SLOVENSKÁ TECHNICKÁ UNIVERZITA
Fakulta informatiky a informačních technologií
Ilkovičova 2, 842 16 Bratislava 4

Zadanie 1 - Správca pamäti

Dávid Penťa ID: 110871
Datové štruktúry a algoritmy
FIIT STU 2020/2021
Cvičenie: Štvrtok 9:00
10.3.2021

Dávid Penťa
ID: 110871

Zadanie 1 –Správca pamäti

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 veľkosť vyhradiť.
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 prideľuje vaša 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 pridelíte pamäť funkciou **memory_alloc**, mala by byť dostupná pre program (nemala by presahovať pôvodný blok, ani prekrývať 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:

- prideľovanie 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),
- prideľovanie 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),
- prideľovanie nerovnakých blokov väčšej veľkosti (veľkosti 500 až 5000 bytov) pri použití väčších celkových blokov pre správcu pamäte (aspoň veľkosti 1000 bytov),
- prideľovanie 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).

Dávid Penťa
ID: 110871

V testovacích scenároch okrem iného vyhodnoťte koľko % blokov sa vám podarilo alokovať oproti ideálnemu riešeniu (bez vnútornej aj vonkajšej fragmentácie). Teda snažte sa prideľovať 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 nákresmi/obrázkami a krátkymi ukážkami zdrojového kódu, vyberajte len kód, na ktorý chcete extra upozorniť. Pri opise sa snažte dbať osobitý 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 prispôsobiť dĺžku (počet bytov) hlavičky bloku podľa jeho veľkosti. Každé funkčné vylepšenie cvičiaci zohľadní pri bodovaní.
Cvičiaci prideľuje body podľa kvality vypracovania. 8 bodov môžete získať za vlastný funkčný program prideľovania 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**).

Dávid Penťa
ID: 110871

Dokumentácia musí obsahovať:

1. titulná strana
    a. názov inštitúcie,
    b. názov predmetu,
    c. názov zadania,
    d. meno a priezvisko,
    e. ais id,
    f. akademicky 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
    a. použite 2 spôsoby opisu
    b. doplňte ukážky zdrojového kódu na ktorý chcete extra upozorniť
4. testovanie
    a. scenár 1
        i. 8 do 50/100/200
        ii. 15 do 50/100/200
        iii. 24 do 50/100/200
    b. scenár 2
        i. rand (8-24) do 50/100/200 – zápis 5 hodnôt cyklicky do naplnenia pamäte
    c. scenár 3
        i. rand (500 – 5000) do 10 000 – zápis 5 hodnôt cyklicky do naplnenia pamäte
    d. scenár 4
        i. rand (8 – 50 000) do 100 000 – zápis 5 hodnôt cyklicky do naplnenia pamäte
    e. 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
    f. 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

