10 biţi de bază a unei pagini de 4MB care este combinată cu offset-ul pentru a forma o adresă completă de 32 de biţi.

Algoritmii de scoaterea a paginii sunt următoarele:

a) Algoritmul aleator

După cum spune și numele este vorba despre o soluţie aleatoare.Soluția nu este bună pentru că sistemul de operare poate schimba o pagină des folosită.

b) Algoritmul optim

Modul de desfășurare este următorul:după ce se înregistrează necesarul de pagini pentru un program,se numără câte instrucțiuni se execută fără a face saltul la alte pagini;în momentul în care apare un salt la altă pagină se calculează în fiecare pagină din memorie unde există cele mai multe instrucțiuni fară salt.Aceste pagini,care amână cel mai mult saltul în alte pagini,sevor păstra în memorie.

c) Algoritmul NRU(Not Recently Used)

Acest algoritm scoate din memorie paginile ce nu au fost referite de curînd. Se alege din setul de pagini o pagină oarecare din clasa nevidă cea mai mică.

d) Algoritmul FIFO(First In First Out)

Se bazează pe principiul căţtilor introduse în bibliotecă:cărțile folosite se adaugă în dreaptă și la stânga se va afla cartea cea mai puţin folosită.

e) Algoritmul SECOND CHANCE

Reprezintă o variantă a algoritmului FIFO.

f) Metoda clock

Se alcătuiesc cozi circulare și se iau pe rând paginile în ordine(algoritmul acorda şanse egale tuturor celor din coadă).

g) **Algoritmul LRU(Last Recently Used)**

Ideea acestui algoritm este de a determina cea mai nefolosită pagină.Cu alte cuvinte se bazează pe faptul că paginile care au fost folosite până acum vor fi folosite și de acum încolo,iar paginile care nu au fost folosite de mult nu vor fi folosite mult timp de acum încolo. În principiu se va elimina pagina ce nu a fost folosită de cel mai mult timp.

C.Memoria comună în Win32 API

În afară de algoritmii de paginare Windows folosește și mapare de fișiere pentru crearea memoriei virtuale.De la Windows XP, dimensiunea ce poate fi mapată a crescut la 200GB pentru sisteme pe 32 biți(aceasta estevaloarea maximă,in general se folosec 4GB).

Prezentare generală pentru a crea o regiune de comun, memorie, folosind fişiere mapate:

API-ul Win32 implică,mai întâi,crearea a unei cartografii pentru fişier şi de instituire,după care a unei vederi a fişierului mapat.Al doilea proces implică crearea uneivederi a fişierului

mapat în spaţiul de adrese virtuale.Fişierul mapat reprezintă obiectul memorie partajat, care va permite comunicarea între procesoare.

În acest exemplu,într-un prim proces,producătorul creează un obiect partajat de memorie cu ajutorul memoriei de cartografiere caracteristică,disponibile în API Win32.Producătorul scrie apoi un mesaj pentru a partaja memoria.Consumatorul deschide memoria obiect partajată şi citeşte mesajul scris de către producător.Pentru a definii un fişier de memorie mapată, un proces deschide prima dată fişierul de a fi mapat cu funcţia *CreateFile(),* care returnează un HANDLE pentru a deschide fișierul.Procesul creează apoi o cartografiere a acestui fişier HANDLE folosind funcția *CreateFileMapping()*. Odată ce fişierul cartografiat este definit, procesul definește o vedere a fişierului mapat în spaţiul său de adrese virtuale cu funcția *FileCMapViewOf().*

```
#ir.clude <windows . h>
#irdude <stdio.h>
int main(int argc,  char *argv[]i)
{
HANDLE hFile, hKapFile;
LPVCID lpMapAddress;
hFile = CreateFile ( "temp, txt" , /,/ file name
GENERIC-READ | GENERIC-WRITE, // read/write access
0, // no sharing of the file
NULL, // default security
OPEN_ALWAYS, /./ open new or existing  file
FILE_ATTRIBUTE_NORMAL, // routine file attributes
NULL) ; /./ no file template
hKapFile = CreateFileMapping(hFile, // file handle
NULL, /./ default security
PAGE_READWRITE, // read/write access ;o mapped pages
0, // map entire file
0,
TEXT("SharedObject")); // named shared memory object
lpMapAddress = MapViewOfFile(hMapFile, // mapped object handle
FILE_MAP_ALLJ\CCESS, // read/write access
0, // mapped view of entire file
0,
0) ;
/./ write to shared memory
sprintf(lpMapAddress,"Shared memory message");
UnmapViewOfFile (lpMapAddress) ,-
CloseHandle(hFile);
CloseHandle (hMapFile) ,•
}
```

