Laborator 04 - Gestiunea memoriei

Materiale ajutătoare

- lab04-slides.pdf
- lab04-refcard.pdf

Nice to read

- TLPI - Chapter 7, `Memory Allocation`

Gestiunea memoriei

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

Rolul subsistemului de gestiune a memoriei este de :

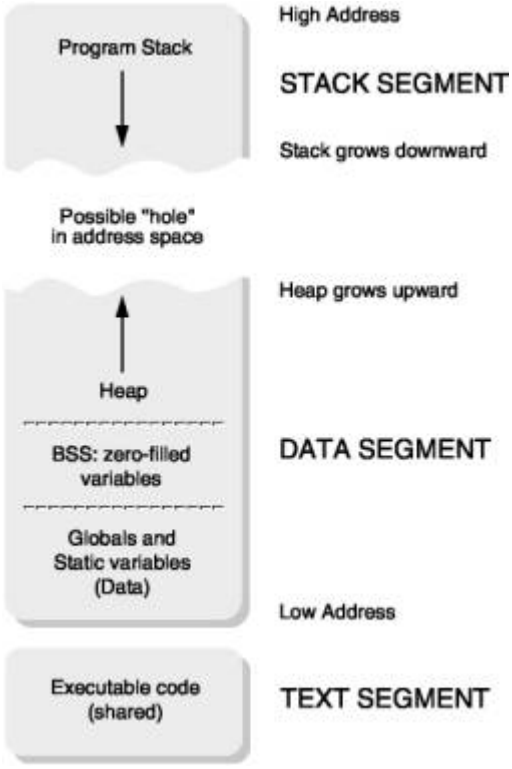
- a ţine evidenţa zonelor de memorie fizică (ocupate sau libere)
- a oferi proceselor sau celorlalte subsisteme acces la memorie
- a mapa paginile de memorie virtuală ale unui proces (pages) peste paginile fizice (frames).

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

Din păcate, nivelul limitat de înţelegere a acestor interfeţe şi a acţiunilor ce se petrec în spate conduc la o serie de probleme foarte des întâlnite în aplicaţiile software: memory leak-uri, accese nevalide, suprascrieri, buffer overflow, corupere de zone de memorie.

Este, în consecinţă, fundamentală cunoaşterea contextului în care acţionează subsistemul de gestiune a memoriei şi înţelegerea interfeţei pusă la dispoziţie de sistemul de operare programatorului.

Spaţiul de adresă al unui proces



Spaţiul de adrese al unui proces, sau, mai bine spus, spaţiul virtual de adresă al unui proces reprezintă zona de memorie virtuală utilizabilă de un proces. Fiecare proces are un spaţiu de adresă propriu. Chiar în situaţiile în care două procese partajează o zonă de memorie, spaţiul virtual este distinct, dar se mapează peste aceeaşi zonă de memorie fizică.

În figura alăturată este prezentat un spaţiu de adresă tipic pentru un proces. În sistemele de operare moderne, în spaţiul virtual al fiecărui proces se mapează memoria nucleului, aceasta poate fi mapată fie la început, fie la sfârşitul spaţiului de adresă. În continuare, ne vom referi numai la spaţiul de adresă din user-space pentru un proces.

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

Zona de cod

Segmentul de cod (denumit şi `text segment`) reprezintă instrucţiunile în limbaj maşină ale programului. Registrul de tip `instruction pointer` (IP) va referi adrese din zona de cod. Se citeşte instrucţiunea indicată de către IP, se decodifică şi se interpretează, după care se incrementează contorul programului şi se trece la următoarea instrucţiune.
Zona de cod este, de obicei, o zonă read-only pentru ca procesul să nu poată modifica propriile instrucţiuni prin folosirea greşită a unui pointer. Zona de cod este partajată între toate procesele care rulează acelaşi program. Astfel, o singură copie a codului este mapată în spaţiul de adresă virtual al tuturor proceselor.

Zone de date

Zonele de date conţin variabilele globale definite într-un program şi variabilele de tipul read-only. În funcţie de tipul de date există mai multe subtipuri de zone de date.

.data

Zona `.data` conţine variabilele globale şi statice *iniţializate* la valori nenule ale unui program. De exemplu:

```
static int a = 3;
char b = 'a';
```

.bss

Zona `.bss` conţine variabilele globale şi statice *neiniţializate* ale unui program. Înainte de execuţia codului, acest segment este iniţializat cu 0. De exemplu:

```
static int a;
char b;
```

În general aceste variabile nu vor fi prealocate în executabil, ci în momentul creării procesului. Alocarea zonei .bss se face peste pagini fizice zero (zeroed frames).

