

Dátové štruktúry a algoritmy

Dokumentácia k zadaniu

Richard Szarka

Slovenské technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

`xszarkar@stuba.sk`

ID: 111958

Zadanie: 1

Cviciači: Mgr. Martin Sabo, PhD.

Obsah

1 Zadanie	3
2 Implementácia.....	4
2.1 Makrá, globálne premenné a iné vedľajšie funkcie	5
2.2 Knižnice.....	5
3 Funkcie.....	6
3.1 Funkcia memory_init.....	6
3.2 Funkcia memory_alloc.....	7
3.3 Funkcia memory_free	9
3.4 Funkcia memory_check.....	11
4 Testovanie	12
4.1 TEST 1	12
4.2 TEST 2	13
5. Záver	15

1 Zadanie

V tomto zadaní, máme vytvoriť štyri nové funkcie na správu pamäti.

Funkcie sú nasledovné:

- `void *memory_alloc(unsigned int size);`
- `int memory_free(void *valid_ptr);`
- `int memory_check(void *ptr);`
- `void memory_init(void *ptr, unsigned int size);`

memory_alloc:

Poskytuje služby analogické štandardnému mallocu. Vstupný parameter je veľkosť požadovaného bloku pamäte. Funkcia vracia ukazovateľ na nájdený vyhradený blok pamäte. Ak taký súvislý blok sa nenašiel, funkcia vráti NULL.

memory_free:

Poskytuje služby na uvoľnenie vyhradenej pamäti (ako štandardný free()). Funkcia vráti 0, ak sa podarilo blok uvoľniť, inak vráti 1. Môžeme predpokladať, že parameter bude vždy platný ukazovateľ, vyhradený z prechádzajúcich volaní funkcie memory_alloc.

memory_check:

Funkcia memory_check slúži na skontrolovanie či ukazovateľ daný ako parameter funkcie memory_check bol ukazovateľ vrátený pri predchádzajúcich volaniach funkciou memory_alloc a zatiaľ nebol uvoľnený funkciou memory_free.

memory_init:

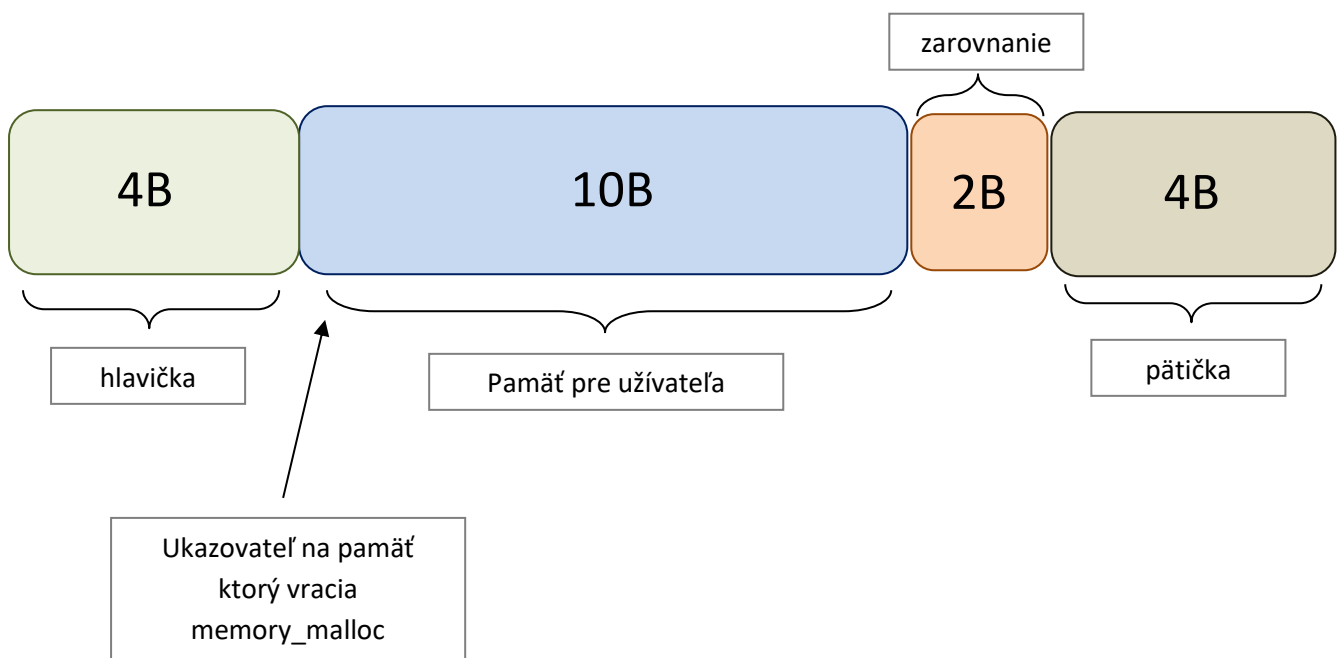
Funkcia slúži na inicializáciu voľnej pamäte. Funkcia sa zavolá práve raz pred všetkými ostatnými funkciami na začiatku. Vstupný parameter bude blok pamäte, ktorú použijeme pre organizovanie a aj pridelenie voľnej pamäte. Môžeme použiť jeden globálny ukazovateľ na začiatok pamäte, ktorá vstupuje práve do funkcie memory_init.