4.ALOCAREA MEMORIEI

Alocarea memoriei este realizată static de compilator sau dinamic, în timpul execuției. Alocarea statică este realizată în segmentele de date pentru variabilele locale sau pentru literali. În timpul execuției, variabilele se alocă pe stivă sau în heap. Alocarea pe stivă se realizează automat de compilator pentru variabilele locale unei funcții (mai puțin variabilele locale prefixate de identificatorul static).
Alocarea dinamică se realizează în heap și are loc atunci când nu se știe în momentul compilării câtă memorie va fi necesară pentru o variabilă,o structură,un vector. Daca se știe din Momentul compilării cât spațiu va ocupa o variabilă, se recomandă alocarea ei statică, pentru a preveni erorile frecvente apărute în contextul alocării dinamice.
Pentru a fragmenta cât mai puțin spațiul de adrese al procesului, ca urmare a alocărilor și dezalocărilor unor zone de dimensiuni variate, alocatorul de memorie va organiza segmentul de date alocate dinamic sub forma de heap, de unde și numele segmentului.

A.Alocarea memoriei în Linux

În Linux alocarea memoriei pentru procesele utilizator se realizează prin intermediul funcțiilor de bibliotecă *malloc*, *calloc* și *realloc*,iar dezalocarea ei prin intermediul funcției *free*. Aceste funcții reprezintă apeluri de bibliotecă și rezolvă cererile de alocare și dezalocare de memorie pe cât posibil în user space. Așadar,se țin niște tabele care specifică zonele de memorie alocate în heap. Dacă există zone libere pe heap, un apel *malloc* care cere o zona de memorie care poate fi încadrată într-o zona liberă din heap va fi satisfacut imediat marcând în tabel zona respectivă ca fiind alocată și întorcând programului apelant un pointer spre ea. Dacă,în schimb,se cere o zonă care nu încape în nicio zonă liberă din heap, *malloc* va încerca extinderea heap-ului prin apelul de sistem *brk* sau *mmap*.

*void *calloc(size_t nmemb, size_t size);*
*void *malloc(size_t size);*
*void *realloc(void *ptr, size_t size);*
*void free(void *ptr);*

Memoria alocată de proces este eliberată automat la terminarea procesului,însă în cazul unui proces server, de exemplu, care ruleaza foarte mult timp și nu eliberează memoria Alocată,acesta va ajunge să ocupe toata memoria disponibilă în sistem cauzând astfel consecințe nefaste. Atenție să nu se elibereze de doua ori aceeași zonă de memorie,întrucât acest lucru va avea drept urmare coruperea tabelelor ținute de *malloc* ceea ce va duce din nou la consecințe nedorite. Întrucât funcția *free* se întoarce imediat dacă primește ca parametru un pointer NULL, este recomandat ca dupa un apel *free*, pointer-ul să fie resetat la NULL.

Câteva exemple de alocare a memoriei sunt prezentate în continuare:

int n = atoi(argv[1]);
*char *str;*

/* de obicei malloc-ul primește argumentul de spațiu în forma size_elems * num_elems */
*str = (char *) malloc((n + 1) * sizeof(char));*

```
if (NULL == str) {
perror("malloc");
exit(EXIT_FAILURE);
}
[...]
free(str);
str = NULL;
---
/* crearea unui vector de șiruri ce conține doar argumentele unui program */
char **argv_no_exec;

/* se alocă spațiu pentru argumente */

argv_no_exec = (char **) malloc((argc - 1) * sizeof(char*));
if (NULL == argv_no_exec) {
perror("malloc");
exit(EXIT_FAILURE);
}

/* se creează referințe către argumente */

for (i = 1; i < argc; i++)
argv[in-o1]e x=e cargv[i];
[...]
free(argv_no_exec);
argv_no_exec = NULL;
```

Apelul *realloc* este folosit pentru modificarea spațiului de memorie alocat după un apel *malloc*:

```
int *p;
p = (int *) malloc(n * sizeof(int));
if (NULL == p) {
perror("malloc");
exit(EXIT_FAILURE);
}
[...]
p = (int *) realloc(p, (n + extra) * sizeof(int));
[...]
free(p);
p = NULL;
```

