SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

Fakulta informatiky a informačných technológií

Ilkovičova 2, 842 16 Bratislava 4

Správca pamäti

(DSA - Zadanie 1)

Peter Škríba

Cvičiaci: Ing. Peter Pištek, PhD.

Cvičenie: Pondelok, 16:00

15.3.2020

DOKUMENTÁCIA

Zadanie správca pamäti, kde sme mali vytvoriť algoritmy na alokovanie a uvoľnenie pamäte, podobný funkciám malloc a free z knižnice stdlib.h som riešil nasledovne.

Opis implementácie

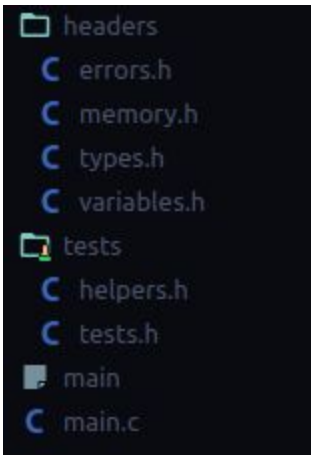
Pre moju implementáciu som použil metódu **explicitného zoznamu**, ktorá pozostáva hlavne z jednosmerného spájaného zoznamu voľných blokov. Na reprezentáciu bloku som použil **štruktúry**, ktoré slúžia ako **hlavička** a **päta**. Hlavička obsahuje veľkosť bloku ktorá je prístupná užívateľovi a ukazovateľ na ďalší voľný blok. Päta obsahuje len veľkosť bloku, ktorá sa ako aj v hlavičke mení na zápornú ak je blok alokovaný.

Na vyhľadávanie voľných blokov používam algoritmus **first-fit** trochu spojený s **best-fit** algoritmom, pretože udržiavam spájaný zoznam usporiadaný vzostupne podľa veľkosti blokov. Po alokácií sa mení hlavička a nevyužitý ukazovateľ sa pošle užívateľovi ako začiatok na jeho pamäť. Týmto som zaručil balans medzi **fragmentaciou** a **rýchlosťou**.

Na testovanie implementácie som vytvoril rôzne testy, ktorých výsledky majú grafický výstup. Nižšie je celá implementácie prebratá do hĺbky.