2 Implementácia

Metóda ktorú som si zvolil na správu alokovaných, nealokovaných blokov a samotnej pamäte je implicitná metóda správy pamäte. Bloky ktoré budem vyhradzovať užívateľovi budem zarovnávať na 4 bajty. To jest, ak si užívateľ vypýta napríklad pamäť o veľkosti 10 bajtov, program (správca – **memory_alloc**) mu vráti ukazovateľ, ktorý ukazuje na blok o veľkosti 12 bajtov (dva zvyšné bajty sú kvôli zarovnaniu). Takýmto zarovnaním zmenším externú fragmentáciu, keďže bloky nebudú mať náhodnú veľkosť. Tak isto je veľká pravdepodobnosť, že sa vyhnem zbytočným komplikáciám pri vytváraní samotného kódu a čítaní celých čísel (integer) z pamäte (pri čítaní celých čísel sa prečítajú 4 bajty – to jest int má veľkosť 4 bajty).

Budem využívať aj hlavičku a pätičku. Hlavička bude zaznamenávať veľkosť samotného bloku vrátane samej seba aj pätičky (to isté platí pre pätičku). Hlavička aj pätička budú mať vyhradené 4 bajty (spolu 8 bajtov navyše). Užívateľovi sa z funkcie vráti ukazovateľ o bunku ďalej ako je hlavička.

Ak užívateľ si vypýta veľkosť `memory_alloc(10)` :



Ako môžeme vidieť, ak používateľ si vypýta 10 bajtov, v skutočnosti táto jeho požiadavka zaberie 20 bajtov. Pri takýchto malých veľkostiach sa môže zdať, že je to „plytvanie“ pamäťou, ale v skutočnosti pri bežnej pamäti je zarovnávanie a využívanie hlavičky a pätičky výhodné. Zarovnávanie nám ušetrí viac pamäte ako spotrebuje kvôli zníženej externej fragmentácii a urýchli samotný priebeh program. Pätička nám urýchli samotnú správu pamäte.

2.1 Makrá, globálne premenné a iné vedľajšie funkcie

```
#define zarovanie 4 //zarovnanie pri alokácii pamäte pre lepšiu
fragmentáciu
#define headsize 8 //kolko bytov potrebujem na header a footer
#define hodnota_ptr(x) (*((unsigned int*)x)) //hodnota int na mieste pamati
#define one_Byte_back(x) (((unsigned int*)(x))-1) //o jeden bajt spat sa
pozri
#define one_Byte_forward(x) (((unsigned int*)(x))+1) //o jeden bajt pozri
dopredu
#define mem_SIZE 100

void * root;//pointer na začiatok mojej statickej pamäte

void vycisti_pamat(char * ptr,unsigned int size) //"vyčistí" mi pamat aby
sa v debugeri lepšie orientovalo

void pekny_vypis(char * ptr,unsigned int size) // pekný výpis na moju
kontrolu pamate

unsigned int check_4(unsigned int size)

unsigned int check_mem4(unsigned int size) //zarovna pamat aby bola
delitelna stvrokou

void test_mem(unsigned int min_blok,unsigned int max_blok, unsigned int
min_mem, unsigned int max_mem,char * memory) //testovac pamate
```