.rodata

Zona .rodata conţine informaţie care poate fi doar citită, nu şi modificată. Aici sunt stocate *constantele*:

```
const int a;
const char *ptr;
```

şi *literalii*:

```
"Hello, World!"
"En Taro Adun!"
```

Stiva

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

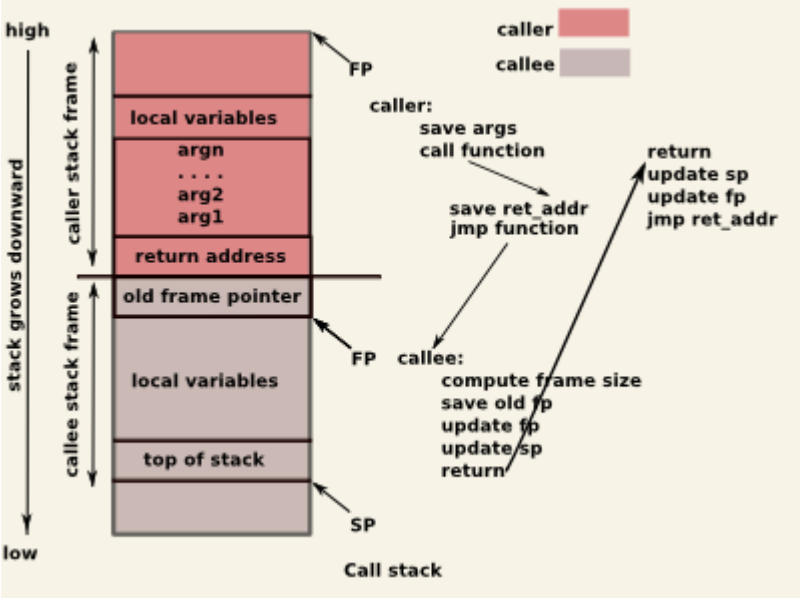
Stiva este folosită pentru a stoca "stack frame-uri". Pentru fiecare apel de funcţie se va crea un nou "stack frame".

Un "stack frame" conţine:

- variabile locale
- argumentele funcţiei
- adresa de retur

Pe marea majoritate a arhitecturilor moderne stiva creşte în jos şi heap-ul creşte în sus. Stiva creşte la fiecare apel de funcţie şi scade la fiecare revenire din funcţie.

În figura de mai jos este prezentată o vedere conceptuală asupra stivei in momentul apelului unei funcţii.



Heap-ul

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

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

În limbaje precum Java, Lisp etc. unde nu există "pointer freedom", eliberarea spaţiului alocat se face automat prin

intermediul unui garbage collector. Pe aceste sisteme se previne problema pierderii referinţelor, dar încă rămâne activă problema referirii zonelor nealocate.

Alocarea/Dealocarea memoriei

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

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

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

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

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

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

Nu trebuie neapărat realizată dealocarea diverselor zone înainte de un apel exit sau înainte de încheierea programului pentru că acestea sunt automat eliberate de sistemul de operare.

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

Alocarea memoriei în Linux

În Linux, alocarea memoriei pentru procesele utilizator se realizează prin intermediul funcţiilor de bibliotecă malloc, calloc şi realloc, iar dealocarea ei prin intermediul funcţiei free. Aceste funcţii reprezintă apeluri de bibliotecă şi rezolvă cererile de alocare şi dealocare de memorie, pe cât posibil, în user space.

Implementarea funcţiei malloc

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

Întotdeauna eliberaţi (free) memoria alocată. Memoria alocată de proces este eliberată automat la terminarea procesului, însă, de exemplu în cazul unui proces server care rulează foarte mult timp şi nu eliberează memoria alocată acesta va ajunge să ocupe toată memoria disponibilă în sistem, având astfel consecinţe nefaste.

Atenţie! Nu eliberaţi de două ori aceeaşi zonă de memorie întrucât acest lucru va avea drept urmare coruperea tabelelor ţinute de malloc ceea ce va avea din nou consecinţe nefaste. Întrucât funcţia free se întoarce imediat dacă primeşte ca parametru un pointer `NULL`, este recomandat ca după un apel free, pointer-ul să fie resetat la `NULL`.

