Zadanie 1 – Správca pamäti

**Dokumentácia**

1. **Opis použitého algoritmu**

void memory\_init(void \*ptr, unsigned int size);

Na prvých 4B celkovej pamäte je zapísaná jej veľkosť.

Pointer na začiatok celkovej pamäte je uložený do globálnej premennej.

Je vytvorená prvá hlavička bloku pamäte dĺžky 3B, do ktorej je zapísaná voľná pamäť.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **20** | **0** | **0** | **0** | **13** | **0** | **-128** | **x** | **x** | **x** | **x** | **x** | **x** | **x** | **x** | **x** | **x** | **x** | **x** | **x** |

char \*space;

Hlavička bloku pamäte je definovaná ako bitová štruktúra.

typedef struct header

{

unsigned int size : 23;

unsigned int is\_free : 1;

}HEADER;

Všetky veľkosti sú definované ako konštanty a závisí od nich aké maximálne dĺžky blokov je možné alokovať!:

#define BLOCK\_HEAD\_SIZE 3

#define SPACE\_HEAD\_SIZE 4

#define MIN\_BLOCK\_SIZE 4

void \*memory\_alloc(unsigned int size);

Pri alokácii sa prehľadávajú bloky pamäte od začiatku a blok sa vyberá stratégiou *first fit*. Ak sa prechádza cez voľný blok skúsi sa spojiť s nasledujúcim.

Funkcia **tryJointNext{** Ak existuje nasledujúci blok a je tiež voľný – aktuálny blok sa navýši o veľkosť a hlavičku nasledujúceho.**}**

Ak sa nájde vhodný blok (voľný a požadovanej alebo väčšej veľkosti) použije sa. *Tento spôsob je rýchlejší ako „najlepší vhodný blok“ avšak môže priniesť horšie využitie pamäte.*

Ak je vhodné ho rozdeliť (t.j. vznikne nový blok veľkosti min. 4B ) – rozdelí sa. Rozdelenie znamená vytvorenie novej hlavičky – voľná a s ostávajúcou veľkosťou.

Pôvodná hlavička sa zmenší na požadovanú veľkosť.

Ak sa nepodarilo alokovať pamäť, funkcia vracia NULL.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **20** | **0** | **0** | **0** | **5** | **0** | **0** | **x** | **x** | **x** | **x** | **x** | **5** | **0** | **-128** | **x** | **x** | **x** | **x** | **x** |

int \*p;

int memory\_free(void \*valid\_ptr);

Najskôr sa skusí uvoľňovaný blok spojiť s nasledujúcim.

Funkcia **tryJointNext{** Ak existuje nasledujúci blok a je tiež voľný – uvolňovaný blok sa navýši o veľkosť a hlavičku nasledujúceho.**}**

Nastaví sa bit na hlavičke is\_free=TRUE;

Funkcia vráti 0, ak sa podarilo uvoľniť blok pamäti.

int memory\_check(void \*ptr);

Pri kontrole platnosti smerníka, prechádza funkcia smerníkmi na bloky pamäte a ak medzi nimi nájde hľadaný blok – a ten je pridelený vráti 1. Inak vráti 0.

1. **Odhad priestorovej zložitosti**

Počet pomocných premenných vo funkciách je fixný, nezávisí od veľkosti alokovanej/ uvoľňovanej pamäte. Priestorová zložitosť algoritmu je preto konštantná.

Ak chceme vyjadriť zložitosť algoritmu vzhľadom na pamäť, ktorú spravujeme, vychádzame z veľkosti hlavičiek. Algoritmus potrebuje vždy 4B na zapísanie celkovej dĺžky a 3B pre každú alokáciu. Táto zložitosť preto závisí lineárne od počtu alokácií.

1. **Odhad časovej zložitosti**

Inicializácia spravovanej pamäte a Uvoľnovanie pamäte

Program nezávisle od dĺžky spravovanej pamäte vykoná pri inicializácii a uvoľňovaní určitý fixný počet operácií. Časová zložitosť je preto konštantná. **.**

Alokácia pamäte a Kontrola smerníka

Obidve funkcie pri svojom vykonávaní prechádzajú pamäťou len raz, vždy od začiatku a v najhoršom prípade až po koniec. Časová zložitosť preto závisí lineárne od dĺžky spravovanej pamäte. .

