

FIIT STU

DSA - Dátové štruktúry a algoritmy

Zadanie 1 - Správca pamäti

Meno a Priezvisko: **Michal Magula**

AIS ID: **110845**

Akademický rok: **2020/2021**

Znenie zadania

V štandardnej knižnici jazyka C sú pre alokáciu a uvoľnenie pamäti k dispozícii funkcie `malloc` a `free`. V tomto zadaní je úlohou implementovať vlastnú verziu alokácie pamäti.

Konkrétnejšie je vašou úlohou je implementovať v programovacom jazyku C nasledovné ŠTYRI funkcie:

- `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);`

Vo vlastnej implementácii môžete definovať aj iné pomocné funkcie ako vyššie spomenuté, nesmiete však použiť existujúce funkcie `malloc` a `free`.

Funkcia **`memory_alloc`** má poskytovať služby analogické štandardnému `malloc`. Teda, vstupné parametre sú veľkosť požadovaného súvislého bloku pamäte a funkcia mu vráti: ukazovateľ na úspešne alokovaný kus voľnej pamäte, ktorý sa vyhradil, alebo `NULL`, keď nie je možné súvislú pamäť požadovanej veľkosti vyhraďiť.

Funkcia **`memory_free`** slúži na uvoľnenie vyhradeného bloku pamäti, podobne ako funkcia `free`. Funkcia vráti 0, ak sa podarilo (funkcia zbehla úspešne) uvoľniť blok pamäti, inak vráti 1. Môžete predpokladať, že parameter bude vždy platný ukazovateľ, vrátený z predchádzajúcich volaní funkcie `memory_alloc`, ktorý ešte nebol uvoľnený.

Funkcia **`memory_check`** slúži na skontrolovanie, či parameter (ukazovateľ) je platný ukazovateľ, ktorý bol v nejakom z predchádzajúcich volaní vrátený funkciou `memory_alloc` a zatiaľ nebol uvoľnený funkciou `memory_free`. Funkcia vráti 0, ak je ukazovateľ neplatný, inak vráti 1.

Funkcia `memory_init` slúži na inicializáciu spravovanej voľnej pamäte. Predpokladajte, že funkcia sa volá práve raz pred všetkými inými volaniami `memory_alloc`, **`memory_free`** a **`memory_check`**. Viď testovanie nižšie. Ako vstupný parameter funkcie príde blok pamäte, ktorú môžete použiť pre organizovanie a aj pridelenie voľnej pamäte. Vaše funkcie nemôžu používať globálne premenné okrem jednej globálnej premennej na zapamätanie ukazovateľa na pamäť, ktorá vstupuje do funkcie `memory_init`. Ukazovatele, ktoré prideliť vaše funkcia `memory_alloc` musia byť výhradne z bloku pamäte, ktorá bola pridelená funkcii `memory_init`.

Okrem implementácie samotných funkcií na správu pamäte je potrebné vaše riešenie dôkladne otestovať. Vaše riešenie musí byť 100% korektné. Teda pokiaľ pridelite pamäť funkciou `memory_alloc`, mala by byť dostupná pre program (nemala

by presahovať pôvodný blok, ani prekryvať doteraz pridelenú pamäť) a mali by ste ju úspešne vedieť uvoľniť funkciou **memory_free**. Riešenie, ktoré nespĺňa tieto minimálne požiadavky je hodnotené 0 bodmi. Testovanie implementujte vo funkcii main a výsledky testovania dôkladne zdokumentuje. Zamerajte sa na nasledujúce scenáre:

- pridelenie rovnakých blokov malej veľkosti (veľkosti 8 až 24 bytov) pri použití malých celkových blokov pre správcu pamäte (do 50 bytov, do 100 bytov, do 200 bytov),
- pridelenie nerovnakých blokov malej veľkosti (náhodné veľkosti 8 až 24 bytov) pri použití malých celkových blokov pre správcu pamäte (do 50 bytov, do 100 bytov, do 200 bytov),
- pridelenie nerovnakých blokov väčšej veľkosti (veľkosti 500 až 5000 bytov) pri použití väčších celkových blokov pre správcu pamäte (aspoň veľkosti 1000 bytov),
- pridelenie nerovnakých blokov malých a veľkých veľkostí (veľkosti od 8 bytov do 50 000) pri použití väčších celkových blokov pre správcu pamäte (aspoň veľkosti 1000 bytov).

V testovacích scenároch okrem iného vyhodnoťte koľko % blokov sa vám podarilo alokovať oproti ideálnemu riešeniu (bez vnútornej aj vonkajšej fragmentácie). Teda snažte sa pridelať bloky až dovtedy, kým v ideálnom riešení nebude veľkosť voľnej pamäte menšia ako najmenší možný blok podľa scenára.

Príklad jednoduchého testu:

```
#include <string.h>
int main()
{
    char region[50]; //celkový blok pamäte o veľkosti 50 bytov
    memory_init(region, 50);
    char* pointer = (char*) memory_alloc(10); //alokovaný blok o veľkosti 10 bytov
    if (pointer)
        memset(pointer, 0, 10);
    if (pointer)
        memory_free(pointer);
    return 0;
}
```

Riešenie zadania sa odovzdáva do miesta odovzdania v AIS do stanoveného termínu (oneskorené odovzdanie je prípustné len vo vážnych prípadoch, ako napr. choroba, o možnosti odovzdať zadanie oneskorene rozhodne cvičiaci, príp. aj o bodovej penalizácii). Odovzdáva sa jeden zip archív, ktorý obsahuje jeden zdrojový súbor s implementáciou a jeden súbor s dokumentáciou vo formáte pdf.

Pri implementácii zachovávajte určité konvencie písania prehľadných programov (pri odovzdávaní povedzte cvičiacemu, aké konvencie ste pri písaní kódu dodržiavali) a

zdrojový kód dôkladne okomentujte. Snažte sa, aby to bolo na prvý pohľad pochopiteľné.