În continuare, sunt prezentate câteva exemple de alocare a memoriei folosind malloc:

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

/* usually malloc receives the size argument like:
   num_elements * size_of_element */
str = malloc((n + 1) * sizeof(char));
if (NULL == str) {
        perror("malloc");
        exit(EXIT_FAILURE);
}

[...]

free(str);
```

```
str = NULL;
```

```
/* Creating an array of references to the arguments received by a program */
char **argv_no_exec;

/* allocate space for the array */
argv_no_exec = malloc((argc - 1) * sizeof(char*));
if (NULL == argv_no_exec) {
        perror("malloc");
        exit(EXIT_FAILURE);
}

/* set references to the program arguments */
for (i = 1; i < argc; i++)
        argv_no_exec[i-1] = argv[i];

[...]

free(argv_no_exec);
argv_no_exec = NULL;
```

Apelul realloc este folosit pentru modificarea spaţiului de memorie alocat cu un apel malloc sau calloc:

```
int *p;

p = malloc(n * sizeof(int));
if (NULL == p) {
        perror("malloc");
        exit(EXIT_FAILURE);
}

[...]

p = realloc(p, (n + extra) * sizeof(int));

[...]

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

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

```
list_t *list_v; /* list_t could be any C type ( except void ) */

list_v = calloc(n, sizeof(list_t));
if (NULL == list_v) {
        perror("calloc");
        exit(EXIT_FAILURE);
}

[...]

free(list_v);
list_v = NULL;
```

Atenţie Conform standardului C, este redundant (şi considerat bad practice) să faceţi *cast* la valoarea întoarsă de malloc.

```
int *p = (int *)malloc(10 * sizeof(int));
```

malloc întoarce *void* * care în C este automat convertit la tipul dorit. Mai mult, dacă se face cast, iar headerul `stdlib.h` necesar pentru funcţia malloc nu este inclus, nu se va genera eroare! Pe anumite arhitecturi, acest caz poate conduce la un comportament nedefinit. Spre deosebire de C, în C++ este nevoie de cast. Mai multe detalii despre această problemă: aici

Mai multe informaţii despre funcţiile de alocare găsiţi în manualul bibliotecii standard C şi în pagina de manual **man malloc**.

Alocarea memoriei în Windows

În Windows, un proces poate să-şi creeze mai multe obiecte `Heap` pe lângă `Heap`-ul implicit al procesului. Acest lucru este foarte util în momentul în care o aplicaţie alocă şi dealocă foarte multe zone de memorie cu câteva dimensiuni fixe. Aplicaţia poate să-şi creeze câte un `Heap` pentru fiecare dimensiune şi, în cadrul fiecărui `Heap`, să aloce zone de aceeaşi dimensiune reducând 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 (în cazul în 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 într-un heap, validează unul sau toate blocurile alocate într-un heap sau întorc 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
);
```

În 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 în executabil.

Lucru cu memoria - Probleme

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/dealocare, 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 nevalid la memorie - ce prespune accesarea unor zone care nu au fost alocate sau au fost eliberate.
- leak-urile de memorie - situaţiile în care se pierde referinţa la o zonă alocată anterior. Acea zonă va rămâne ocupată până la încheierea procesului.

Ambele probleme şi utilitarele care pot fi folosite pentru combaterea acestora vor fi prezentate în continuare.

Acces nevalid

De obicei, accesarea unei zone de memorie nevalide rezultă într-o eroare de pagină (page fault) şi terminarea procesului (în Unix înseamnă trimiterea semnalului SIGSEGV afişarea mesajului 'Segmentation fault'). Totuşi, dacă eroarea apare la o adresă nevalidă, dar într-o pagină validă, hardware-ul şi sistemul de operare nu vor putea sesiza acţiunea ca fiind nevalidă. Acest lucru se datorează faptului că alocarea memoriei se face la nivel de pagină. Spre exemplu, pot exista situaţii în care să fie folosită doar jumătate din pagină. Deşi cealaltă jumătate conţine adrese nevalide, sistemul de operare nu va putea detecta accesele nevalide la acea zonă. Mai multe detalii în laboratorul de Memorie virtuală

Asemenea accese pot duce la coruperea heap-ului şi la pierderea consistenţei memoriei alocate. După cum se va vedea în continuare, există utilitare care ajută la detectarea acestor situaţii.

Un tip special de acces nevalid 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. Protejarea împotriva atacurilor de tip buffer overflow se realizează prin verificarea limitelor unui buffer/vector fie la compilare, fie la rulare.

GDB - Detectarea zonei de acces nevalid de tip page fault

O comandă foarte utilă atunci când se depanează programe complexe este **`backtrace`**. Această comandă afişează toate apelurile de funcţii în curs de execuţie.

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);
                                                                                                    < a
href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("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 afişarea apelurilor de funcţii se listează şi parametrii cu care a fost apelată funcţia. Acest lucru este posibil datorită faptului că atât variabilele locale, cât şi parametrii acesteia sunt păstraţi pe stivă până la ieşirea din funcţie. (pentru detalii, revedeţi secţiunea despre stivă)

Fiecare funcţie are alocată pe stivă un frame, în care sunt plasate variabilele locale funcţiei, parametrii funcţiei şi adresa de revenire din funcţie. În momentul în care o funcţie este apelată, se creează un nou frame prin alocarea de spaţiu pe stivă de către funcţia apelată. Astfel, dacă avem apeluri de funcţii imbricate, atunci stiva va conţine toate frame-urile tuturor funcţiilor 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 depăşeşte dimensiunea buffer-ului alocat (static). Acest tip de eroare poartă denumirea de *buffer overflow şi este extrem de gravă. Cele mai multe atacuri de la distanţă pe un sistem sunt cauzate de acest tip de erori. Din păcate, acest tip de eroare nu este uşor 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 afectat?.*