Apelul *calloc* este folosit pentru alocarea de zone de memorie al caror conținut este nul (plin de valori de zero). Spre deosebire de *malloc*, apelul va primi două argumente: numărul de elemente și dimensiunea unui element.

```
list_t *list_v; /* list_t poate fi orice tip de date din C (mai putin void) */
list_v = (list_t *) calloc(n, sizeof(list_t));
if (NULL == list_v) {
perror("calloc");
exit(EXIT_FAILURE);
}
[...]
```

```
free(p);
p = NULL;
```

B.Alocarea memoriei în Windows

În Windows un proces poate să-i creeze mai multe obiecte Heap pe lânga Heap-ul cu care este creat procesul. Acest lucru este foarte util în momentul în care o aplicație alocă și dezalocă foarte multe zone de memorie cu câteva dimensiuni fixe. Aplicația poate sa-i creeze câte un Heap pentru fiecare dimensiune și, în cadrul fiecarui Heap, să aloce zone de aceeași dimensiune reducând astfel la maxim fragmentarea heapului.

Pentru crearea, respectiv distrugerea, unui Heap se vor folosi funcțiile *HeapCreate* și *HeapDestroy*:

```
HANDLE HeapCreate(
DWORD flOptions,
SIZE_T dwInitialSize,
SIZE_T dwMaximumSize
);
BOOL HeapDestroy(
HANDLE hHeap
);
```

Pentru a obține un descriptor al heap-ului cu care a fost creat procesul (în cazul în care nu dorim crearea altor heapuri) se va apela funcia *GetProcessHeap*. Pentru a obține descriptorii tuturor heapurilor procesului se va apela *GetProcessHeaps*.
Există, de asemenea funcții care enumeră toate blocurile alocate într-un heap, validează unul sau toate blocurile alocate într-un heap sau întorc dimensiunea unui bloc pe baza descriptorului de heap și a adresei blocului: *HeapWalk, HeapSize, HeapValidate*.

Pentru alocarea, dezalocarea, redimensionarea unui bloc de memorie din Heap, Windows pune la dispoziția programatorului funcțiile *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
);
```

Câteva exemple de folosire a acestor funcții sunt prezentate în continuare:

```
/* alocarea unui vector de intregi */
HANDLE processHeap;
DWORD *data;
processHeap = GetProcessHeap();
if (NULL == processHeap) {
fprintf(stderr, "GetProcessHeap failed with error %ud.\n", GetLastError());
ExiProcess;
}
data = HeapAlloc(processHeap, HEAP_ZERO_MEMORY, count * sizeof(DWORD));
if (NULL == data) {
fprintf(stderr, "HeapAlloc failed with error %ud.\n", GetLastError());
Exi(t-P1r)o;cess
}
[...]
if (HeapFree(processHeap, 0, data) == FALSE) {
fprintf(stderr, "HeapFree failed with error %ud.\n", GetLastError());
ExiProcess;
}
```