Dokumentácia musí obsahovať hlavičku (kto, aké zadanie odovzdáva), stručný opis použitého algoritmu s názornými nákresemi/obrázkami a krátkymi ukážkami zdrojového kódu, vyberajte len kód, na ktorý chcete extra upozorniť. Pri opise sa snažte dbať osobitný dôraz na zdôvodnenie správnosti vášho riešenia – teda dôvody prečo je dobré/správne, spôsob a vyhodnotenie testovania riešenia. Nakoniec musí technická dokumentácia obsahovať odhad výpočtovej (časovej) a priestorovej (pamäťovej) zložitosti vášho algoritmu. Celkovo musí byť cvičiacemu jasné, že viete čo ste spravili, že viete odôvodniť, že to je správne riešenie, a viete aké je to efektívne.

Hodnotenie

Môžete získať 15 bodov, minimálna požiadavka 6 bodov.

Jedno zaujímavé vylepšenie štandardného algoritmu je prispôbiť dĺžku (počet bytov) hlavičky bloku podľa jeho veľkosti. Každé funkčné vylepšenie cvičiaci zohľadní pri bodovaní.

Cvičiaci prideluje body podľa kvality vypracovania. 8 bodov môžete získať za vlastný funkčný program pridelovania pamäti (aspoň základnú funkčnosť musí študent preukázať, inak 0 bodov; metóda implicitných zoznamov najviac 4 body, metóda explicitných zoznamov bez zoznamov blokov voľnej pamäti podľa veľkosti najviac 6 bodov), 3 body za dokumentáciu (bez funkčnej implementácie 0 bodov), 4 body môžete získať za testovanie (treba podrobne uviesť aj v dokumentácii). Body sú ovplyvnené aj prezentáciou cvičiacemu (napr. keď neviete reagovať na otázky vzniká podozrenie, že to nie je vaša práca, a teda je hodnotená 0 bodov).

Doplnenie zadania

Dokumentácia musí obsahovať:

1. titulná strana
 - a. názov inštitúcie,
 - b. názov predmetu,
 - c. názov zadania,
 - d. meno a priezvisko,
 - e. ais id,
 - f. akademický rok
 - g. na každej strane v hlavičke: meno a priezvisko, ais id; v päte: názov zadania a číslovanie strán
2. znenie zadania

- a. to ktoré bolo vložené do AIS
 - b. toto doplnenie
3. stručný opis algoritmu
 - použite 2 spôsoby opisu
4. doplňte ukážky zdrojového kódu na ktorý chcete extra upozorniť
 - a. použite 2 spôsoby opisu
 - b. doplňte ukážky zdrojového kódu na ktorý chcete extra upozorniť
5. testovanie
 - a. scenár 1
 - i. 8 do 50/100/200
 - ii. 15 do 50/100/200
 - iii. 24 do 50/100/200
 2. scenár 2
 - i. rand (8-24) do 50/100/200 – zápis 5 hodnôt cyklicky do naplnenia pamäte
 3. scenár 3
 - i. rand (500 – 5000) do 10 000 – zápis 5 hodnôt cyklicky do naplnenia pamäte
 4. scenár 4
 - i. rand (8 – 50 000) do 100 000 – zápis 5 hodnôt cyklicky do naplnenia pamäte
 5. teoreticky výpočet alokácie vs. reálna hodnota
 - i. pri výpočte neberte do úvahy vnútornú a vonkajšiu fragmentáciu
 - ii. reálna je to čo alokujete
 - iii. vyhodnoťte percentuálne
 - i. nezabudnite na časovú zložitosť

Zdrojový kód:

1. memory_alloc
 - a. blok spolu s réžiu dorovnať na najbližší vyšší násobok čísla 2
 - b. blok/y na konci pamäte do ktorých nie je možné nič alokovať pripojiť k vedľajšej alokovanej časti
2. memory_free
 - a. spájanie voľných blokov tak ako bolo uvedené na prednáške

Stručný opis algoritmu

Pre moju implementáciu som sa rozhodol použiť oddelené zoznamy blokov voľnej pamäti (angl. **segregated free list**). Je to druh explicitného zoznamu, ktorý je rozšírený o to, že voľné bloky sú rozdelené do samostatných zoznamov podľa veľkosti. Explicitný zoznam je taký zoznam, kde sú pospájané do zoznamu iba voľní bloky pamäte. Vďaka tomu je pridelenie blokov rýchlejšie ako u implicitných zoznamov, kde sú prepojené všetky (voľné aj alokované) bloky navzájom. Kvôli tomuto prístupu na hľadanie voľných blokov bol využitý algoritmus najlepší-vhodný (angl. **best-fit**). Tento algoritmus vyberá najvhodnejšiu veľkosť voľného bloku tak, aby sa predišlo deleniu príliš veľkých blokov na menšie a tým pádom by mohol nastať scenár, že by nebolo dostatok súvislého miesta na alokovanie veľkého bloku.

Dizajn pamäťových blokov

Na reprezentáciu blokov používam hlavičky (ang. **header**) a pätičky (angl. **footer**), v ktorých je uložená veľkosť bloku bez veľkosti hlavičky a pätičky. Voľný blok pamäte je reprezentovaný tak, že veľkosť bloku je kladná a alokovaný blok pamäte je reprezentovaný zápornou hodnotou (veľkosť bloku * (-1)). Na zistenie polohy ďalších voľných blokov som si ukladal do obsahu voľných blokov hodnotu posunutia od začiatku pamäte (angl. **offset**) na ďalší a predošlý blok. V už alokovaných blokoch nie je potrebné ukladať žiadne posunutia. Voľné bloky sú ešte navyše rozdelené do zoznamov podľa veľkosti. Zoznamy sú veľkostí mocnín čísla 2, začínajúce 3 mocninou (2^3) a horným ohraničením veľkosti je nasledujúca mocnina čísla dva -1.

Hlavička	Obsah pamäte	Pätička
Veľkosť: 4 byty	Min. veľkosť: 8 bytov (ďalší a predošlý blok vo voľných blokoch)	Veľkosť: 4 byty

V dôsledku tohto dizajnu bloku je veľkosť najmenšieho možného bloku pamäte **16 B**.