2.2 Knižnice

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
```

3 Funkcie

3.1 Funkcia `memory_init`

```
void memory_init(void *ptr , unsigned int size)
{
    //vycisti_pamat((char*)ptr,size); //pre lepsie ladenie programu
    size=size-8; //posledne 4 bity na označenie konca pamate a v prvých 4
    bytoch bude veľkosť pamate
    size=check_mem4(size);

    //nastavenie globalneho pointra na zaciatok pamate
    root=ptr;

    *(int*)root=(int)size+4; //nastaví sa veľkosť pamate

    *(int*)one_Byte_forward(root)=size; //nastaví sa hlavička prveho
    velekho bloku

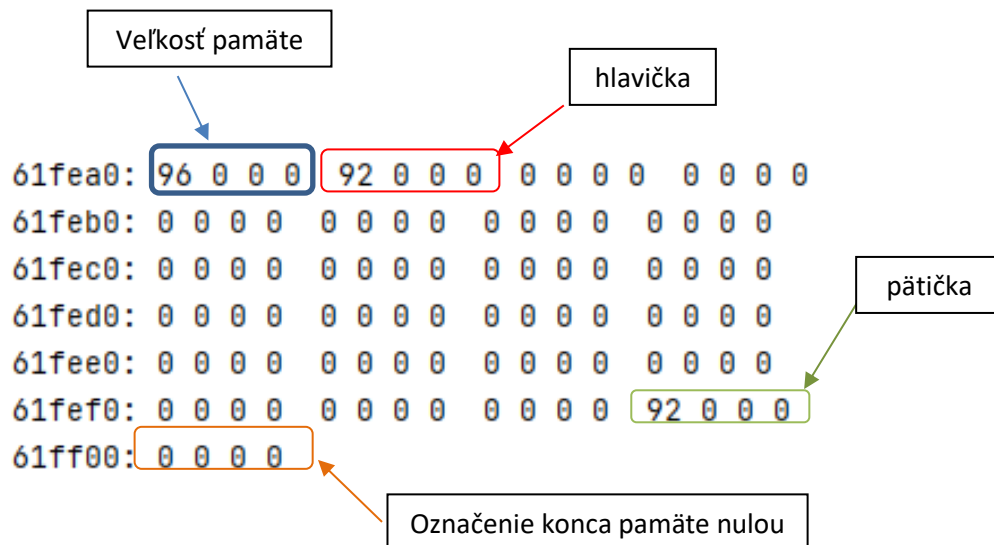
    //pridanie na koniec 1. velkeho bloku jeho veľkosť - paticka
    volatile unsigned int * footer=(unsigned int*)(root+size);
    *footer=size;
    *(footer+1)=0; //nastavenie 0 na konci pamate
}
```

Funkcia **memory_init** slúži na inicializáciu pamäte. Najprv sa zavolá funkcia **vycisti_pamat**, ktorá nastaví hodnotu všetkým pamäťovým bunkám na 0, ale program funguje správne aj bez nej (ľahšie sa bude kontrolovať alokácia a uvoľňovanie- **vycisti_pamat** môže ostať zakomentovaná). Veľkosť pamäte sa zmenší o 8 buniek, keďže nulová hodnota na konci samotnej pamäte bude slúžiť ako označenie konca pamäti a 4 bajty na začiatku pamäte využijeme na označenie veľkosti pamäte.

Následne sa zavolá funkcia **check_mem4**, ktorá zistí či veľkosť pamäte je deliteľná štvorkou. Ak nie je, zmenší veľkosť pamäte na najbližšie celé číslo, ktoré je deliteľné štvorkou (z veľkosti 97 sa stane veľkosť 96).

Nastaví sa globálny ukazovateľ, ktorý má slúžiť na zapamätania si začiatku pamäte. Vytvorí sa prvý veľký blok v pamäti s hlavičkou a pätičkou. To znamená, že o 1 bajt ďalej ako miesto kam ukazuje globálny ukazovateľ sa dá veľkosť bloku a na koniec pamäte tiež (pred označením konca). Na konci pamäte sa nastaví hodnota 0 – označenie konca pamäte. (ak je **vycisti_pamat** zakomentovaná nenastaví sa tam automaticky)

Pamäť po inicializácii o veľkosti 100 bajtov (vypísane funkciou **pekny_vypis**) :



3.2 Funkcia `memory_alloc`

Funkcia **memory_alloc** má na základe požadovanej veľkosti užívateľom nájsť blok na alokáciu. Prvé čo vo funkcii **memory_alloc** skontrolujem, je či požadovaná veľkosť je deliteľná 4. Ak nie, funkciou `chcek_4` požadovanú veľkosť upravím, aby bola deliteľná štvorkou. Algoritmus(funkciu), ktorý využijem sa nazýva **first_fit** (prvý fit).

first_fit:

```
void * first_fit(unsigned int size) //dorieš zarovnavanie o 4
{
    void * zaciatok=one_Byte_forward(root);
    void * vyhovujuci_blok=zaciatok; //nasmerovanie na blok
    unsigned int block_size=hodnota_ptr(zaciatok);

    while(((int)block_size-(int)headsize)<(int)size || ((block_size &
1)==1)) //zakomponovať nech nevyjde z pamäte
    {
        if (block_size==0)
        {
            return NULL;
        }
        block_size=block_size>>1<<1; //zbavenie sa 1 na konci
        vyhovujuci_blok+=block_size; //
        block_size=hodnota_ptr(vyhovujuci_blok);
    }

    return vyhovujuci_blok;
}
```

Do funkcii je poslaný v argumente údaj o požadovanej veľkosti od užívateľa. Funkcia funguje na princípe, že while cyklom kontroluje či dané voľné miesto v bloku je dostatočne veľké a či náhodou kontrolovaný blok už nie je vyhradený predchádzajúcim volaním funkcie **memory_alloc**. Ak nie je dostatočne veľký, alebo je obsadený, posúva sa na ďalší blok. Ak by

hodnota (veľkosť bloku) hlavičky bola rovná 0, znamená to, že sme na konci našej pamäte a žiadny blok s požadovanou veľkosťou sa v pamäti nenachádza. V takom prípade funkcia vráti hodnotu NULL. Ak nájde blok, ktorý je voľný a dostatočne veľký, funkcia vráti ukazovateľ na hlavičku daného bloku.

Ďalej vo funkcii **memory_alloc** sa nastaví ukazovateľ na pätičku nového bloku. Vypočíta sa aký zvyšok voľných pamäťových buniek ostáva po alokácii v danom bloku (ak v bloku veľkosti 40 alokujem 20 bajtov ostane mi ešte 20 voľných bajtov, ktoré viem neskôr využiť). Po vypočítaní veľkosti zvyšku sa nastaví na miestach ukazovateľov na hlavičku a pätičku veľkosť alokovaného miesta zväčšená o 1 (indikátor alokovanej pamäte). Ak ostal zvyšok po alokovaní bloku, nastaví sa jeho hlavička a pätička (označená ako voľná). Funkcia **memory_alloc** vráti ukazovateľ na bajt za posledným bajtom hlavičky

Príklady na alokovanie blokov (výpis robený funkciou pekny_vypis):

```
61fea0: 96 0 0 0 92 0 0 0 0 0 0 0 0 0 0 0
61feb0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fec0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fed0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fee0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fef0: 0 0 0 0 0 0 0 0 0 0 92 0 0 0 0
61ff00: 0 0 0 0
```

- Pamäť po inicializácii
- Jeden voľný blok o veľkosti 92

```
61fea0: 96 0 0 0 21 0 0 0 0 0 0 0 0 0 0 0
61feb0: 0 0 0 0 21 0 0 0 72 0 0 0 0 0 0 0
61fec0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fed0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fee0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fef0: 0 0 0 0 0 0 0 0 0 0 72 0 0 0 0
61ff00: 0 0 0 0
```

- Zavolaná funkcia s **memory_alloc** s argumentom 10
- Jeden alokovaný blok o veľkosti 20 a voľných 72