Dávid Penťa
ID: 110871

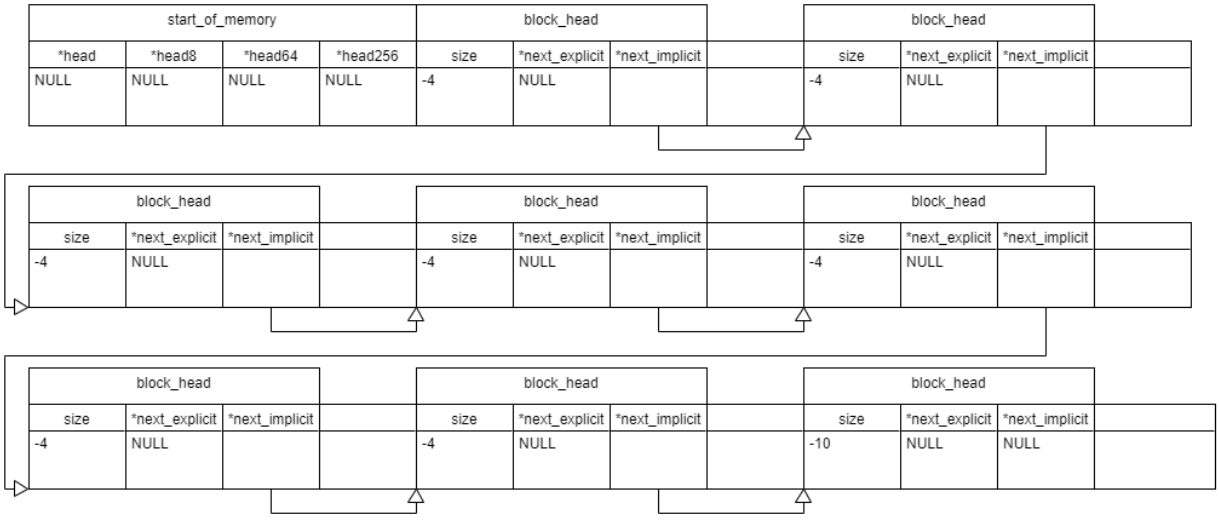
Použitá metóda

Môj program využíva oddelené explicitné zoznamy blokov voľnej pamäti. Používa štyri zoznamy pre triedy voľných blokov pamäti veľkosti 1 až 8 bajtov, 9 až 64 bajtov, 65 až 256 bajtov a bloky väčšie ako 257 bajtov. Ukazovatele na prvý blok v zozname sú uložené na začiatku pamäte, ktorú program dostane. V programe sú uložené pomocou štruktúry *start_of_memory,* ktorá je použitá v programe len raz. Program používa ešte jednu štruktúru *block_head*, ktorá jednotne definuje hlavičku každého bloku, takže všetky majú veľkosť 12 bajtov

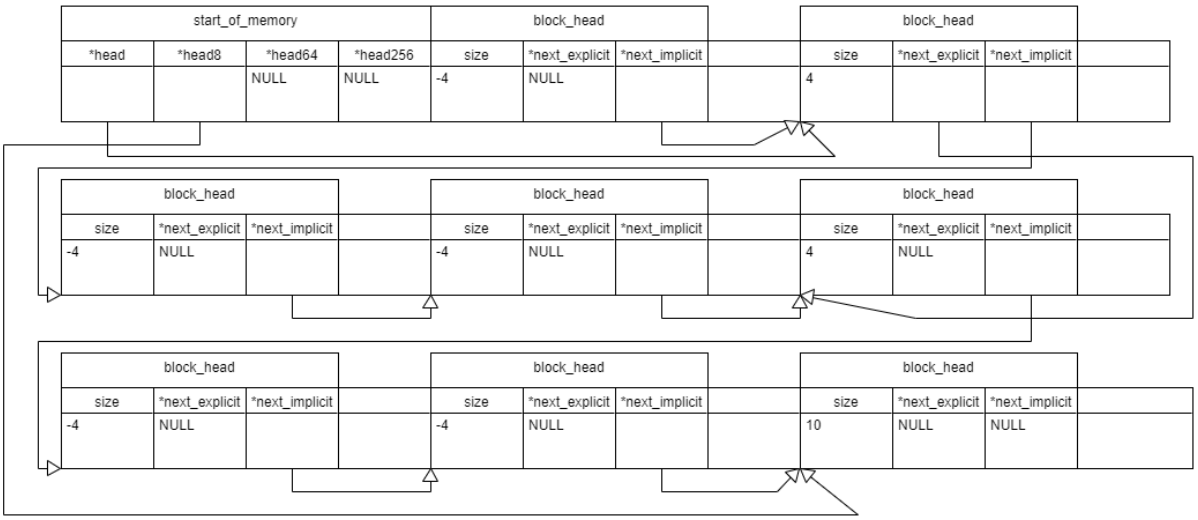
```
struct block_head{
    int size;
    struct block_head *next_explicit;
    struct block_head *next_implicit;
};

struct start_of_memory{
    struct block_head *head;
    struct block_head *head8;
    struct block_head *head64;
    struct block_head *head256;
};
```

Obr. 1.:Definica štruktúr block_head a start_of_memory.

Hlavička obsahuje informáciu o počte bajtov v tele, s využitím znamienka na indikovanie, či je blok voľný. Ak je počet záporný, tak je blok používaný, ak je kladný, tak je možné do neho ukladať. V hlavičke sú aj dva ukazovatele. Prvý je implicitný, ktorý bez ohľadu na to, či je blok voľný, ukazuje na ďalší blok v pamäti, tak ako sú v nej uložené. Posledný prvok v pamäti má tento ukazovateľ nastavený na NULL. Druhý ukazovateľ v hlavičke je explicitný a všetky bloky ktoré nie sú voľné, ho majú nastavený na NULL. Voľné bloky ním ukazujú na ďalší voľný blok v rovnakej veľkostnej triede, čím tvoria spájaný zoznam. Posledný prvok v spájanom zozname má tento ukazovateľ nastavený na NULL.