```
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 funcţia fibonacci a fost apelată cu valoarea 0. Cum funcţia 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 funcţia 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
 class="code bash" = 0
(gdb) print baza
  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)  = 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 primeşte ca parametru o expresie şi va opri execuţia 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 funcţia `_IO_vfscanf`, care la rândul ei este apelată de către functia `scanf`. Dacă se analizează apoi parametrii pasaţi functiei `scanf`, se observă imediat cauza erorii.

Pentru mai multe informaţii despre GDB consultaţi documentaţia online (alternativ pagina info - `info gdb`) sau folosiţi comanda `help` din cadrul GDB.

mcheck - verificarea consistenţei heap-ului

glibc permite verificarea consistenţei heap-ului prin intermediul apelului mcheck definit în `mcheck.h`. Apelul mcheck forţează `malloc` să execute diverse verificări de consistenţă precum scrierea peste un bloc alocat cu `malloc`.

Alternativ, se poate folosi opţiunea `-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 execuţie cu variabila `MALLOC_CHECK_` configurată, atunci vor fi afişate mesaje de eroare (eventual programul va fi terminat forţat - aborted).

În continuare, este prezentat un exemplu de cod cu probleme în alocarea şi 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 şi rulat. Mai întâi este rulat fără opţiuni de mcheck, după care se defineşte variabila de mediu MALLOC_CHECK_ la rularea programului. Se observă că deşi se depăşeşte spaţiul alocat pentru vectorul `v1` şi se referă vectorul **după** eliberarea spaţiului, o rulare simplă nu rezultă în afişarea nici unei erori.

Totuşi, 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 soluţie completă şi nu detectează toate erorile ce pot apărea în lucrul cu memoria. Detectează, totuşi, un număr important de erori şi reprezintă o facilitate importantă a glibc.

O descriere completă găsiţi în pagina asociată din manualul glibc.

Leak-uri de memorie

Un leak de memorie apare în două situaţii:

- un program omite să elibereze o zonă de memorie
- un program pierde referinţa la o zonă de memorie alocată şi, drept consecinţă, nu o poate elibera

Memory leak-urile au ca efect reducerea cantităţii de memorie existentă în sistem. Se poate ajunge, în situaţii extreme, la consumarea întregii memorii a sistemului şi la imposibilitatea de funcţionare a diverselor aplicaţii ale acestuia.