```
61fea0: 96 0 0 0 21 0 0 0 0 0 0 0 0 0 0 0
61feb0: 0 0 0 0 21 0 0 0 21 0 0 0 0 0 0 0
61fec0: 0 0 0 0 0 0 0 0 21 0 0 0 52 0 0 0
61fed0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fee0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fef0: 0 0 0 0 0 0 0 0 0 0 52 0 0 0 0
61ff00: 0 0 0 0
```

- Druhý krát zavolaná funkcia s **memory_alloc** s argumentom 10
- Dva alokované blok o veľkosti 20 a voľných 52

Alokovaný blok
Voľný blok

3.3 Funkcia `memory_free`

Uvoľňovanie alokovaného bloku v tejto konkrétnej implementácii je pomerne triviálna záležitosť. Užívateľ argumentom funkcie **`memory_free`** poskytne ukazovateľ, ktorý chce uvoľniť. **`Memory_alloc`** vracia ukazovateľ, ktorý ukazuje na pamäťový priestor za hlavičkou (služi na správu pamäte), makrom **`one_byte_back`** sa nastaví ukazovateľ na hlavičku daného bloku. V hlavičke sa následne deaktivuje posledný bit (označí sa, že je už voľný).

- Pamäť je zaplnená
- Všetky hlavičky a pätičky majú posledný bit aktívny

```
61fea0: 96 0 0 0 21 0 0 0 0 0 0 0 0 0 0 0
61feb0: 0 0 0 0 21 0 0 0 21 0 0 0 0 0 0 0
61fec0: 0 0 0 0 0 0 0 0 21 0 0 0 21 0 0 0
61fed0: 0 0 0 0 0 0 0 0 0 0 0 0 21 0 0 0
61fee0: 21 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fef0: 21 0 0 0 13 0 0 0 0 0 0 0 13 0 0 0
61ff00: 0 0 0 0
```