Opis funkcií

Pomocné funkcie

int get_list_offset (int size) - O(n)

Funkcia vracia posunutie (od začiatku pamäte) zoznamu pre určitú veľkosť. Veľkosť je zadaná do funkcie ako argument.

int find_free_block(u_int size) - O(n)

Funkcia vracia posunutie (od začiatku pamäte) na nájdený voľný blok, ktorý je väčší alebo rovnaký ako argument size (veľkosť ktorú užívateľ zadá, že chce alokovať). Funkcia najskôr prechádza zoznamami podľa veľkosti, ak sa v tom zozname niečo nachádza, tak skontroluje či je veľkosť dostatočná. Ak veľkosť nie je dostatočná, tak funkcia sa prechádza zoznamom tej veľkosti (ak existuje). Ak nájde vyhovujúci zoznam, tak vráti posunutie. Ak nenájde, tak sa posúva do nasledujúceho zoznamu pre väčšie voľné bloky a ten prechádza rovnakým štýlom. V prípade, že taký blok neexistuje, funkcia vráti 0.

int get_memory_offset(void *ptr) - O(n)

Funkcia prijíma ako argument ukazovateľ na miesto, ktorého posunutie od začiatku je potrebné zistiť. Je očakávané, že táto adresa sa nachádza v pamäti. Funkcia vracia posunutie od začiatku pamäte.

void case1(void *ptr) - O(n)

Funkcia slúži na uvoľnenie pamäte v scenári, že uvoľňovaná pamäť sa nachádza medzi alokovanými blokmi.

Alokovaný blok	Uvoľňovaný blok	Alokovaný blok
----------------	------------------------	----------------

void case2(void *ptr) - O(n)

Funkcia slúži na uvoľnenie pamäte v scenári:

Alokovaný blok	Uvoľňovaný blok	Uvoľnený blok
----------------	------------------------	---------------

Uvoľňovaný blok sa spojí s uvoľneným blokom, a následne sa uloží na začiatok zoznamu adekvátnej veľkosti.

void case3(void *ptr) - O(n)

Funkcia slúži na uvoľnenie pamäte v scenári:

Uvoľnený blok	Uvoľňovaný blok	Alokovaný blok
---------------	------------------------	----------------

Uvoľňovaný blok sa spojí s uvoľneným blokom, a následne sa uloží na začiatok zoznamu adekvátnej veľkosti.

void case4(void *ptr) - O(n)

Funkcia slúži na uvoľnenie pamäte v scenári:

Uvoľnený blok	Uvoľňovaný blok	Uvoľnený blok
---------------	------------------------	---------------

Uvoľňovaný blok sa uvoľní, následne prebehne spojenie s ľavým blokom a nakoniec sa spojí s pravým blokom. Počas týchto spájání sa zoznamy preusporiadajú tak, aby sa nestratili hodnoty posunutia na žiaden voľný blok. Na záver sa voľný blok uloží na začiatok zoznamu vhodnej veľkosti.

Hlavné funkcie

Funkcia **void memory_init(void *ptr, u_int size) (O(n))** berie ako vstupné argumenty ukazovateľ (angl. **pointer**) na pole na ktorom sa bude simulovať dynamická správa pamäte a veľkosť tohto pola. V prípade, že vstupné pole je ukazovateľ na NULL alebo veľkosť menšia alebo rovná nule, tak vypíše chybovú hlášku a program skončí. Ak takýto prípad nenastane, tak funkcia pole vynuluje, nastaví globálny ukazovateľ na začiatok pola, nastaví hlavičku pamäte, v ktorej sa nachádza veľkosť pamäte - veľkosť hlavičky. Ďalej vyhradí miesto pre zoznamy voľných blokov a vytvorí prvý voľný blok, ktorý priradí do adekvátneho zoznamu.

Funkcia **void *memory_alloc(u_int size) (O(n))** očakáva ako vstupný argument veľkosť miesta, ktoré užívateľ chce alokovať. Pomocou pomocnej funkcie **find_free_block** nájde voľný blok. Ak sa takýto blok v pamäti nenachádza, tak funkcia **memory_alloc** vráti **NULL**. Ak bol takýto blok nájdený, funkcia skontroluje, či je takýto voľný blok možné rozdeliť, ak áno, vytvorí nový voľný blok, umiestni ho do príslušného zoznamu a vráti ukazovateľ na začiatok užívateľského miesta.

Alokácia bloku prebieha nasledovne: Každý byte v bloku je nastavený na hodnotu 1, veľkosť v hlavičke a pätičke je vynásobená hodnotou -1, aby sa z kladného čísla stalo záporné. Podľa tohto príznaku je možné neskôr v iných funkciách určiť, či je blok voľný alebo alokovaný.

Funkcia **int memory_check(void *ptr) (O(n))** zisťuje, či ukazovateľ, ktorý vstupuje do funkcie je validný ukazovateľ na alokovaný blok. Najskôr funkcia otestuje, či vstupný ukazovateľ sa nachádza v rozsahu pola, následne funkcia prechádza po blokoch a porovnáva vstupný ukazovateľ s ukazovateľmi na alokované bloky. Ak sa tieto bloky rovnajú, funkcia vracia hodnotu 1, inak vracia hodnotu 0.

Funkcia **int memory_free(void *ptr) (O(n))** slúži na uvoľňovanie alokovaných blokov. Vstupným argumentom je validný ukazovateľ na alokovanú pamäť. Úlohou je porovnávať, na akej pozícii vzhľadom na iné bloky sa uvoľňovaný blok nachádza. Po

vyhodnotení pozície sa volá jedna z pomocných funkcií (**case1**, **case2**, **case3**, **case4**) na následné uvoľnenie pamäte. Tieto pomocné funkcie boli navrhnuté tak, aby sa dali použiť aj v okrajových scenároch na okrajoch pamäte. Funkcia vracia hodnotu 0 v prípade správne uvoľnenej pamäte. Keďže sa predpokladá, že vstupný ukazovateľ je vždy validný, tak funkcia nepotrebuje riešiť iné scenáre, t.j. nie je potrebné vracať inú hodnotu.