Odhadnutý čas vykonávania jednotlivých funkcií sme testovali aj programovo. Počet alokácií sme postupne zvyšovali.

**for**(i=1000; i<=81000; i+=20000)

timeTester(i);

Vypisovali sme čas po vykonaní jednotlivých funkcií.

**void** timeTester(**int** pocet)

{

...

printf(**"%.3lf\n"**,(**double**)clock()/ 1000.0); *//zaciatok init*

memory\_init(region,size);

printf(**"%.3lf\n"**,(**double**)clock()/ 1000.0); *//zaciatok alloc*

**for**(i=0; i<pocet; i++)

**if**((p[i] = (**char**\*)memory\_alloc(SIZE\_BLOCK))==NULL)

printf(**"chyba %d. alokacie"**,i);

printf(**"%.3lf\n"**,(**double**)clock()/ 1000.0); *//zaciatok free*

**for**(i=pocet-1; i>=0; i--

**if**(memory\_free(p[i]))

printf(**"chyba %d. uvolnenia"**,i);

printf(**"%.3lf\n"**,(**double**)clock()/ 1000.0); *//koniec testu*

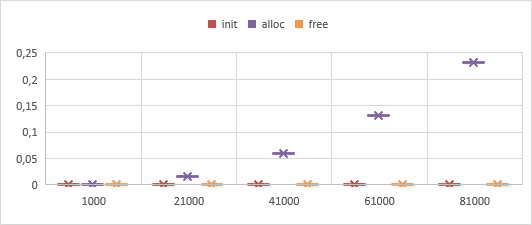
...

}

Po spriemerovaní viacerých pokusov vznikla nasledujúca tabuľka.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **N blokov** | | | | |
|  | | **1000** | **21000** | **41000** | **61000** | **81000** |
| **init** | | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| **alloc N- blokov** | | 0.000 | 1.526 | 5.771 | 12.752 | 22.532 |
| **free N- blokov** | | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |

Pri znázornení údajov do grafu vidíme, že čas vykonávania funkcií init a free nezávisí od počtu blokov. Naopak čas alokácie N blokov pamäte rastie s 2. mocninou. To znamená, že alokácia jedného bloku rastie s 1. mocninou = lineárna.



1. **Testovač**

K testovaniu korektnosti implementácie sme použili 3 rôzne kontrolné funkcie. Veľkosť celkovej pamäte bola vždy nastavená tak, aby funkcie využívali celú jej dĺžku.

1. Kontrola funkčnosti spájania blokov:
   1. Alokácia 1 bloku na celú dĺžku a uvoľnenie.
   2. Alokácia 3 blokov a uvoľnenie 1.+3.+2..
   3. Alokácia 1 bloku na celú dĺžku a uvoľnenie.
   4. Alokácia 3 blokov a uvoľnenie 3.+2.+1.
   5. Alokácia 1 bloku na celú dĺžku a uvoľnenie.
2. Kontrola opakovaného prideľovania a uvoľňovania blokov: Prideľovanie rovnakých blokov veľkosti 8B, postupne uvoľňovanie a znova prideľovanie
   1. Alokácia 10 blokov.
   2. for each n in {1,2,…,10}

{

Uvoľnenie prvých n-blokov.

Alokácia prvých n-blokov.

Kontrola platnosti smerníkov na alokované bloky.

}

* 1. Uvoľnenie 10 blokov.

1. Kontrola funkčnosti pri zápise do pamäte: Prideľovanie nerovnakých blokov rôznych veľkostí, kontrola obsahu a uvoľňovanie
   1. Uloženie veľkosti voľnej pamäte „pred“.
   2. for each s in {1024,32,8,128,16,64,2048,512,256}

{

Alokácia bloku veľkosti s.

Zapísanie na každý bajt bloku poradové číslo alokácie.

}

* 1. Uvoľňovanie blokov od začiatku:

for each s in {1024,32,8,128,16,64,2048,512,256}

{

Kontrola obsahu s bajtov v bloku.

Obsah bajtu sa musí rovnať poradovému číslu alokácie.

Uvoľnenie bloku.

}

* 1. Porovnanie Uloženie voľnej pamäte „pred“ a „po“.