```
61fea0: 96 0 0 0 20 0 0 0 0 0 0 0 0 0 0 0
61feb0: 0 0 0 0 20 0 0 0 21 0 0 0 0 0 0 0
61fec0: 0 0 0 0 0 0 0 0 21 0 0 0 21 0 0 0
61fed0: 0 0 0 0 0 0 0 0 0 0 0 0 21 0 0 0
61fee0: 21 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fef0: 21 0 0 0 13 0 0 0 0 0 0 0 13 0 0 0
61ff00: 0 0 0 0
```

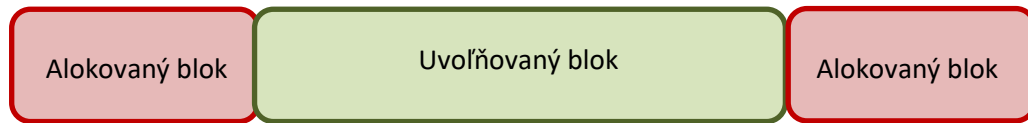
- Požiadavka na uvoľnenie bloku v pamäti (1. blok)
- Posledný bit bloku sa nastavil na 0 (blok je už voľný)

Spájanie uvoľnených blokov vo funkcii `memory_free`

Dôležitá vec na ktorú sa musí myslieť pri uvoľňovaní blokov je spájanie voľných blokov. To jest, ak užívateľ zadal požiadavku na uvoľnenie bloku, ktorý má veľkosť 20 bajtov a vedľa neho v pamäti je ďalší voľný blok o veľkosti 20 bajtov, treba aby ich program spojil do jedného bloku o veľkosti 40 bajtov. Ak by tak nespravil, a užívateľ požiadala neskôr o blok veľkosti 30 bajtov, program by nevedel nájsť dostatok voľnej pamäte (aj napriek tomu že existuje 40 voľných bajtov, ktoré sú pri sebe). Či je blok pred a za uvoľňovaným voľný zistíme podľa pätičky predchádzajúceho (makro **`one_byte_back`**) a hlavičky nasledujúceho bloku. Veľkosť nových spojených blokov zistíme sčítaním veľkostí spájaných. Pri uvoľňovaní pamäte nám vzniknú teda 4 prípady uvoľňovania:

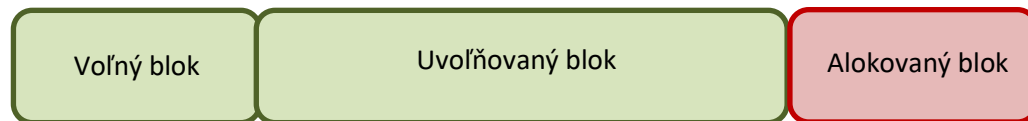
1. Blok pred uvoľňovaným a za uvoľňovaným je alokovaný
2. Blok pred uvoľňovaným je voľný a za uvoľňovaným je alokovaný
3. Blok pred uvoľňovaným je voľný a za uvoľňovaným je alokovaný
4. Blok pred uvoľňovaným a za uvoľňovaným je voľný

Prípád 1



V prípade jedna nám stačí nastaviť posledný bit na hlavičke a pätičke uvoľňovanému bloku na 0. Nie je ho s čím spájať, keďže blok pred aj za nie je voľný.

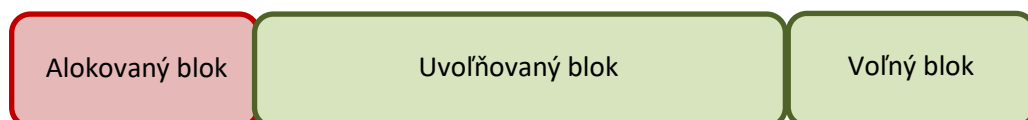
Prípád 2



V prípade dva treba nastaviť hlavičku predchádzajúceho bloku ako hlavičku nového bloku. Pätička predchádzajúceho bloku bude pätička nového bloku. Hlavičku uvoľňovaného a pätičku predchádzajúceho nastavíme na hodnotu 0 pre lepšiu čitateľnosť v pamäti pri ladení programu. Posledný bit hlavičky predchádzajúceho bloku a pätičky uvoľňovaného bloku nastavíme na znak „voľný“ (posledný bit=0).

Tento istý prípad sa dá použiť aj vtedy, keď uvoľňovaný blok je na konci pamäte a predchádzajúci blok je voľný.

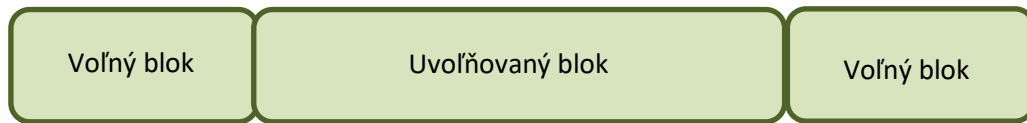
Prípád 3



V prípade tri treba nastaviť hlavičku uvoľňovaného bloku ako hlavičku nového bloku. Pätička nasledujúceho bloku bude pätička nového bloku. Hlavičku nasledujúceho a pätičku uvoľňovaného bloku nastavíme na hodnotu 0 pre lepšiu čitateľnosť v pamäti pri ladení. Posledný bit hlavičky uvoľňovaného bloku a pätičky nasledujúceho bloku nastavíme na znak „voľný“ (posledný bit=0).

Tento istý prípad sa dá použiť aj vtedy, keď je uvoľňovaný blok je na začiatku pamäte (1. blok v pamäti) a nasledujúci blok je voľný.

Prípád 4



V prípade 4 treba nastaviť hlavičku predchádzajúceho bloku na ako hlavičku nového bloku. Pätičku nasledujúceho bloku nastavíme ako pätičku nového veľkého bloku. Pätičku predchádzajúceho bloku, pätičku uvoľňovaného bloku, hlavičku uvoľňovaného bloku a hlavičku nasledujúceho bloku nastavíme na 0 kvôli lepšej čitateľnosti pri ladení a testovaní. Posledný bit pri hlavičke a pätičke nového bloku musia byť nastavené na hodnotu 0 (znak „voľný“).

3.4 Funkcia `memory_check`

```
int memory_check(void *ptr)
{
    if(ptr) //ak pointer je vobec platny
    {
        unsigned int *pointer = one_byte_back((unsigned int *) ptr);
//pozri sa na hlavičku
        unsigned int *koniec = (unsigned int*)root+(*(unsigned int*)root);
// zistí kde je koniec pamäte
        if(pointer>(unsigned int*)root && pointer<koniec && ((*pointer & 1)
== 1)) //ak je pointer alokovany a je medzi zaciatkom a koncom pamäte
        {
            return 1;
        }
        return 0; //ak nie
    }