Testovanie

Scenár 1a

V testovacom scenári 1 sú vyvolané funkcie vopred napísané a testujú sa rôzne možnosti alokácie a uvoľňovania pamäti na pamäti o veľkostiach 50, 100, 200 bytov.

```
char region1[50];
memory_init(region1, 50);
char *p0 = memory_alloc(8);
char *p1 = memory_alloc(8);
memory_free(p0); //freeing block scenario notMemory/allocated/allocated
if (!memory_check(p0))
{
    printf("Not allocated block\n");
}
char *p2 = memory_alloc(8); //this will allocate on the same spot as p0
used to be
char *p3 = memory_alloc(8);
if (p3 == NULL)
{
    printf("Memory is full.\n");
}
```

V tomto konkrétnom prípade (1a pamäť - 50 bytov) je testované alokácia blokov o veľkosti 8 bytov, uvoľňovanie na okraji pamäte, následná kontrola ukazovateľa, či bola pamäť správne uvoľnená. Na koniec sa otestuje, prípade, keď sa užívateľ snaží alokovať viac pamäte ako je dostupnej. V takomto prípade tester otestuje ukazovateľ a vypíše hlášku "Memory is full.", čo v preklade znamená "Pamäť je plná."

```
char region2[100];
memory_init(region2, 100);
char *p4 = memory_alloc(8);
char *p5 = memory_alloc(8);
char *p6 = memory_alloc(8);
char *p7 = memory_alloc(8);
memory_free(p4);
memory_free(p6);
memory_free(p5); //freeing block where the block is in the middle
```

```

free/allocated/free
char *p8 = memory_alloc(8);
char *p9 = memory_alloc(8);
char *p10 = memory_alloc(8);

```

Alokácia 8 B blokov na 100 B pamäti. Testovanie alokácie štyroch blokov, následné uvoľňovanie dvoch blokov ,aby nastal scenár, kedy sa **uvoľňovaný blok** nachádza medzi **dvoma voľnými blokmi**. Toto je test na správne spájanie voľných blokov.

0x61fb60:	96	0	0	0	56	0	0	0
0x61fb68:	0	0	0	0	0	0	0	0
0x61fb70:	0	0	0	0	8	0	0	0
0x61fb78:	88	0	0	0	56	0	0	0
0x61fb80:	8	0	0	0	-8	-1	-1	-1
0x61fb88:	1	1	1	1	1	1	1	1
0x61fb90:	-8	-1	-1	-1	8	0	0	0
0x61fb98:	24	0	0	0	0	0	0	0
0x61fba0:	8	0	0	0	-8	-1	-1	-1
0x61fba8:	1	1	1	1	1	1	1	1
0x61fbb0:	-8	-1	-1	-1	8	0	0	0
0x61fbb8:	0	0	0	0	24	0	0	0
0x61fbc0:	8	0	0	0				

Pamäť po uvoľnení:

0x61fb60:	96	0	0	0	88	0	0	0
0x61fb68:	0	0	0	0	24	0	0	0
0x61fb70:	0	0	0	0	40	0	0	0
0x61fb78:	0	0	0	0	0	0	0	0
0x61fb80:	0	0	0	0	0	0	0	0
0x61fb88:	0	0	0	0	0	0	0	0
0x61fb90:	0	0	0	0	0	0	0	0
0x61fb98:	0	0	0	0	0	0	0	0
0x61fba0:	40	0	0	0	-8	-1	-1	-1
0x61fba8:	1	1	1	1	1	1	1	1
0x61fbb0:	-8	-1	-1	-1	8	0	0	0
0x61fbb8:	0	0	0	0	0	0	0	0
0x61fbc0:	8	0	0	0				

```

char region3[200];
memory_init(region3, 200);
char *p11 = memory_alloc(8);
char *p12 = memory_alloc(8);
char *p13 = memory_alloc(8);

```

```

char *p14 = memory_alloc(8);
char *p15 = memory_alloc(8);
char *p16 = memory_alloc(8);
char *p17 = memory_alloc(8);
char *p18 = memory_alloc(8);
char *p19 = memory_alloc(8);
memory_free(p12);
memory_free(p14);
memory_free(p16);
memory_free(p18);

```

Tento prípad je zameraný na testovanie vytvárania spájaného zoznamu, kde voľné bloky sú z rovnakého zoznamu. Na tento test bola využitá pamäť o veľkosti 200 B a alokované bloky čo najmenšej veľkosti - 8 B. Najskôr bolo alokovaných veľa blokov a následne bol uvoľnený každý druhý blok. Bloky boli uvoľnené od najmenšieho poradového čísla ukazovateľa po najväčší. To znamená, že zoznam, ktorý je pre veľkosti voľných blokov 8 B - 15B bude mať uložený v sebe posunutie pre posledný uvoľnený blok. Prvý uvoľnený blok bude na konci tohto zoznamu.

-exec x/200b heap

Hodnoty uložené v 4 B prevedené na čísla v desiatkovej sústave:

-60	0	0	0	=	196
-116	0	0	0	=	140
-84	0	0	0	=	172
108	0	0	0	=	108
76	0	0	0	=	76
44	0	0	0	=	44

Pamäť:

Zelená - voľné bloky

Fialová - hlavička pamäte

Žltá - posunutie na posledne uvoľnený blok

Tmavo modrá - posunutie na blok v inom zozname

Ostatné farby - ostatné posunutia v rámci jedného zoznamu

0x61fbd0:	-60	0	0	0	-116	0	0	0
0x61fbd8:	-84	0	0	0	0	0	0	0
0x61fbe0:	0	0	0	0	0	0	0	0
0x61fbe8:	-8	-1	-1	-1	1	1	1	1
0x61fbf0:	1	1	1	1	-8	-1	-1	-1
0x61fbf8:	8	0	0	0	0	0	0	0
0x61fc00:	76	0	0	0	8	0	0	0