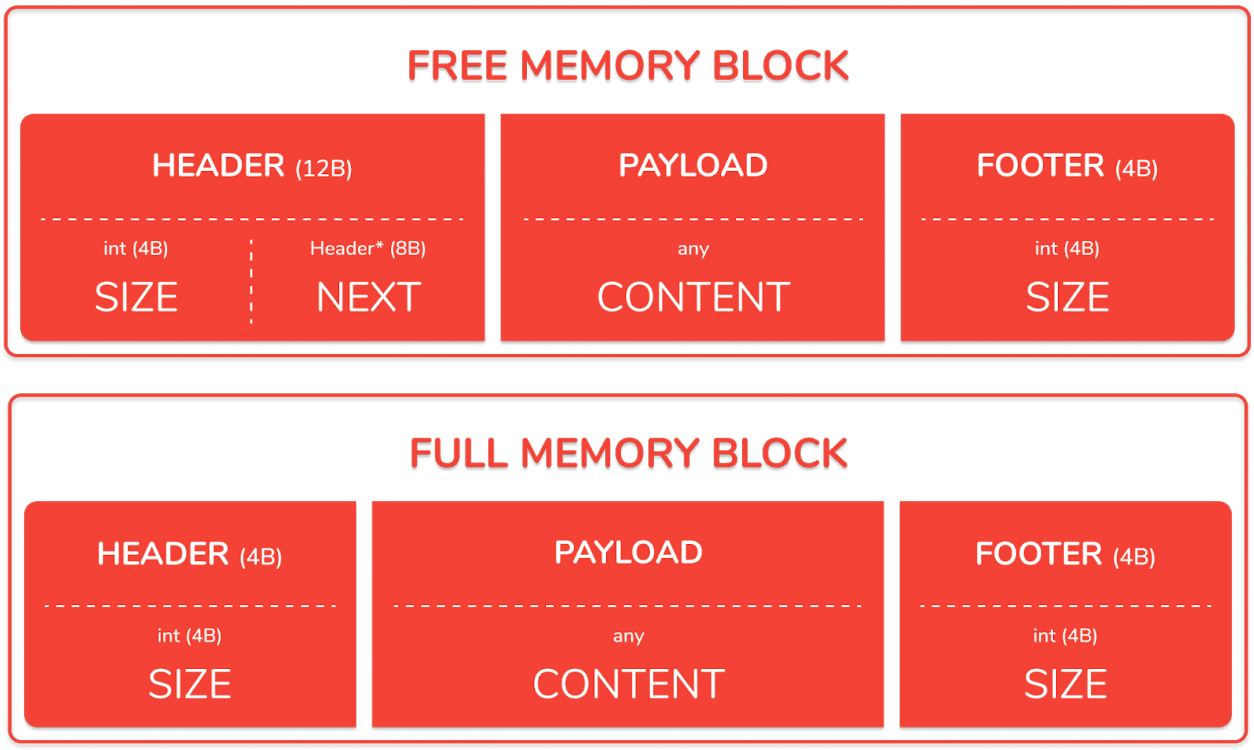
Štruktúra projektu

Pre lepšiu prehľadnosť som projekt štrukturoval do priečinkov, ktoré pozostávajú z **hlavného súboru** (main.c), hlavičkových súborov ako **memory**, **types**, **variables** a **errors**, ktoré spolupracujú a riešia implementáciu alokovania pamäti a nakoniec súborov ktoré zabezpečujú **testovanie** implementácie.



Dizajn pamäťových blokov

Ako som už spomínal sú dva druhy pamäťových blokov, ktoré pozostávajú z troch častí. V hlavičke sa nachádza **veľkosť** typu int (4B) a **ukazovateľ** na ďalšiu hlavičku (8B). Päta obsahuje len veľkosť bloku rovnú veľkosti v hlavičke. **Najmenší** blok má veľkosť **16B**. Na obrázku sú znázornené tieto dva typy pamäťových blokov.



Obr. 1 - Štruktúra voľného a obsadeného pamäťového bloku

Opis funkcií

Pre správne fungovanie mojej implementácie som potreboval riešiť štyri hlavné funkcie.

memory_init - časová zložitosť: O(1), pamäťová zložitosť: O(1)

Pre správne fungovanie nasledujúcich funkcií potrebujeme funkciu memory_init, ktorá pozostáva z troch častí: **zapísanie globálneho ukazovateľa na pamäť, vytvorenie**

pamäťovej hlavičky a **vytvorenie prvého voľného bloku**. Funkcia prijme parametre ako **region** (ukazovateľ na začiatok pamäte) a **size** (veľkosť pamäte). Ako prvé sa zapíše tento ukazovateľ na region do globálnej premennej s názvom **heap_g** a potom sa začnú vytvárať hlavičky a päty, ktoré sa použijú na vytvorenie pamäťovej hlavičky a voľného bloku. Po vykonaní môžu začať ostatné funkcie využívať pamäť, ktorá vyzerá nasledovne *(Obr. 2)*.

Obr. 2 - Pamäť po funkcií memory_init

memory_alloc - časová zložitosť: O(n), pamäťová zložitosť: O(1), kde n je počet voľných blokov spájaného zoznamu

Funkcia memory_alloc podobná funkcií malloc (stdlib.h) má na starosti nájsť a vrátiť ukazovateľ na začiatok payloadu užívateľovi, ktorý zadal ako vstupný argument požadovanú veľkosť. Skladá sa z dvoch častí, z ktorých každú obsluhuje osobitná pomocná funkcia. Ako prvé sa **nájde voľný blok**, ktorý zodpovedá našej veľkosti, keďže je väčšia pravdepodobnosť že užívateľ bude požadovať skôr blok menšej veľkosti ako väčšej, sú bloky zoradené vzostupne a tento blok bude sa nachádzať medzi prvými. Ak sa tento blok nenájde vráti sa NULL a ukáže sa chybová hláška ale ak sa nájde, odpojí sa zo spájaného zoznamu a prechádza do ďalšej funkcie, ktorá rieši **rozdelenie bloku**. Tam sa zistí či sa blok oplatí rozdeľovať, to znamená, že blok má presnú veľkosť ako požadujeme alebo po rozdelení by bol blok moc malý na to aby sa dal z neho vytvoriť blok ktorý obsahuje hlavičku a pätu a vtedy vrátime užívateľovi aj tú zvyšnú veľkosť, ktorá by zostala nevyužitá. Ak sa tento blok rozdelí nový blok sa vloží do spájaného zoznamu a požadovaný blok sa pošle užívateľovi s tým, že je označený ako plný **zápornou veľkosťou** v hlavičke a päte. *(Obr. 1)*