    return 0;
}
```

Vo funkcii **memory_check** sa má zistiť či ukazovateľ `ptr`, ktorý do funkcie vojde ako parameter bol vrátený funkciou **memory_alloc** v predchádzajúcom volaní a ešte nebol uvoľnený funkciou **memory_free**.

Najprv sa zistí či daný ukazovateľ nie je NULL. Ak nie inicializuje sa ukazovateľ `pointer`, ktorý ukazuje o jeden bajt späť ako ukazovateľ `ptr`. Teda ukazuje na hlavičku bloku. Inicializujem ukazovateľ `koniec`, ktorý bude ukazovať na koniec pamäte. Ak ukazovateľ `ptr` ukazuje na adresu medzi začiatkom pamäte (`root`) a koncom pamäte(`koniec`) a zároveň je alokovaný (spodný bit hodnoty hlavičky má hodnotu 1), ukazovateľ `ptr` je ukazovateľ, ktorý bol vrátený funkciou **memory_alloc** a zatiaľ nebol uvoľnený.


4 Testovanie

4.1 TEST 1

Test číslo 1 bude zámerné volanie funkcií **memory_alloc** a **memory_free** v takom poradí aby som otestoval spájanie voľných blokov.


```
61fea0: 96 0 0 0 21 0 0 0 0 0 0 0 0 0 0 0
61feb0: 0 0 0 0 21 0 0 0 21 0 0 0 0 0 0 0
61fec0: 0 0 0 0 0 0 0 0 21 0 0 0 21 0 0 0
61fed0: 0 0 0 0 0 0 0 0 0 0 0 0 21 0 0 0
61fee0: 21 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fef0: 21 0 0 0 13 0 0 0 0 0 0 0 13 0 0 0
61ff00: 0 0 0 0
```

- pamäť zaplnená blokmi




```
61fea0: 96 0 0 0 20 0 0 0 0 0 0 0 0 0 0 0
61feb0: 0 0 0 0 20 0 0 0 21 0 0 0 0 0 0 0
61fec0: 0 0 0 0 0 0 0 0 21 0 0 0 21 0 0 0
61fed0: 0 0 0 0 0 0 0 0 0 0 0 0 21 0 0 0
61fee0: 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fef0: 20 0 0 0 13 0 0 0 0 0 0 0 13 0 0 0
61ff00: 0 0 0 0
```

- dva uvoľnené bloky funkciou **memory_free** (prípád 1)



```
61fea0: 96 0 0 0 40 0 0 0 0 0 0 0 0 0 0 0
61feb0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fec0: 0 0 0 0 0 0 0 0 40 0 0 0 21 0 0 0
61fed0: 0 0 0 0 0 0 0 0 0 0 0 0 21 0 0 0
61fee0: 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fef0: 20 0 0 0 13 0 0 0 0 0 0 0 13 0 0 0
61ff00: 0 0 0 0
```

- uvoľnený blok (otestované spájanie voľných blokov prípad 2)



```
61fea0: 96 0 0 0 80 0 0 0 0 0 0 0 0 0 0 0
61feb0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fec0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fed0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fee0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fef0: 80 0 0 0 13 0 0 0 0 0 0 0 13 0 0 0
61ff00: 0 0 0 0
```

- Uvoľnený blok (otestované spájanie voľných blokov prípad 4)

```
61fea0: 96 0 0 0 41 0 0 0 0 0 0 0 0 0 0 0
61feb0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fec0: 0 0 0 0 0 0 0 0 41 0 0 0 40 0 0 0
61fed0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fee0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fef0: 40 0 0 0 13 0 0 0 0 0 0 0 13 0 0 0
61ff00: 0 0 0 0
```

- Alokovaný blok o veľkosti 40 bajtov
- Za ním voľný blok o veľkosti 40 bajtov

↓