Obr. 1.: Reprezentácia pamäte bez voľných blokov.



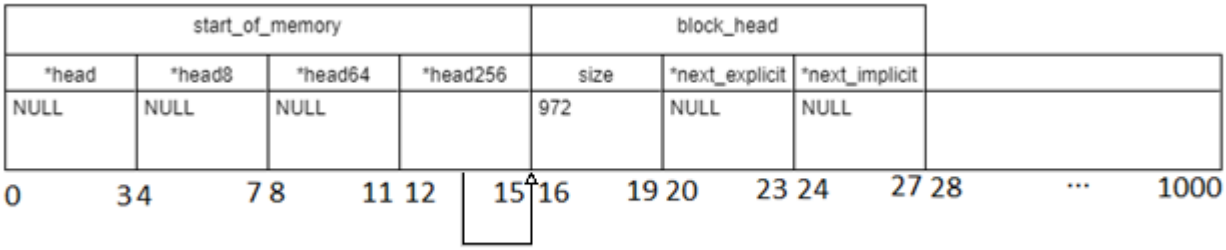
Obr. 2.: Reprezentácia pamäte s voľnými blokmi.

Dávid Penťa
ID: 110871

Implementácia funkcií

1. void memory_init(void *ptr, unsigned int size)

Po zavolaní funkcia uloží do globálnej premennej ukazovateľ na pamäť, ktorá vstupuje do funkcie a vloží na začiatok tejto pamäte štruktúru start_of_memory, ktorá obsahuje štyri ukazovatele. Potom zo zvyšku pamäte urobí jeden blok, ktorý má pôvodnú veľkosť mínus 28, ako je možné vidieť na obrázku nižšie. Následne na tento voľný blok nastaví ukazovateľ správnej veľkostnej triedy v štruktúre start_of_memory a ostatné ukazovatele nastaví na NULL.



Obr. 3.: Reprezentácia pamäte po použití *memory_init(p,1000).*

2. void *memory_alloc(unsigned int size)

Vo funkcii memory_alloc(), program najprv skúsi nájsť voľný blok rovnakej veľkosti, aký chce používateľ alokovať. Postupuje tak, že prehľadá spájaný zoznam správnej veľkostnej triedy. Ak sa mu taký nepodarí nájsť, prejde na upravenú verziu first fit, ktorá začne prechádzať spájané zoznamy od najnižšie možnej veľkostnej triedy do ktorej sa blok zmestí. To má za následok, že sa dlhé bloky pamäte nebudú zbytočne zapĺňať malými blokmi, ak pre nich bude menšie miesto niekde inde v pamäti, a preto sa bude môcť alokovať viac blokov. Týmto spôsobom nájdený blok sa rozdelí vo funkcii *split()* na dva bloky. Prvý, ktorý má veľkosť akú si vypýtal používať, sa vráti, a druhý, voľný blok, sa pridá do spájaného zoznamu správnej veľkostnej triedy. Ak sa nájdený blok nemôže rozdeliť z dôvodu nedostatku miesta, tak sa celý použije a program ho vráti používateľovi. Takto sa zabráni, aby sa v pamäti tvoril garbage.

3. int memory_free(void *valid_ptr)

Táto funkcia, ako jediná, využíva implicitný list. Vždy, keď sa zavolá táto funkcia, tak skontroluje, či sa na ukazovateli nachádza blok, a ak áno, a zároveň v jeho veľkosti je uložená záporná hodnota, tak sa táto hodnota zmení na kladnú. Potom sa prechádza implicitným zoznamom vo funkcii connect(), a keď sa nájdu tri alebo dva voľné bloky vedľa seba v pamäti, tak sa spoja do

jedného bloku a zavolá sa ďalšia funkcia connect2(), ktorá znovu začne prechádzať implicitným zoznamom a priraďuje ukazovatele na voľné bloky do správnych spájaných zoznamov.

4. int memory_check(void *ptr)

Toto je najjednoduchšia funkcia, ktorá len skontroluje, či sa na ukazovateli nachádza blok, a ak áno, a v jeho veľkosti je uložená záporná hodnota, čiže nie je voľný, tak funkcia vráti hodnotu 1. V akomkoľvek inom prípade funkcia vráti hodnotu 0.