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

Valgrind

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

În continuare, ne vom referi doar la utilitarul Memcheck de detectare a erorilor de lucru cu memoria. Mai precis, acest utilitar detectează următoarele tipuri de erori:

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

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

În exemplul de rulare, de mai jos, se foloseşte programul prezentat la secţiunea ''mcheck'':

```
so@spook$ valgrind --tool=memcheck ./mcheck_test
==17870== Memcheck, a memory error detector.
```

```
==17870== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==17870== Using LibVEX rev 1804, a library for dynamic binary translation.
==17870== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==17870== Using valgrind-3.3.0-Debian, a dynamic binary instrumentation framework.
==17870== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==17870== For more details, rerun with: -v
==17870==
==17870== Invalid write of size 4
==17870==    at 0x4005B1: main (mcheck_test.c:17)
==17870==  Address 0x5184048 is 4 bytes after a block of size 20 alloc'd
==17870==    at 0x4C21FAB: malloc (vg_replace_malloc.c:207)
==17870==    by 0x400589: main (mcheck_test.c:10)
==17870==
==17870== Invalid write of size 4
==17870==    at 0x4005C8: main (mcheck_test.c:22)
==17870==  Address 0x5184048 is 4 bytes after a block of size 20 free'd
==17870==    at 0x4C21B2E: free (vg_replace_malloc.c:323)
==17870==    by 0x4005BF: main (mcheck_test.c:19)
==17870==
==17870== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 8 from 1)
==17870== malloc/free: in use at exit: 40 bytes in 1 blocks.
==17870== malloc/free: 2 allocs, 1 frees, 60 bytes allocated.
==17870== For counts of detected errors, rerun with: -v
==17870== searching for pointers to 1 not-freed blocks.
==17870== checked 76,408 bytes.
==17870==
==17870== LEAK SUMMARY:
==17870==    definitely lost: 40 bytes in 1 blocks.
==17870==      possibly lost: 0 bytes in 0 blocks.
==17870==    still reachable: 0 bytes in 0 blocks.
==17870==         suppressed: 0 bytes in 0 blocks.
==17870== Rerun with --leak-check=full to see details of leaked memory.
```

S-a folosit utilitarul Memcheck pentru obţinerea informaţiilor de acces la memorie.

Se recomandă folosirea opţiunii `-g` la compilarea programului pentru a include în executabil informaţii de depanare. În rularea de mai sus, Valgrind a identificat două erori: una apare la linia 17 de cod şi este corelată cu linia 10 (`malloc`), iar cealaltă apare la linia 22 şi este corelată cu linia 19 (`free`):

9

```
v1 = (int *) malloc (5 * sizeof(*v1));
if (NULL == v1) {
       perror ("malloc");
       exit (EXIT_FAILURE);
}

/* overflow */
v1[6] = 100;

free(v1);

/* write after free */
v1[6] = 100;
```

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;
}
```

În continuare, se prezintă comportamentul executabilului obţinut la o rulare obişnuită şi la o rulare sub Valgrind:

```
so@spook$ make
cc -Wall -g    valgrind_test.c   -o valgrind_test
so@spook$ ./valgrind_test
so@spook$ valgrind --tool=memcheck ./valgrind_test
==18663== Memcheck, a memory error detector.
==18663== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==18663== Using LibVEX rev 1804, a library for dynamic binary translation.
==18663== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==18663== Using valgrind-3.3.0-Debian, a dynamic binary instrumentation framework.
==18663== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==18663== For more details, rerun with: -v
==18663==
==18663== Conditional jump or move depends on uninitialised value(s)
==18663==    at 0x40050D: main (valgrind_test.c:10)
==18663==
==18663== Invalid write of size 1
==18663==    at 0x400554: main (valgrind_test.c:20)
==18663==  Address 0x5184031 is 1 bytes inside a block of size 70 free'd
==18663==    at 0x4C21B2E: free (vg_replace_malloc.c:323)
==18663==    by 0x40054B: main (valgrind_test.c:17)
==18663==
==18663== Invalid write of size 1
==18663==    at 0x40057C: main (valgrind_test.c:28)
==18663==  Address 0x51840e7 is 1 bytes before a block of size 10 alloc'd
==18663==    at 0x4C21FAB: malloc (vg_replace_malloc.c:207)
==18663==    by 0x40056E: main (valgrind_test.c:24)
==18663==
==18663== ERROR SUMMARY: 6 errors from 3 contexts (suppressed: 8 from 1)
==18663== malloc/free: in use at exit: 20 bytes in 2 blocks.
==18663== malloc/free: 3 allocs, 1 frees, 90 bytes allocated.
==18663== For counts of detected errors, rerun with: -v
==18663== searching for pointers to 2 not-freed blocks.
==18663== checked 76,408 bytes.
==18663==
==18663== LEAK SUMMARY:
==18663==    definitely lost: 20 bytes in 2 blocks.
==18663==      possibly lost: 0 bytes in 0 blocks.
==18663==    still reachable: 0 bytes in 0 blocks.
==18663==         suppressed: 0 bytes in 0 blocks.
==18663== Rerun with --leak-check=full to see details of leaked memory.
```

Se poate observa că, la o rulare obişnuită, programul nu generează nici un fel de eroare. Totuşi, la rularea cu Valgrind, apar erori în 3 contexte:

1. la apelul `strcat` (linia 10) şirul nu a fost iniţializat
2. se scrie în memorie după `free` (linia 20: `p[1] = 'a'`)
3. underrun (linia 28)

În plus, există leak-uri de memorie datorită noului apel `malloc` care asociază o nouă valoare lui `p` (linia 24).

Valgrind este un utilitar de bază în depanarea programelor. Este facil de folosit (nu este intrusiv, nu necesită modificarea surselor) şi permite detectarea unui număr important de erori de programare apărute ca urmare a gestiunii defectuoase a memoriei.

Informaţii complete despre modul de utilizare a Valgrind şi a utilitarelor asociate se găsesc în paginile de documentaţie Valgrind.

mtrace

Un alt utilitar care poate fi folosit la depanarea erorilor de lucru cu memoria este mtrace. Acest utilitar ajută la identificarea leak-urilor de memorie ale unui program.

Utilitarul mtrace se foloseşte cu apelurile mtrace şi muntrace implementate în biblioteca standard C:

```
void mtrace(void);
void muntrace(void);
```

Utilitarul mtrace introduce handlere pentru apelurile de biblioteca pentru lucrul cu memoria (`malloc`, `realloc`, `free`). Apelurile mtrace şi muntrace activează, respectiv dezactivează monitorizarea apelurilor de bibliotecă de lucru cu memoria.

Jurnalizarea operaţiilor efectuate se realizează în fişierul definit de variabila de mediu `MALLOC_TRACE`. După ce apelurile au fost înregistrate în fişierul specificat, utilizatorul poate să folosească utilitarul `mtrace` pentru analiza acestora.

În exemplul de mai jos este prezentată o situaţie 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 secvenţa de comenzi ce urmează se compilează fişierul de mai sus, se stabileşte fişierul de jurnalizare şi 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
```

Mai multe informaţii despre detectarea problemelor de alocare folosind mtrace găsiţi în pagina asociată din manualul glibc.

Dublă dealocare

Denumirea de "dublă dealocare" oferă o bună intuiţie asupra cauzei: eliberarea de două ori a aceluiaşi spaţiu 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:

dubla_dealocare.c

```
#include <stdlib.h>

int main(void)
{
        char *p;
```

```
        p = malloc(10);
        free(p);
        free(p);

        return 0;
}
```

Rularea executabilului obţinut din programul de mai sus duce la afişarea unui mesaj specific al glibc de eliberare dublă a unei regiuni de memorie şi terminare a programului:

```
so@spook$ make
cc -Wall -g    dfree.c   -o dfree
so@spook$ ./dfree
*** glibc detected *** ./dfree: double free or corruption (fasttop): 0x0000000000601010 ***
======= Backtrace: =========
/lib/libc.so.6[0x2b675fdd502a]
/lib/libc.so.6(cfree+0x8c)[0x2b675fdd8bbc]
./dfree[0x400510]
/lib/libc.so.6(__libc_start_main+0xf4)[0x2b675fd7f1c4]
./dfree[0x400459]
```

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

Alte utilitare pentru depanarea problemelor de lucru cu memoria

Utilitarele prezentate mai sus nu sunt singurele folosite pentru detectarea problemelor apărute in lucrul cu memoria. Alte utilitare sunt:

- dmalloc
- mpatrol
- DUMA
- Electric Fence, prezentat în laboratorul de Memorie virtuală

Exerciţii

În rezolvarea laboratorului folosiţi arhiva de sarcini lab04-tasks.zip

Observaţii: Pentru a vă ajuta la implementarea exerciţiilor din laborator, în directorul `utils` din arhivă există un fişier `utils.h` cu funcţii utile.