```
61fea0: 96 0 0 0 80 0 0 0 0 0 0 0 0 0 0 0
61feb0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fec0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fed0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fee0: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
61fef0: 80 0 0 0 13 0 0 0 0 0 0 0 13 0 0 0
61ff00: 0 0 0 0
```

- Uvoľnený blok (otestované spájanie voľných blokov prípad 3)

Celkový uvoľnený blok (aj po spájaní)
Konkrétny blok, ktorý bol uvoľňovaný

4.2 TEST 2

Na test bude použitá funkcia **test_mem**.

- pridelovanie rovnakých blokov malej veľkosti (veľkosti 8 až 24 bajtov) pri použití malých celkových blokov pre správcu pamäte (od 25 do 200 bajtov)

Veľkosť blokov: 16 (priemer z 8 a 24)

Veľkosť pamäti: 25-200 bajtov



- pridelovanie nerovnakých blokov malej veľkosti (náhodné veľkosti 8 až 24 bajtov) pri použití malých celkových blokov pre správcu pamäte (od 25 do 200 bajtov),

Veľkosť blokov: 8 až 24

Veľkosť pamäti: 25-200 bajtov



- pridelovanie nerovnakých blokov väčšej veľkosti (veľkosti 500 až 5000 bajtov) pri použití väčších celkových blokov pre správcu pamäte (aspoň veľkosti 1000 až 100000 bajtov),

Veľkosť blokov: 500 až 5000

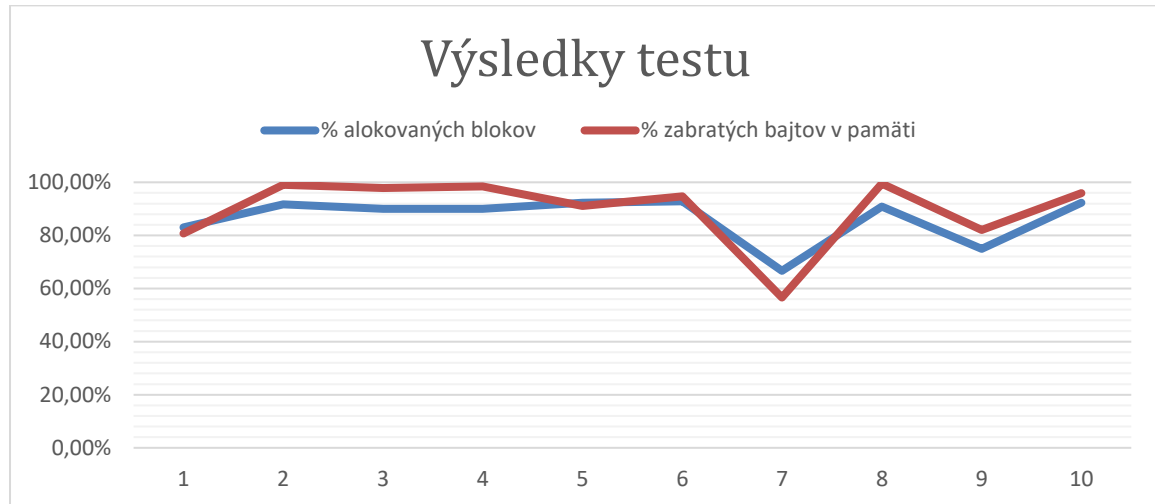
Veľkosť pamäti: 1000-100000 bajtov



- pridelovanie nerovnakých blokov malých a veľkých veľkostí (veľkosti od 8 bajtov do 50 000) pri použití väčších celkových blokov pre správcu pamäte (od 1000 do 100000 bajtov).

Veľkosť blokov: 8 až 50000

Veľkosť pamäti: 1000-100000 bajtov



5. Záver

Moje riešenie považujem za správne na koľko pri viacnásobnej opakovanej alokácii neprišlo k žiadnej chybe. V niektorých konkrétnych scenároch testu sa alokovalo viacnásobne aj vyše 1000 blokov. Každý z nich sa alokoval a uvoľnil bez problémov. Myslel som na všetky chyby a výnimky, ktoré by mohli nastať a následne ich ošetril. Spájanie voľných blokov znižuje externú fragmentáciu a zvyšuje pamäťovú efektívnosť.

Časová a pamäťová efektívnosť funkcií je nasledovná:

- `memory_init` – časová $O(1)$ (**s vycisti_pamat** – pre lepšie ladenie $O(n)$),
- pamäťová $O(1)$
- `memory_alloc` – časová $O(n)$, pamäťová $O(1)$ (n – veľkosť samotnej pamäte)
- `memory_free` – časová $O(1)$, pamäťová $O(1)$
- `memory_check` – časová $O(1)$, pamäťová $O(1)$

Zlepšenie programu by bolo možné iným algoritmom na hľadanie voľného bloku, napríklad algoritmom best fit. First fit je časovo efektívnejší kým pamäť nie je plná, avšak po zaplnení je časová efektívnosť takmer rovná best fitu. Best fit má na rozdiel od first fitu nižšiu externú fragmentáciu.