Dokumentácia testovaní

Pri písaní kódu som kontroval cez funkcionalitu v prostredí CLion, či sa správne delia bunky a či sa správne vytvárajú spájané zoznamy. Testoval som ukladanie rovnako veľkých, ako aj náhodne veľkých blokov do malej, strednej a veľkej vyhradenej pamäte. Z výsledkov je zrejmé, že tento program nie je veľmi výhodné používať, ak chceme vyhradiť malú pamäť, pretože ak si napríklad vyhradíme pamäť veľkosti 50 bajtov, po odčítaní štyroch pointerov na začiatku, ktoré majú spolu 16 bajtov a odčítaní prvej hlavičky, ktorá má 12 bajtov, tak ostane len 22 voľných bajtov na alokovanie, čo je z pôvodných 50 bajtov menej ako polovica. Pri malej pamäti je tiež zbytočné mať viacero zoznamov voľných blokov. Avšak pri veľkej vyhradenej pamäti, kde je 16 bajtov na začiatku zanedbateľných, je tento program vďaka viacerým explicitným zoznamom veľmi rýchly. A ak sa priebežne pamäť aj uvoľňuje a vznikajú v strede voľné bloky, dokáže ekonomicky pracovať s dostupnou pamäťou a umiestni blok do voľného bloku čo najmenšej veľkostnej kategórie aby sa šetrilo miesto na alokáciu väčších blokov. Pri testovaní sa do pamäte väčšej ako 10.000 bajtov sa väčšinou zmestilo viac ako 90% možných blokov a využilo sa nad 99% dostupných bajtov.

Výpis testovania

```
Scenario 1
Block of size 8 in memory size of 50 bytes: allocated 16.67% blocks (16.67% bytes).
Block of size 8 in memory size of 100 bytes: allocated 33.33% blocks (33.33% bytes).
Block of size 8 in memory size of 200 bytes: allocated 36.00% blocks (36.00% bytes).

Block of size 15 in memory size of 50 bytes: allocated 33.33% blocks (33.33% bytes).
Block of size 15 in memory size of 100 bytes: allocated 50.00% blocks (50.00% bytes).
Block of size 15 in memory size of 200 bytes: allocated 46.15% blocks (46.15% bytes).

Block of size 24 in memory size of 50 bytes: allocated 0.00% blocks (0.00% bytes).
Block of size 24 in memory size of 100 bytes: allocated 50.00% blocks (50.00% bytes).
Block of size 24 in memory size of 200 bytes: allocated 62.50% blocks (62.50% bytes).

Scenario 2
Block of random size in memory size of 50 bytes: allocated 33.33% blocks (32.61% bytes).
Block of random size in memory size of 100 bytes: allocated 40.00% blocks (40.40% bytes).
Block of random size in memory size of 200 bytes: allocated 58.33% blocks (50.51% bytes).

Scenario 3
Block of random size in memory size of 10000 bytes: allocated 100.00% blocks (100.00% bytes).
Block of random size in memory size of 10000 bytes: allocated 80.00% blocks (93.34% bytes).
Block of random size in memory size of 10000 bytes: allocated 80.00% blocks (87.58% bytes).
Block of random size in memory size of 10000 bytes: allocated 100.00% blocks (100.00% bytes).
Block of random size in memory size of 10000 bytes: allocated 83.33% blocks (90.54% bytes).
Block of random size in memory size of 10000 bytes: allocated 100.00% blocks (100.00% bytes).
Block of random size in memory size of 10000 bytes: allocated 100.00% blocks (100.00% bytes).
```

```
Block of random size in memory size of 10000 bytes: allocated 100.00% blocks (100.00% bytes).
Block of random size in memory size of 10000 bytes: allocated 100.00% blocks (100.00% bytes).
Block of random size in memory size of 10000 bytes: allocated 100.00% blocks (100.00% bytes).

Scenario 4
Block of random size in memory size of 100000 bytes: allocated 90.00% blocks (99.80% bytes).
Block of random size in memory size of 100000 bytes: allocated 90.00% blocks (98.16% bytes).
Block of random size in memory size of 100000 bytes: allocated 90.91% blocks (99.20% bytes).
Block of random size in memory size of 100000 bytes: allocated 91.67% blocks (99.85% bytes).
Block of random size in memory size of 100000 bytes: allocated 72.22% blocks (99.82% bytes).
Block of random size in memory size of 100000 bytes: allocated 88.89% blocks (99.69% bytes).
Block of random size in memory size of 100000 bytes: allocated 90.91% blocks (97.65% bytes).
Block of random size in memory size of 100000 bytes: allocated 78.57% blocks (99.85% bytes).
Block of random size in memory size of 100000 bytes: allocated 84.62% blocks (99.85% bytes).
Block of random size in memory size of 100000 bytes: allocated 85.71% blocks (99.84% bytes).

Scenario 5
Block of size 1000000 cannot fit in memory size of 1000.
Process finished with exit code 0
```

Rozbor zložitosti

Časová zložitosť

1. *void *memory_alloc(unsigned int size)* má v najhoršom prípade lineárnu časovú zložitosť O(n), ale pretože pri hľadaní vhodného voľného bloku prechádza len voľnými blokmi v možných veľkostných triedach, je väčšinou oveľa lepšia, ako keby by mal program prechádzať všetkými blokmi implicitne.

