Ondrej Harnúšek, ID: 79545
utorok 16:00

ZADANIE 1 – SPRÁVCA PAMÄTI

DOKUMENTÁCIA

I. Prehľad dátových štruktúr:

```
typedef struct metadata//hlavička a päta pre všetky bloky pamäte
{
    unsigned int capacity : 31;
    unsigned int is_free : 1;
}METADATA;

typedef struct free_block//hlavička pre voľné bloky pamäte
{
    unsigned int capacity : 31;
    unsigned int is_free : 1;
    struct free_block *next; //smerník na nasledujúci
    struct free_block *prev; //smerník na predošlý voľný blok
}FREE_BLOCK;

typedef struct preamble//preambula na začiatku celkovej pamäte
{
    FREE_BLOCK *root;//prvý voľný blok
    unsigned int size; //veľkosť celkovej pamäte
}PREAMBLE;
```

Pri alokácii sa smerníky na nasledujúci a predchádzajúci blok prepisujú!

II. Opis použitého algoritmu – Metóda explicitných zoznamov

void memory_init(void *ptr, unsigned int size);
Na začiatku celkovej pamäte sa nachádza preambula. Na prvých 4B je odkaz na prvý voľný blok – **ROOT** a ďalších 4B je zapísaná celková veľkosť - **SIZE**. Pointer na začiatok celkovej pamäte je uložený do globálnej premennej `*memory`.
Pri inicializácii sa vytvorí prvý voľný blok. Každý blok začína hlavičkou a končí pätou typu– **METADATA - hlavička a päta**. Minimálna kapacita bloku je aspoň `2*sizeof(FREE_BLOCK*)`, čo sú odkazy na **ĎALŠÍ** a **PREDCHÁDZAJÚCI** voľný blok.

V príklade na obrázku je stav pamäte po `p=memory_init(ptr,24);`
(`memory->root->`**NEXT** sa rovná NULL aj `memory->root->`**PREV** sa rovná NULL)

36	-2	97	0	24	0	0	0	8	0	0	-128	x	x	x	x	x	x	x	x	8	0	0	-128

PREAMBLE *memory; //jediná globálna premenná programu

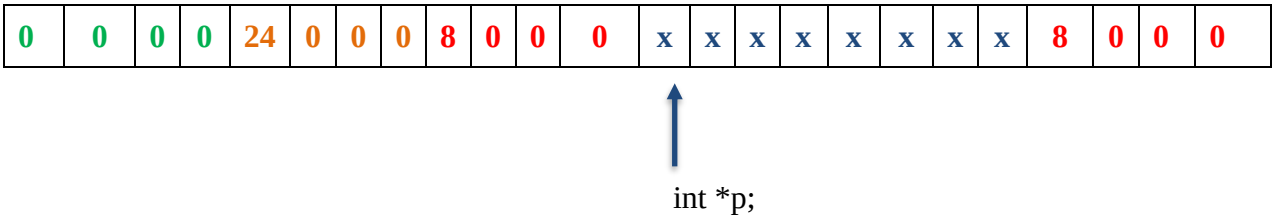
Ondrej Harnúšek, ID: 79545
utorok 16:00

void *memory_alloc(unsigned int size);
Pri alokácii sa prehľadávajú **len voľné bloky pamäte** od začiatku a blok sa vyberá stratégiou *first fit*..

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 kapacity min. 2*veľkosť dvoch smerníkov) – rozdelí sa. Rozdelenie znamená vytvorenie novej hlavičky a päty–s ostávajúcou kapacitov.

V príklade na obrázku celková pamäť pozostáva z jedného **prideleného bloku 8B**. Odkaz na prvý voľný blok je **NULL**.



int memory_free(void *valid_ptr);
Nastaví sa bit na hlavičke a päte na `is_free=TRUE;`
Ak sú susedné bloky voľné spojí sa s nimi a zaradí na začiatok zoznamu voľných blokov
Funkcia vráti 0, ak sa podarilo uvoľniť blok pamäti.

int memory_check(void *ptr);
Pri kontrole platnosti smerníka, kontroluje funkcia či je smerník z oblasti celkovej pamäte. Následne porovnáva hlavičku a pätu a kontroluje či je blok voľný alebo alokovaný, t.j. `is_free=FALSE`.

I. Odhad priestorovej zložitosti

Počet pomocných premenných vo funkciách je fixný, nezávisí od veľkosti alokovanej/ uvoľňovanej pamäte. Priestorová zložitosť algoritmu je preto konštantná. $\boldsymbol{O(1)}$

Ak chceme vyjadriť zložitosť algoritmu vzhľadom na pamäť, ktorú spravujeme, vychádzame z veľkosti hlavičiek. Algoritmus potrebuje vždy 4B na smerník na prvý voľný blok (root), 4B na zapísanie celkovej dĺžky. Ďalej 8B pre každý voľný blok a 4B pre každý pridelený blok. Táto zložitosť preto závisí lineárne od počtu blokov. $O(n)$

II. Odhad časovej zložitosti

Inicializácia spravovanej pamäte
Funkcia nezávisle od dĺžky spravovanej pamäte vykoná pri inicializácii a uvoľňovaní určitý fixný počet operácií. Časová zložitosť je preto konštantná. $\boldsymbol{O(1)}$.

Ondrej Harnúšek, ID: 79545
utorok 16:00

Uvoľnovanie pamäte
Funkcia vykoná konštantný počet operácií a zaradí uvoľnený blok na začiatok zoznamu voľných blokov (LIFO politika), preto . $\boldsymbol{O(1)}$.
Kontrola smerníka
Funkcia vykoná vždy konštantný počet operácii, preto . $\boldsymbol{O(1)}$.
Alokácia pamäte
Funkcia pri svojom vykonávaní prechádza len voľnými blokmi, vždy od začiatku a v najhoršom prípade až po koniec. Časová zložitosť preto závisí lineárne od počtu voľných blokov. $\boldsymbol{O(n)}$.

III. 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. Základný test- alokácia, kontrola, uvoľnenie:
   a. Alokácia 3 blokov rovnakej dĺžky a uloženie čísiel ID.
   b. Memory_check alokovaných blokov
   c. Kontrola uložených ID.
   d. Uvoľnenie blokov.

2. Kontrola opakovaného prideľovania a uvoľňovania blokov: Prideľovanie rovnakých blokov veľkosti 8B, postupne uvoľňovanie a znova prideľovanie
   a. Alokácia 10 blokov.
   b. for each n in {1,2,…,10}
      {
         Uvoľnenie prvých n-blokov.
         Alokácia prvých n-blokov.
         Kontrola platnosti smerníkov na alokované bloky.
      }
   c. Uvoľnenie 10 blokov.

3. Kontrola funkčnosti pri zápise do pamäte: Prideľovanie nerovnakých blokov rôznych veľkostí, kontrola obsahu a uvoľňovanie
   a. 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.
      }
   b. 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.
      }