0x61fc08:	-8	-1	-1	-1	1	1	1	1
0x61fc10:	1	1	1	1	-8	-1	-1	-1
0x61fc18:	8	0	0	0	44	0	0	0
0x61fc20:	108	0	0	0	8	0	0	0
0x61fc28:	-8	-1	-1	-1	1	1	1	1
0x61fc30:	1	1	1	1	-8	-1	-1	-1
0x61fc38:	8	0	0	0	76	0	0	0
0x61fc40:	-116	0	0	0	8	0	0	0
0x61fc48:	-8	-1	-1	-1	1	1	1	1
0x61fc50:	1	1	1	1	-8	-1	-1	-1
0x61fc58:	8	0	0	0	108	0	0	0
0x61fc60:	0	0	0	0	8	0	0	0
0x61fc68:	-8	-1	-1	-1	1	1	1	1
0x61fc70:	1	1	1	1	-8	-1	-1	-1
0x61fc78:	24	0	0	0	0	0	0	0
0x61fc80:	0	0	0	0	0	0	0	0
0x61fc88:	0	0	0	0	0	0	0	0
0x61fc90:	0	0	0	0	24	0	0	0

Scenár 1b

```

char region1[50];
memory_init(region1, 50);
char *p0 = memory_alloc(15);
char *p1 = memory_alloc(15); //cant allocate
char *p2 = memory_alloc(15); //cant allocate
memory_free(p0);             //freeing block scenario
notMemory/allocated/notMemory

```

Tento príklad znázorňuje internú fragmentáciu podobne ako scenár 1c pri 50 B.

0x61fb20:	46	0	0	0	0	0	0	0
0x61fb28:	0	0	0	0	0	0	0	0
0x61fb30:	-26	-1	-1	-1	1	1	1	1
0x61fb38:	1	1	1	1	1	1	1	1
0x61fb40:	1	1	1	1	1	1	1	1
0x61fb48:	1	1	1	1	1	1	-26	-1
0x61fb50:	-1	-1						

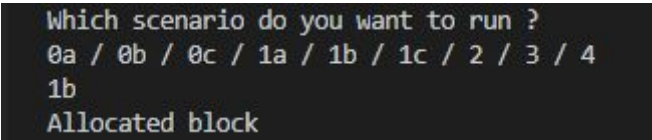
Užívateľ zadal, že chce alokovať 15 B, to sa zaokrúhlilo na 16 B (násobok čísla 2). Z 26 voľných bytov ostávalo už iba 10 B, ale z toho nebolo možné vytvoriť nový blok. lebo na nový blok je potrebných minimálne 16 B (4 B hlavička, 4 nasledujúce posunutie, 4 B predošlé posunutie, 4B pätička).

```

char region2[100];
memory_init(region2, 100);
char *p3 = memory_alloc(15);
char *p4 = memory_alloc(15);
char *p5 = memory_alloc(15);
char *p6 = memory_alloc(15);
if (memory_check(p5))
{
    printf("Allocated block\n");
}
char *p7 = memory_alloc(15);
memory_free(p4);
memory_free(p5); //freeing block scenario free/allocated/notMemory
char *p8 = memory_alloc(15);

```

Testovanie memory_check, na ukazovateli p5. Program vypíše do terminálu "Allocated block", čo znamená, že blok je alokovaný.



```

Which scenario do you want to run ?
0a / 0b / 0c / 1a / 1b / 1c / 2 / 3 / 4
1b
Allocated block

```

(snímka obrazovky z terminálu)

```

char region3[200];
memory_init(region3, 200);
char *p9 = memory_alloc(15);
char *p10 = memory_alloc(15);
char *p11 = memory_alloc(15);
char *p12 = memory_alloc(15);
char *p13 = memory_alloc(15);
char *p14 = memory_alloc(15);
memory_free(p11);
memory_free(p12); //freeing block scenario free/allocated/allocated

```

0x61fbd0:	-60	0	0	0	0	0	0	0
0x61fbd8:	76	0	0	0	0	0	0	0
0x61fbe0:	0	0	0	0	0	0	0	0
0x61fbe8:	-16	-1	-1	-1	1	1	1	1
0x61fbf0:	1	1	1	1	1	1	1	1
0x61fbf8:	1	1	1	1	-16	-1	-1	-1
0x61fc00:	-16	-1	-1	-1	1	1	1	1
0x61fc08:	1	1	1	1	1	1	1	1
0x61fc10:	1	1	1	1	-16	-1	-1	-1
0x61fc18:	16	0	0	0	-84	0	0	0
0x61fc20:	0	0	0	0	0	0	0	0
0x61fc28:	0	0	0	0	16	0	0	0

0x61fc30:	-16	-1	-1	-1	1	1	1	1
0x61fc38:	1	1	1	1	1	1	1	1
0x61fc40:	1	1	1	1	-16	-1	-1	-1
0x61fc48:	-16	-1	-1	-1	1	1	1	1
0x61fc50:	1	1	1	1	1	1	1	1
0x61fc58:	1	1	1	1	-16	-1	-1	-1
0x61fc60:	-16	-1	-1	-1	1	1	1	1
0x61fc68:	1	1	1	1	1	1	1	1
0x61fc70:	1	1	1	1	-16	-1	-1	-1
0x61fc78:	24	0	0	0	0	0	0	0
0x61fc80:	76	0	0	0	0	0	0	0
0x61fc88:	0	0	0	0	0	0	0	0
0x61fc90:	0	0	0	0	24	0	0	0

Testovanie uvoľňovania bloku v prípade, že predošlý blok je voľný a nasledujúci je alokovaný.