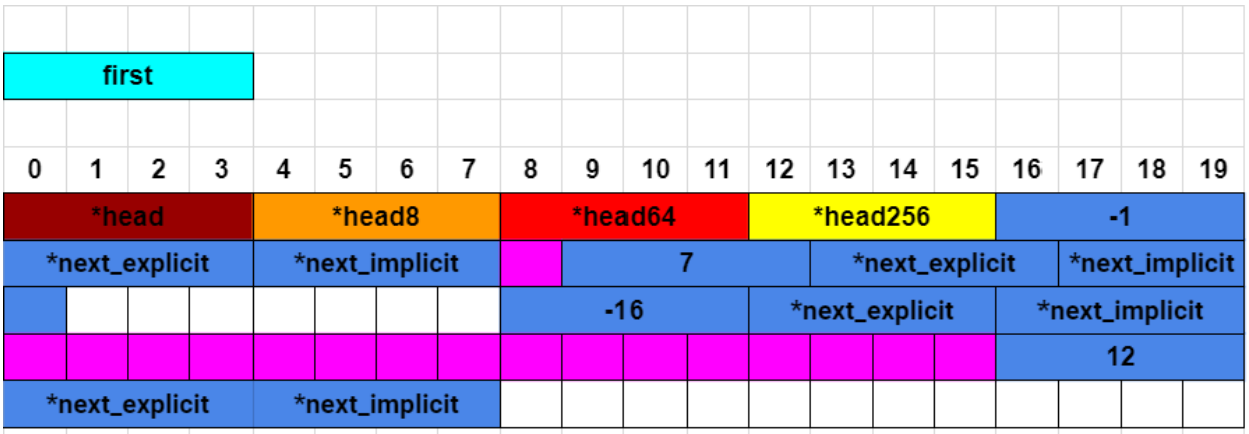
2. int memory_free(void *valid_ptr) je v najhoršom prípade najviac časovo zložitá funkcia v mojom kóde, ale stále jej zložitosť je len O(n). Je to kvôli tomu, že vždy keď sa uvoľňuje blok, tak sa zavolá funkcia *connect(),* ktorá obsahu while cyklus, ktorým implicitne hľadá voľné bloky vedľa seba. Ak také nájde, zavolá sa ďalšia funkcia *connect2(),* ktorá začne prideľovať ukazovatele od začiatku pridelenej pamäte, až po jej koniec. Nejedná sa tu o kvadratickú zložitosť pamäte, pretože po zavolaní funkcie *connect2(),* sa prvý while cyklus preruší príkazom return.

3. int memory_check(void *ptr) má vždy časovú zložitosť O(1), pretože nepoužíva cykly, a len sa pozrie, či ukazovateľ ukazuje na blok, a či blok na ktorý ukazuje, je voľný.

4. void memory_init(void *ptr, unsigned int size) má časovú zložitosť O(1), pretože len vytvorí prvý blok a správne nastaví začiatky spájaných polí, k čomu nepotrebuje využívať cykly.

Pamäťová zložitosť

Pamäťová zložitosť programu je O(n), pretože využíva len pridelenú pamäť a mimo nej len jedinú globálnu premennú *first*, na označenie začiatku vyhradenej pamäte.



Obr. 4.: Reprezentácia využitia pamäte po bajtoch.

Záver

Najväčšou prednosťou tohto programu je práca s veľkou vyhradenou pamäťou, pretože sa môže naplno využiť rýchlosť alokácie bloku a rozumné využívanie vyhradenej pamäte. Nevýhodou je pomalosť funkcie na uvoľňovanie pamäte, ktorá prechádza pamäť implicitne, ale zato je veľmi spoľahlivá a je zaručené, že všetky bloky vedľa seba sa spoja a že spájané zoznamy budú vždy správne prepojené.