memory_free - časová zložitosť: O(n), pamäťová zložitosť: O(1), kde n je počet voľných blokov spájaného zoznamu

Funkcia memory_free, ktorá pracuje s alokovaným blokom a jej úlohou je uvoľniť blok a zapísať ho do zoznamu blokov voľnej pamäti je rozdelená na tri časti. Ako prvé **skontroluje** či **ukazovateľ**, ktorý vstupuje do funkcie ako argument je správny ukazovateľ na alokovaný blok. Ďalej označí blok že je voľný a pokračuje k najdôležitejšej časti a to je spájanie voľných blokov. Podmienky skontrolujú či je voľný **ľavý** a **pravý sused** a podľa toho sa rozhodne či posunie hlavičku na hlavičku ľavého suseda alebo pätu na pravého suseda a pritom vypočíta novú veľkosť bloku. Tento nový blok sa nakoniec zapíše do spájaného zoznamu voľných blokov a vráti sa 0 ak sa nevyskytla chyba, inak sa vráti 1. *(Obr. 1)*

memory_check - časová zložitosť: O(n), pamäťová zložitosť: O(1), kde n je počet všetkých blokov pamäte

Funkcia memory_check, ktorá pomáha funkcií memory_free, zisťuje či vstupný argument je ukazovateľ na payload alokovaného bloku v pamäti. Aby funkcia nebežala zbytočne, ako prvé skontroluje či je tento ukazovateľ z rozsahu pamäte. Prvý cyklus slúži na **nájdenie** prvého **voľného bloku**, ktorého ukazovateľ je väčší ako ukazovateľ hľadaného miesta. Nasledujúci cyklus začína od tohto ukazovateľa a skáče po obsadených blokoch smerom vzad až pokým nenarazí na **hľadaný blok** alebo na **prázdny blok**. Ak narazil na prázdny blok v tom prípade nenašiel hľadaný blok a funkcia vráti 0, naopak ak narazí na správny blok vráti 1.

Možné nastavenia projektu

Projekt obsahuje konštanty, ktoré ovplyvňujú správanie algoritmu. Jednou z nich je **CLEAR**, stará sa o to aby sa po rôznych úkonoch ako napríklad alokovaní pamäte, naplnil payload bloku určitým znakom, ktorý nám uľahčí testovanie. Ďalšia konštanta s názvom **ERROR** ovláda zobrazovanie chybových hlášok ako je napríklad, že pamäť

nemá požadované miesto alebo je plná. Posledná a najdôležitejšia je konštanta **REWRITE_POINTER**, ktorá sa stará o to či algoritmus pošle užívateľovi po alokácií payload, ktorý začína od nevyužitého ukazovateľa a týmto sa nám zmení hlavička na celkovú veľkosť 4B.