0x61fbd0:	-60	0	0	0	0	0	0	0
0x61fbd8:	0	0	0	0	76	0	0	0
0x61fbe0:	0	0	0	0	0	0	0	0
0x61fbe8:	-16	-1	-1	-1	1	1	1	1
0x61fbf0:	1	1	1	1	1	1	1	1
0x61fbf8:	1	1	1	1	-16	-1	-1	-1
0x61fc00:	-16	-1	-1	-1	1	1	1	1
0x61fc08:	1	1	1	1	1	1	1	1
0x61fc10:	1	1	1	1	-16	-1	-1	-1
0x61fc18:	40	0	0	0	0	0	0	0
0x61fc20:	0	0	0	0	0	0	0	0
0x61fc28:	0	0	0	0	0	0	0	0
0x61fc30:	0	0	0	0	0	0	0	0
0x61fc38:	0	0	0	0	0	0	0	0
0x61fc40:	0	0	0	0	40	0	0	0
0x61fc48:	-16	-1	-1	-1	1	1	1	1
0x61fc50:	1	1	1	1	1	1	1	1
0x61fc58:	1	1	1	1	-16	-1	-1	-1
0x61fc60:	-16	-1	-1	-1	1	1	1	1
0x61fc68:	1	1	1	1	1	1	1	1
0x61fc70:	1	1	1	1	-16	-1	-1	-1
0x61fc78:	24	0	0	0	0	0	0	0
0x61fc80:	0	0	0	0	0	0	0	0
0x61fc88:	0	0	0	0	0	0	0	0
0x61fc90:	0	0	0	0	24	0	0	0

Červená - nový voľný blok

Zadanie 1 - Správca pamäti

Scenár 1c

```
char region1[50];
memory_init(region1, 50);
char *p0 = memory_alloc(24);
char *p1 = memory_alloc(24); //can't allocate
char *p2 = memory_alloc(24); //can't allocate
if (p2 == NULL)
{
    printf("Couldn't allocate p2\n");
}
memory_free(p0); //freeing block scenario notMemory/allocated/notMemory
```

V scenári 1c pre pamäť o veľkosti 50 B je test zameraný na uvoľňovanie pamäte, keď je celá alokovaná jedným blokom. Takýto prípad je potrebné špeciálne opatriť vo funkcii `memory_free`.

-exec x/50b heap

0x61fb20:	46	0	0	0	0	0	0	0
0x61fb28:	0	0	0	0	0	0	0	0
0x61fb30:	-26	-1	-1	-1	1	1	1	1
0x61fb38:	1	1	1	1	1	1	1	1
0x61fb40:	1	1	1	1	1	1	1	1
0x61fb48:	1	1	1	1	1	1	-26	-1
0x61fb50:	-1	-1						

Z analýzy pamäte je vidieť, že **internú fragmentáciu**. Tá nastala z dôvodu, že 2 B je príliš malé miesto na vytvorenie nového voľného bloku a preto tieto 2 B boli pridelené k tým 24B. Ďalej je možné vidieť, že všetky **zoznamy voľných blokov** sú prázdne.

0x61fb20:	46	0	0	0	0	0	0	0
0x61fb28:	20	0	0	0	0	0	0	0
0x61fb30:	26	0	0	0	0	0	0	0
0x61fb38:	0	0	0	0	0	0	0	0
0x61fb40:	0	0	0	0	0	0	0	0
0x61fb48:	0	0	0	0	0	0	26	0
0x61fb50:	0	0						

Hore je možné vidieť už uvoľnenú pamäť.

```

char region2[100];
memory_init(region2, 100);
char *p3 = memory_alloc(24);
char *p4 = memory_alloc(24);
char *p5 = memory_alloc(24); //this won't be allocated because of
lack of space
memory_free(p3);
memory_free(p4); //just free everything from the memory

```

V tomto prípade sa tester snaží alokovať viac ako sa v skutočnosti dá, p5 sa nealokuje, hoci to vyzerá, že by v pamäti o veľkosti 100B bolo miesto na 3*24B alokáciu, v skutočnosti pri týchto malých pamätiach je veľká časť pamäte okupovaná réžiou (hlavičky, pätičky, zoznamy). Následne prebehne uvoľnenie celej pamäte.

-exec x/100b heap - príkaz v gdb debuggeri na zobrazenie 100B pamäte od adresy ukazovateľa heap

Stav pamäte po vykonaní troch alokácií. (3 nealokuje, kvôli nedostatku miesta). **Oranžová** farba znázorňuje prvú alokáciu, **modrá** druhú alokáciu pamäte. **Zelenou** je vyznačený voľný blok, ktorý vznikol pri druhej alokácii. **Červenou** je vyznačený zoznam, kde je uložené posunutie na voľný blok. Je to uložené v prvom zozname, lebo veľkosť prvého zoznamu je od 2^3 do $(2^4)-1$. **Fialová** znázorňuje hlavičku pamäte (veľkosť pamäte v hlavičke je uložená bez veľkosti hlavičky). Všetky číselné hodnoty majú veľkosť 4 byty, z dôvodu, že na ich ukladanie využívam 32 bit integer.

0x61fb90:	96	0	0	0	88	0	0	0
0x61fb98:	0	0	0	0	0	0	0	0
0x61fba0:	0	0	0	0	-24	-1	-1	-1
0x61fba8:	1	1	1	1	1	1	1	1
0x61fbb0:	1	1	1	1	1	1	1	1
0x61fbb8:	1	1	1	1	1	1	1	1
0x61fbc0:	-24	-1	-1	-1	-24	-1	-1	-1
0x61fbc8:	1	1	1	1	1	1	1	1
0x61fbd0:	1	1	1	1	1	1	1	1
0x61fbd8:	1	1	1	1	1	1	1	1
0x61fbe0:	-24	-1	-1	-1	8	0	0	0
0x61fbe8:	0	0	0	0	0	0	0	0
0x61fbf0:	8	0	0	0				

Stav pamäte po vykonaní oboch príkazov na uvoľnenie alokovanej pamäte (farebné označenie je analogické ako v predošlom zobrazení pamäte):

0x61fb90:	96	0	0	0	0	0	0	0
0x61fb98:	0	0	0	0	0	0	0	0

0x61fba0:	24	0	0	0	72	0	0	0
0x61fba8:	0	0	0	0	0	0	0	0
0x61fbb0:	0	0	0	0	0	0	0	0
0x61fbb8:	0	0	0	0	0	0	0	0
0x61fbc0:	0	0	0	0	0	0	0	0
0x61fbc8:	0	0	0	0	0	0	0	0
0x61fbd0:	0	0	0	0	0	0	0	0
0x61fbd8:	0	0	0	0	0	0	0	0
0x61fbe0:	0	0	0	0	0	0	0	0
0x61fbe8:	0	0	0	0	0	0	0	0
0x61fbf0:	72	0	0	0				

```

char region3[200];
memory_init(region3, 200);
char *p6 = memory_alloc(24);
char *p7 = memory_alloc(24);
char *p8 = memory_alloc(24);
memory_free(p8);
memory_free(p6);
char *p9 = memory_alloc(24);
char *p10 = memory_alloc(24);
char *p11 = memory_alloc(24);
memory_free(p11);