Linux

1. (**1.5 puncte**) Spaţiul de adresă al unui proces
   - Intraţi în directorul `1-adr_space` şi deschideţi sursa `adr_space.c`.
   - În alt terminal compilaţi şi rulaţi programul.
   - Observaţi zonele de memorie din executabil în care sunt salvate variabilele, folosind comanda: objdump -t adr_space | grep var
   - Explicaţi de ce apar doar unele variabile.
   - Ce semnfică `l` şi `g` din output-ul obţinut? (Hint: man objdump, flag)
   - Afişaţi conţinutul zonei '.rodata' folosind utilitarul `readelf` (Hint: man readelf, -x)
2. (**1 punct**) Alocarea, realocarea şi dealocarea memoriei
   - Intraţi în directorul `2-alloc`, compilaţi şi rulaţi programul `alloc`.
   - Folosiţi `valgrind` pentru a detecta eventualele probleme de lucru cu memoria şi corectaţi-le.
   - Hints:
     - De ce se generează **leak-uri de memorie**?
     - Revedeţi secţiunea Valgrind din laborator.
     - Revedeţi secţiunea Alocarea memoriei în Linux din laborator
3. (**1 punct**) Minitutorial GDB - rezolvarea unei probleme de tip `Segmentation Fault`
   - Intraţi în directorul `3-gdb` şi inspectaţi sursa.
   - Programul ar trebui să citească un mesaj de la `stdin` şi să-l afişeze.
   - Compilaţi şi rulaţi sursa.
     1. Rulaţi încă o dată programul din gdb
        - Revedeţi rularea unui program din gdb
     2. Pentru a identifica exact unde crapă programul folosiţi comanda backtrace
        - Pentru detalii despre comenzile din gdb: (gdb) help

3. Schimbaţi frame-ul curent cu frame-ul funcţiei `main` (gdb) frame main
   - Hints:
     - Revedeţi detectarea unui acces nevalid de tip page fault.
   - Inspectaţi valoarea variabilei `buf`: (gdb) print buf
4. Acum vrem să vedem de ce este `buf = NULL`
   - Mai întâi, omorâţi actualul proces: (gdb) kill
   - Puneţi un breakpoint la începutul funcţiei `main`: (gdb) break main
   - Rulaţi programul şi inspectaţi valoarea lui `buf` înainte şi după apelul funcţiei `malloc`
   - Hint:
     - Folosiţi `next` pentru a trece la instrucţiunea următoare, fără a urmări apelul de funcţie.
     - Explicaţi sursa erorii, apoi rezolvaţi-o.

4. (**1 punct**) Lucru cu memoria - **Valgrind**
   - Intraţi în directorul `4-struct`.
   - Completaţi fişierul `struct.c` conform comentariilor marcate cu `TODO`.
   - În funcţia `allocate_flowers` alocaţi memorie pentru `no` elemente de tip `flower_info`.
   - În funcţia `free_flowers` eliberaţi memoria alocată în funcţia `allocate_flowers`.
   - Rulează programul fără probleme? Corectaţi eventualele greşeli.
   - **Hints:**
     - Folosiţi opţiunea `--tool=memcheck` pentru `valgrind`.
     - Revedeţi secţiunea Valgrind din laborator.
5. (**2.5 puncte**) Stack overflow
   1. Intraţi în directorul 5-stack şi inspectaţi sursa.
      - Completaţi problemele marcate cu *TODO1*:
      - în funcţia `show_snapshot` iteraţi pe toată lungimea stivei şi afişaţi adresa şi valoarea de la adresa curentă
      - în funcţia `take_snapshot` salvaţi în structura de date ce reţine imaginea stivei câmpurile adresă şi valoare.
      - Hint:
        - Ce reţine structura `stack_elements`?
      - Funcţia `f2` pune pe stivă un vector de 3 întregi. În ce ordine sunt puse elementele vectorului pe stivă?
      - Compilaţi şi rulaţi.
      - Care este adresa de revenire din funcţia `f2`?
   2. Folosindu-vă de vectorul v **fortaţi execuţia** funcţiei `show_message` **fără** a o apela explicit. Astfel, după apelul funcţiei `f2`, fluxul programului nu se va mai întoarce în funcţia `f1`, ci va executa `show_message`.
   - Hint:
     - Urmăriţi comentariile marcate cu *TODO2*.
     - Revedeţi partea din laborator referitoare la stivă.
     - Ce rol au `bp/ebp` şi `sp/esp`?
     - Ce adrese trebuie salvate când se schimbă stack frame-ul?
6. (**1 punct**) Detectare probleme de lucru cu memoria - **mcheck**
   - În directorul `6-trim` analizaţi programul `trim.c`, compilaţi şi rulaţi executabilul `trim`.
   - Încercaţi să detectaţi problema folosind gdb (revedeţi tehnicile folosite la exerciţiul 3).
   - încercaţi să folosiţi mcheck pentru a detecta problema.
   - Corectaţi problema.
   - **Hints**:
     - Rulaţi programul folosind:
     - MALLOC_CHECK_=1 ./trim
     - Citiţi secţiunea mcheck din laborator.
7. (**1 punct**) Endianess
   - Intraţi în directorul `7-endian` şi inspectaţi sursa `endian.c`
   - Folosindu-vă de variabila `w` afişaţi numărul `n=0xDEADBEEF`
   - Ce tip de arhitectură se foloseşte? (big-endian sau little-endian)
   - Hint:
     - Gândiţi-vă la `n` ca la un vector de caractere
     - Mai multe detalii despre big-endian şi little-endian aici
8. (**0.5 punct**) Zone de stocare a variabilelor
   - Intraţi în directorul `8-counter`
   - Implementaţi funcţia `inc()` care întoarce de fiecare dată un întreg reprezentând numărul de apeluri până în momentul respectiv al funcţiei `inc`.
   - Nu aveţi voie să folosiţi variabile globale
9. (**1 puncte**) Zone de stocare a variabilelor
   - Intraţi în directorul `9-bad_stack`
     - Analizaţi fişierul `bad_stack.c`.
     - Compilaţi şi rulaţi programul.
     - Cum se explică rezultatul afişat?
       - De ce în funcţia `main`, prima oară se afişează valoarea din `str`, iar a doua oară nu?
     - Corectaţi funcţia `myfun()`. Găsiţi minim 3 metode de a rezolva problema.