```
/* alocarea unei matrice */
HANDLE processHeap;
DWORD **mat;
INT m, n;
INT i;
processHeap = GetProcessHeap();
if (NULL == processHeap) {
fprintf(stderr, "GetProcessHeap failed with error %ud.\n", GetLastError());
ExiProcess;
}
mat = HeapAlloc(processHeap, 0, m * sizeof(*mat));
if (NULL == mat) {
fprintf(stderr, "HeapAlloc failed with error %ud.\n", GetLastError());
ExiProcess;
}
for (i = 0; i < n; i++) {
[ i ] = mHaetapAlloc(processHeap, 0, n * sizeof(**mat));
if (NULL == mat) {
fprintf(stderr, "HeapAlloc failed with error %ud.\n", GetLastError());
goto freeMem;
}
}
[...]
freeMem:
for (j = 0; j < i; j++)
(HperaopcFerseseHeap, 0, mat[j]);
HeapFree(processHeap, 0, mat);
```

Pe sistemele Windows se pot folosi și funcțiile bibliotecii standard C pentru gestiunea memoriei: *malloc,realloc, calloc, free,* dar apelurile de sistem specifice Windows oferă funcționalități suplimentare și nu implică legarea bibliotecii standard C în executabil.

C.Dezalocarea memoriei

Pentru dezalocarea memoriei, se folosesc funcțiile *free*, respectiv *HeapFree*. Funcțiile primesc ca argument un pointer la un spațiu de memorie alocat anterior cu o funcție de alocare. Daca se omite dezalocarea unei zone de memorie, aceasta va rămâne alocată pe întreaga durata de rulare a procesului. Ori de câte ori nu mai este nevoie de o zoăa de memorie, aceasta trebuie dezalocata pentru eficiența utilizării spațiului de memorie.
Nu trebuie neapărat realizată dezalocarea diverselor zone înainte de un apel *exit* sau *ExitProcess* sau înainte de încheierea programului pentru că acestea sunt automat eliberate de sistemul de operare.
Probleme pot apărea și dacă se încearcă dezalocarea aceleiași regiuni de memorie de două sau mai multe ori și se corup listele.

5.PROBLEME DE LUCRU CU MEMORIA

Lucrul cu heap-ul este una dintre cauzele principale ale apariţiilor problemelor de programare. Lucrul cu pointerii, necesitatea folosirii unor apeluri de sistem/bibliotecă pentru alocare/dezalocare, pot conduce la o serie de probleme care afectează (de multe ori fatal) funcţionarea unui program.
Problemele cele mai des întâlnite în lucrul cu memoria sunt:
· accesul invalid la memorie
· leak-urile de memorie
Accesul invalid la memorie presupune accesarea unor zone care nu au fost alocate sau au fost eliberate.
Leak-urile de memorie sunt situaţiile în care se pierde referinţa la o zona alocată anterior. Acea zona va rămâne ocupată până la încheierea procesului. Ambele probleme şi utilitarele care pot fi folosite pentru combaterea acestora vor fi prezentate în continuare.

A.Acces invalid

De obicei, accesarea unei zone de memorie invalide rezultă într-o eroare de pagina (page fault) şi terminarea procesului (în Unix înseamnă trimiterea semnalului SIGSEGV – afişarea mesajului 'Segmentation fault'). Totuşi,dacă eroarea apare la o adresă invalidă,dar într-o pagină validă, hardware-ul şi sistemul de operare nu vor putea sesiza acţiunea ca fiind invalidă. Acest lucru se datoreaza faptului că alocarea memoriei se face la nivel de pagină. Pot exista situaţii în care să fie folosită doar jumatate din pagină. Deci,cealaltă jumatate conţine adrese invalide, sistemul de operare nu va putea detecta accesele invalide la acea zonă.
Asemenea accese pot duce la coruperea heap-ului şi la pierderea consistenţei memoriei alocate. Dupa cum se va vedea în continuare, există utilitare care ajută la detectarea acestor situaţii.
Un tip special de acces invalid este buffer overflow. Acest tip de atac presupune referirea unor regiuni valide din spaţiul de adresă al unui proces prin intermediul unei variabile care nu ar trebui să poată referenţia aceste adrese. De obicei, un atac de tip buffer overflow rezultă în rularea de cod nesigur. Protecţia la accese de tip buffer overflow se realizează prin verificarea limitelor unui buffer/vector fie la compilare, fie la rulare.

B. GDB - Detectarea zonei de acces invalid de tip page fault

Pe lânga facilităţi de bază,precum urmărirea unei variabile sau configurarea de puncte de oprire (breakpoints),GDB pune la dispoziţia utilizatorilor şi comenzi avansate, utile în anumite cazuri.
Comanda *disassamble* poate fi folosită pentru a afişa codul *main* generat de compilator.
Comanda *info reg* afişeaza conţinutul registrelor. Aceste comenzi sunt folosite rar, atunci când utilizatorul încearcă să depaneze codul generat de compilator, sau când are părţi din program scrise direct în asamblare.
O comandă foarte utilă atunci când se depanează programe complexe este *backtrace*. Această comandă fişeaza toate apelurile de funcţii în curs de execuţie.

Exemplu: fibonacci_gdb_test.c

```
#include <stdio.h>
#include <stdlib.h>
int fibonacci(int no)
{
if (1 == no || 2 == no)
return 1;
return fibonacci(no-1) + fibonacci(no-2);
}
int main()
{
short int numar, baza=10;
char sir[1];
scanf("%s", sir);
s t rntuomla(rs=ir, NULL, baza);
printf("fibonacci(%d)=%d\n", numar, fibonacci(numar));
return 0;
}
```

6.BIBLIOGRAFIE

1. What Every Programmer Should Know About Memory
2. Linux System Programming - Chapter 8 - Memory Management
3. Windows System Programming - Chapter 5 - Memory Management (Win32 and Win64 Memory Management Architecture, Heaps, Managing Heap Memory)
4. Linux Application Programming - Chapter 7 - Memory Debugging Tools
5. Windows Memory Management
6. Virtual Memory Allocation and Paging
7. GDB manual