```

Testovanie alokácie a uvoľňovania tak, ako by to mohlo vyzerat' v reálne.
Pamäť po vykonaní všetkých funkcií:

0x61fbd0:	-60	0	0	0	0	0	0	0
0x61fbd8:	0	0	0	0	0	0	0	0
0x61fbe0:	124	0	0	0	0	0	0	0
0x61fbe8:	-24	-1	-1	-1	1	1	1	1
0x61fbf0:	1	1	1	1	1	1	1	1
0x61fbf8:	1	1	1	1	1	1	1	1
0x61fc00:	1	1	1	1	-24	-1	-1	-1
0x61fc08:	-24	-1	-1	-1	1	1	1	1
0x61fc10:	1	1	1	1	1	1	1	1
0x61fc18:	1	1	1	1	1	1	1	1
0x61fc20:	1	1	1	1	-24	-1	-1	-1
0x61fc28:	-24	-1	-1	-1	1	1	1	1
0x61fc30:	1	1	1	1	1	1	1	1
0x61fc38:	1	1	1	1	1	1	1	1
0x61fc40:	1	1	1	1	-24	-1	-1	-1
0x61fc48:	72	0	0	0	0	0	0	0

0x61fc50:	0	0	0	0	0	0	0	0
0x61fc58:	0	0	0	0	0	0	0	0
0x61fc60:	0	0	0	0	0	0	0	0
0x61fc68:	0	0	0	0	0	0	0	0
0x61fc70:	0	0	0	0	0	0	0	0
0x61fc78:	0	0	0	0	0	0	0	0
0x61fc80:	0	0	0	0	0	0	0	0
0x61fc88:	0	0	0	0	0	0	0	0
0x61fc90:	0	0	0	0	72	0	0	0

Scenár 2

Scenár 2 slúži na testovanie a vyhodnocovanie alokácie pamäte. Funkcia päť-krát vygeneruje náhodnú veľkosť bloku a alokuje, pokým sa už ďalej nedá alokovať a následne vyhodnotí úspešnosť alokácie.

Po výbere funkcie 2, na začiatku vypíše, o aký scenár sa jedná, veľkosť pamäte a veľkosť hlavičky pamäte spolu s veľkosťou zoznamov. Následne pre každý jeden test vypíše info ohľadom toho konkrétneho testu.

Od scenára 2 je taktiež odvodený scenár 0, ktorý robí to isté ako scenár 2 s rozdielom, že je potrebné manuálne zadať veľkosť bloku. V programe som to využil na simulovanie scenára 1.

Hodnoty, pre ktoré je scenár 2 vykonávaný je možné vidieť na začiatku dokumentácie v rozšírení zadania.

Payload size: veľkosť bloku (ktorý si užívateľ vypýtal)

Costs: Réžia (hlavičky, pätičky alokovaných blokov)

Internal fragmentation: vzniknutá interná fragmentácia

Intended sum of memory: súčet užívateľom vyžiadaných veľkostí

Theoretical allocated volume: teoretické zaplnenie pamäte (keby nebola potrebná réžia)

Memory allocated (without internal fragmentation): Alokovaná pamäť bez internej fragmentácie

Memory allocated : Celkovo alokovaná pamäť

=====

Memory allocation scenario 2:

Memory size: 50

Memory header and list size: 16B

Payload size: 18B

Costs: 8B

Internal fragmentation: 8B

Intended sum of memory: 18B

Zadanie 1 - Správca pamäti

Theoretical allocated volume: 36.00%
Memory allocated (without internal fragmentation): 52.94 %
Memory allocated : 76.47 %

Payload size: 18B
Costs: 8B
Internal fragmentation: 8B
Intended sum of memory: 18B
Theoretical allocated volume: 36.00%
Memory allocated (without internal fragmentation): 52.94 %
Memory allocated : 76.47 %

Payload size: 21B
Costs: 8B
Internal fragmentation: 5B
Intended sum of memory: 21B
Theoretical allocated volume: 42.00%
Memory allocated (without internal fragmentation): 61.76 %
Memory allocated : 76.47 %

Payload size: 22B
Costs: 8B
Internal fragmentation: 4B
Intended sum of memory: 22B
Theoretical allocated volume: 44.00%
Memory allocated (without internal fragmentation): 64.71 %
Memory allocated : 76.47 %

Payload size: 21B
Costs: 8B
Internal fragmentation: 5B
Intended sum of memory: 21B
Theoretical allocated volume: 42.00%
Memory allocated (without internal fragmentation): 61.76 %
Memory allocated : 76.47 %

=====

=====

Memory allocation scenario 2:
Memory size: 100
Memory header and list size: 20B

Payload size: 18B
Costs: 24B
Internal fragmentation: 2B
Intended sum of memory: 54B
Theoretical allocated volume: 54.00%
Memory allocated (without internal fragmentation): 67.50 %
Memory allocated : 70.00 %

Payload size: 18B
Costs: 24B
Internal fragmentation: 2B
Intended sum of memory: 54B
Theoretical allocated volume: 54.00%
Memory allocated (without internal fragmentation): 67.50 %
Memory allocated : 70.00 %

Payload size: 21B
Costs: 16B
Internal fragmentation: 2B
Intended sum of memory: 42B
Theoretical allocated volume: 42.00%
Memory allocated (without internal fragmentation): 52.50 %
Memory allocated : 55.00 %

Payload size: 22B
Costs: 16B
Internal fragmentation: 0B
Intended sum of memory: 44B
Theoretical allocated volume: 44.00%
Memory allocated (without internal fragmentation): 55.00 %
Memory allocated : 55.00 %