BONUS Windows

1. (**1 so karma**) Realizarea unui wrapper pentru funcţiile `malloc` şi `free`
   - Deschideţi proiectul Visual Studio din directorul `malloc-wrapper`.
     - Inspectaţi cele două fişiere existente: `xmalloc.c` şi `xmalloc.h`.
     - Completaţi fişierul `xmalloc.c` cu definiţia funcţiei `xmalloc`, şi fişierul `xmalloc.h` cu macrodefiniţia `xfree` după cum urmează:
       - În cazul `xmalloc` se alocă spaţiu folosind `HeapAlloc`; trebuie să verificaţi dacă alocarea are succes sau nu
       - `xfree` este un macro care primeşte ca argument pointer-ul de eliberat; se apelează `HeapFree` şi pointer-ul este resetat la `NULL`
       - De ce este mai dificil să se realizeze o funcţie `xfree` care să realizeze aceleaşi operaţii?
2. (**1 so karma**) Program de test pentru wrapperul `xmalloc`
   - Analizaţi fişierul `test.c`.
   - Implementaţi funcţiile `tensor_alloc`, respectiv `tensor_free` care alocă/dealocă un vector tridimensional (tensor).
   - **Hints**:
     - Folosiţi funcţiile `xmalloc` şi `xfree` implementate în cadrul exerciţiului 1.
     - Urmăriţi comentariile marcate cu `TODO`.
   - Rulaţi executabilul obţinut.

BONUS Linux

1. (**1 so karma**) Realizarea unei implementări sumare a funcţiei malloc
   - Urmăriţi în `man` specificarea apelurilor brk şi sbrk
   - Folosind acest apel de sistem, completaţi implementarea funcţiei malloc din sursa `my_malloc.c`
     - Va trebui întâi să aflaţi limita curentă a heap-ului (program break), apoi să o extindeţi cu valoarea cerută pentru alocare.
   - Compilaţi şi testaţi rulând programul de test:./test
   - Hint:
     - Pentru rularea programului de test, nu uitaţi să exportaţi `LD_LIBRARY_PATH` (revedeţi secţiunea de biblioteci partajate din laboratorul 1)

Soluţii

lab04-sol.zip

Resurse utile

- Linux System Programming - Chapter 8 - Memory Management
- Windows System Programming - Chapter 5 - Memory Management (Win32 and Win64 Memory Management Architecture, Heaps, Managing Heap Memory
- Linux Application Programming - Chapter 7 - Memory Debugging Tools
- Windows Memory Management
- Memory Allocation and Paging
- Valgrind Home
- Using Valgrind to Find Memory Leaks
- The Memory Management Reference
- Using Purify
- Memory Management Software
- Smashing the Stack for Fun and Profit
- Guide to Faster, Less Frustrating Debugging
- GDB tutorial
- BUG of the month