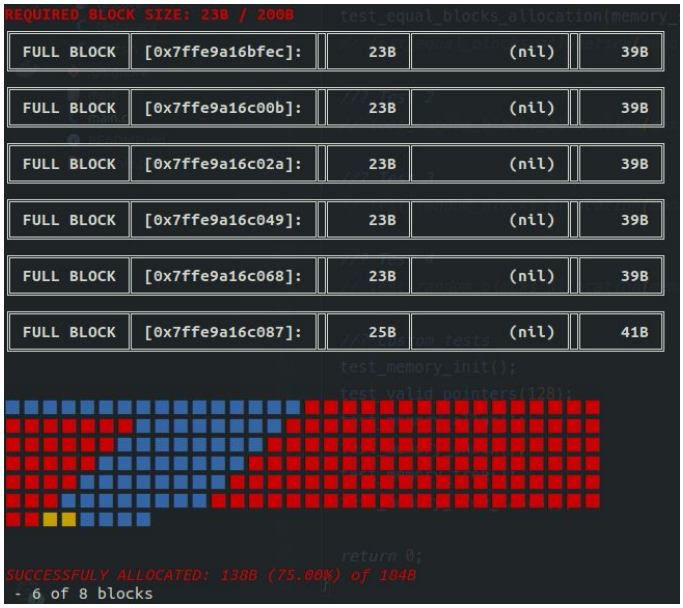
```
#define REWRITE_POINTER
```

Testy

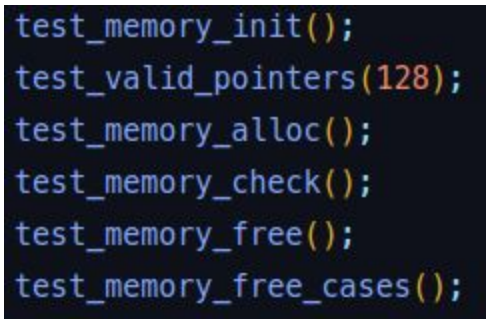
Na správne otestovanie implementácie som spravil funkcie, ktoré obsahujú rôzne prípady správania užívateľa. Okrem tých program obsahuje štyri základné testy na alokovanie so znázornením fragmentácie zo zadania. Niektoré testy sú zakomentované pre lepší prehľad ale pre použitie ich stačí odkomentovať.

Testy zo zadania

Test alokácie blokov pre rôzne veľkosti pamäte vyzerá nasledovne. Na vrchu máme údaj o veľkosti, ktorú sa snažíme alokovať a akú veľkú pamäť máme. Ďalej sú vypísané alokované bloky pamäte a pod tým graficky znázornená pamäť: modrá - hlavičky a päty, červená - alokované miesto v bloku, zelená - voľné miesto v bloku a oranžová vyznačuje bajty, ktoré sa vrátili užívateľovi navyše pretože už sa neoplatilo rozdeliť blok. Pod tým je zoznam voľných blokov a úplne naspodu výpočet koľko percent bajtov sa nám podarilo alokovať oproti ideálnemu riešeniu. V mojom riešení sa mi napríklad podarilo pri 12B blokoch znížiť fragmentáciu pamäte na 25% t.j. alokovách 6 blokov z 8 ideálnych, ako môžete vidieť na obrázku vyššie.



Vlastné testy



Vo vlastných testoch som riešil rôzne okrajové udalosti, ktoré môžu nastať a ako na ne program odpovedá. Každá funkcia má vlastný test, ktorého výstupmi sú hlášky ako OK a FAIL. V teste pre funkciu memory_init riešim hlavne kontrolu argumentov a opakované použitie funkcie. Ako v každom teste aj v teste pre alokovanie pamäte testujem pokus o alokovanie keď ešte nebola vytvorená pamäť. Ďalej sa zisťuje najväčší možný blok pamäte čo je pri veľkosti pamäte 1024B až 1004B a nakoniec alokáciu rôznych dátových typov. V teste pre memory_check sa kontrolujú všetky ukazovatele pamäte a kontroluje sa či alokovaný blok vyhodnotí správne. V memory_free teste ide o uvoľnenie blokov rôznej veľkosti, opakované uvoľňovanie blokov a o ukážku správne zoradených voľných blokov pamäte. V poslednej funkcií je demonštrácia správania sa pamäte pri uvoľňovaní blokov ktoré majú rôznych susedov. Riešia sa tam 4 základné scenáre, pričom vždy stredný blok sa uvoľňuje: full | full | full, free | full | full, full | full | free, free | full | free a dokonca aj scenár navyše, kedy sa uvoľňuje už uvoľnený blok pamäte.

Záver

Mojím zámerom bolo vypracovať problém aby sa jeho fragmentácia a časová zložitosť vyrovnali. Docielil som to použitím hlavičky a päty, ktoré majú malú veľkosť. A tým že využívam nepotrebný ukazovateľ v alokovanom bloku. Myslím že toto riešenie je vhodné či už pri menšej alebo väčšej celkovej veľkosti pamäte, keďže sa dá pracovať aj s blokmi malej veľkosti.