Payload size: 21B
Costs: 16B
Internal fragmentation: 2B
Intended sum of memory: 42B
Theoretical allocated volume: 42.00%
Memory allocated (without internal fragmentation): 52.50 %
Memory allocated : 55.00 %

=====

=====

Memory allocation scenario 2:
Memory size: 200
Memory header and list size: 24B

Payload size: 18B
Costs: 48B
Internal fragmentation: 0B
Intended sum of memory: 108B
Theoretical allocated volume: 54.00%
Memory allocated (without internal fragmentation): 61.36 %
Memory allocated : 61.36 %

Payload size: 18B
Costs: 48B
Internal fragmentation: 0B
Intended sum of memory: 108B

Theoretical allocated volume: 54.00%
Memory allocated (without internal fragmentation): 61.36 %
Memory allocated : 61.36 %

Payload size: 21B
Costs: 40B
Internal fragmentation: 5B
Intended sum of memory: 105B
Theoretical allocated volume: 52.50%
Memory allocated (without internal fragmentation): 59.66 %
Memory allocated : 62.50 %

Payload size: 22B
Costs: 40B
Internal fragmentation: 0B
Intended sum of memory: 110B
Theoretical allocated volume: 55.00%
Memory allocated (without internal fragmentation): 62.50 %
Memory allocated : 62.50 %

Payload size: 21B
Costs: 40B
Internal fragmentation: 5B
Intended sum of memory: 105B
Theoretical allocated volume: 52.50%
Memory allocated (without internal fragmentation): 59.66 %
Memory allocated : 62.50 %

=====

Scenár 3

Scenár 3 slúži na testovanie a vyhodnocovanie alokácie pamäte. Funkcia päť-krát vygeneruje náhodnú veľkosť bloku a alokuje, pokiaľ sa už ďalej nedá alokovať a následne vyhodnotí úspešnosť alokácie.

Po výbere funkcie 3, na začiatku vypíše, o aký scenár sa jedná, veľkosť pamäte a veľkosť hlavičky pamäte spolu s veľkosťou zoznamov. Následne pre každý jeden test vypíše info ohľadom toho konkrétneho testu.

Hodnoty, pre ktoré je scenár 3 vykonávaný je možné vidieť na začiatku dokumentácie v rozšírení zadania.

=====

Memory allocation scenario 3:
Memory size: 10000
Memory header and list size: 48B

Payload size: 4765B

Zadanie 1 - Správca pamäti

Costs: 16B
Internal fragmentation: 2B
Intended sum of memory: 9530B
Theoretical allocated volume: 95.30%
Memory allocated (without internal fragmentation): 95.76 %
Memory allocated : 95.78 %

Payload size: 2801B
Costs: 24B
Internal fragmentation: 3B
Intended sum of memory: 8403B
Theoretical allocated volume: 84.03%
Memory allocated (without internal fragmentation): 84.44 %
Memory allocated : 84.47 %

Payload size: 1634B
Costs: 48B
Internal fragmentation: 0B
Intended sum of memory: 9804B
Theoretical allocated volume: 98.04%
Memory allocated (without internal fragmentation): 98.51 %
Memory allocated : 98.51 %

Payload size: 3387B
Costs: 16B
Internal fragmentation: 2B
Intended sum of memory: 6774B
Theoretical allocated volume: 67.74%
Memory allocated (without internal fragmentation): 68.07 %
Memory allocated : 68.09 %

Payload size: 4865B
Costs: 16B
Internal fragmentation: 2B
Intended sum of memory: 9730B
Theoretical allocated volume: 97.30%
Memory allocated (without internal fragmentation): 97.77 %
Memory allocated : 97.79 %

=====

Scenár 4

Scenár 4 slúži na testovanie a vyhodnocovanie alokácie pamäte. Funkcia päť-krát vygeneruje náhodnú veľkosť bloku a alokuje, pokým sa už ďalej nedá alokovať a následne vyhodnotí úspešnosť alokácie.

Po výbere funkcie 4, na začiatku vypíše, o aký scenár sa jedná, veľkosť pamäte a veľkosť hlavičky pamäte spolu s veľkosťou zoznamov. Následne pre každý jeden test vypíše info ohľadom toho konkrétneho testu.

Hodnoty, pre ktoré je scenár 4 vykonávaný je možné vidieť na začiatku dokumentácie v rozšírení zadania.

=====

Memory allocation scenario 4:

Memory size: 100000

Memory header and list size: 60B

Payload size: 17874B

Costs: 40B

Internal fragmentation: 0B

Intended sum of memory: 89370B

Theoretical allocated volume: 89.37%

Memory allocated (without internal fragmentation): 89.42 %

Memory allocated : 89.42 %

Payload size: 7829B

Costs: 96B

Internal fragmentation: 12B

Intended sum of memory: 93948B

Theoretical allocated volume: 93.95%

Memory allocated (without internal fragmentation): 94.00 %

Memory allocated : 94.02 %

Payload size: 20375B

Costs: 32B

Internal fragmentation: 4B

Intended sum of memory: 81500B

Theoretical allocated volume: 81.50%

Memory allocated (without internal fragmentation): 81.55 %

Memory allocated : 81.55 %

Payload size: 17694B

Costs: 40B

Internal fragmentation: 0B

Intended sum of memory: 88470B

Theoretical allocated volume: 88.47%

Memory allocated (without internal fragmentation): 88.52 %

Memory allocated : 88.52 %

Payload size: 23445B

Costs: 32B

Internal fragmentation: 4B

Intended sum of memory: 93780B

Theoretical allocated volume: 93.78%

Memory allocated (without internal fragmentation): 93.84 %

Memory allocated : 93.84 %

Zadanie 1 - Správca pamäti

Záver

Cieľom zadania bolo naprogramovať správcu pamäte a následne spraviť testy alokácie a vyhodnotenie týchto testov. Z testov vyplýva, že percentuálna úspešnosť alokácie pri malých pamätiach je omnoho menšia ako na obrovských pamätiach. Je to najmä z dôvodu, že pri menších pamätiach veľkosť réžie nie je až tak zanedbateľná ako pri väčších pamätiach, keď sa alokojú väčšie bloky.