FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
SLOVENSKÁ TECHNICKÁ UNIVERZITA
Ilkovičova 2, 842 16 Bratislava 4

2020/2021

Datové štruktúry a algoritmy

Zadanie č.1

Cvičiaci: Mgr. Martin Sabo, PhD.
Čas cvičení: Utorok 18:00 – 19:50

Vypracovala: Monika Zjavková
AIS ID: 105345

Obsah

1. Úvod

1.1. Opis Algoritmu

Zadanie vlastnej implementácie alokovania pamäte je riešené explicitným zoznamom. Pri hľadaní vhodného voľného bloku sú prechádzané len voľné bloky, v porovnaní s implicitným zoznamom, kde je potrebné prejsť každý blok, čím sa zrýchli vykonanie programu.

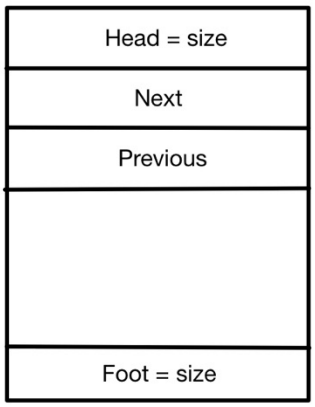
Zoznam voľných blokov sa prehľadáva pomocou ich vzdialenosti v pamäti od začiatku regiónu. Táto vzdialenosť je uložená ako integer. Riešenie cez spájaný zoznam so smerníkmi by bol jednoduchší, no na uloženie smerníka je potrebných 8 bajtov, takže by na uloženie voľného bloku bolo potrebných až 24 bajtov. S použitím vzdialeností od začiatku regiónu je spotrebovaných 20 bajtov.

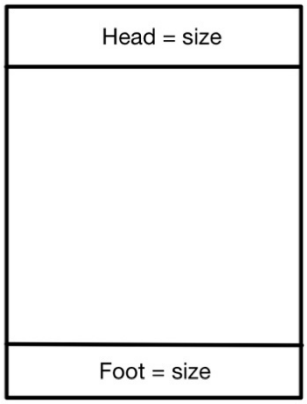
Priestorová zložitosť hlavičky a pätičky označujúcej začiatok a koniec pamäte je $O(1)$, keďže je potrebné ich vytvoriť len raz, pre hlavičky a pätičky jednotlivých blokov rastie s $O(n)$ pre počet vytvorených blokov.

1.2. Zobrazenie pamäte

Na začiatku regiónu sa nachádza integer, ktorý označuje offset prvého voľného bloku. Na konci sa nachádza -1, na označenie konca pamäte. Každý alokovaný blok je zobrazený pomocou štruktúry a obsahuje hlavičku obsahujúcu veľkosť bloku a pätičku, ktorá sa už nenachádza v štruktúre, keďže je potrebné, aby označovala koniec bloku. Voľný blok okrem toho obsahuje aj 2 čísla navyše pre ďalší a predchádzajúci blok.

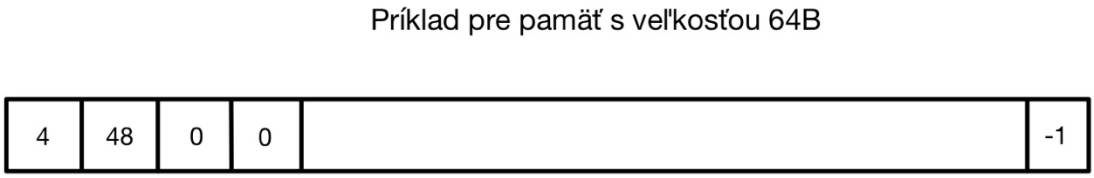
Alokovanie bloku je zaznačený záporným znamienkom, takže sú ušetrené bajty, ktoré by bolo potrebné použiť na označenie obsadenosti bloku ako napríklad pri označení 1/0 alebo písmenom.





1.3. Inicializácia pamäte

Pred spustením funkcie na alokáciu pamäte alebo akejkoľvek inej funkcie je najskôr potrebné inicializovať pamäť. V tejto funkcii sa nastaví začiatok a koniec pamäte. Taktiež sa vytvorí prvý voľný blok s veľkosťou, ktorá je programu k dispozícii. Keďže to je jediný voľný blok, offset pre nasledujúci a predchádzajúci blok je 0.



1.4. Memory Check

```
void *head = memory+sizeof(int);
while (head + sizeof(int) != ptr){

    //ak je v hlavičke -1, znamená to, že funkcia je na konci a nenašla zhodu
    if( (*((int *) (head))) == -1){
        return 0;
    }

    head = head + abs(*((int*)head)) + 2*sizeof(int);
}
```

Funkcia kontroluje, či smerník zadaný ako parameter ukazuje na alokovaný blok pamäte. Overenie prebieha na základe prechádzania všetkých blokov pamäte a hľadá zhodu. Časová zložitosť je preto lineárna $O(n)$. Efektívnejší spôsob by bol porovnanie hlavička a päty, ale v tom prípade by nebolo možné zistiť, či smerník nie je mimo vyhradenej pamäte.

2. Alokovanie pamäte

2.1. Nájdenie voľného bloku

Pri prehľadávaní je použitý algoritmus first fit. Zoznam voľných blokov je prehľadávaný v cykle, v ktorom sa porovnáva veľkosť voľného bloku s požadovanou veľkosťou na vytvorenie ďalšieho bloku a zvolí sa prvý blok, ktorý tejto požiadavke vyhovuje. Časová zložitosť tohoto algoritmu je $O(n)$, pri čom n označuje počet všetkých voľných blokov, zoznam voľných blokov je prehľadaný nanajvýš raz.

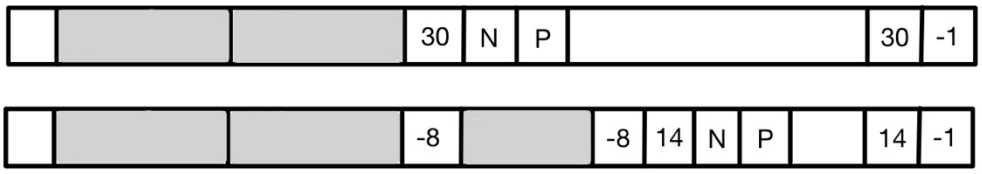
Ak vhodný blok nenájde funkcia vráti NULL. Pri nájdení sa podľa zadaných podmienok program rozhodne, či je potrebné blok rozdeliť alebo sa alokuje celý.

2.2. Rozdelenie bloku

```
new_offset = size + 2*(sizeof(int));                                    //vypočíta sa offset nového bloku
 new = free + new_offset;                                               //vytvorenie nového voľného bloku
 new->next = blk->next;                                  //offset ostane rovnaký ako pôvodného voľného bloku
 new->prev = blk->prev;
 new->head = blk->head-new_offset;                                      //hlavička nového voľného bloku
 *((int*) (free + new_offset + new->head + sizeof(int))) = new->head;   //pätička nového voľného bloku
```

V prípade, že vybraný voľný blok je väčší ako potrebná pamäť, alokuje sa iba nevyhnutná časť a zo zvyšného bloku sa vytvorí nový, ktorého veľkosť sa vypočíta ako pôvodná veľkosť – 2*sizeof(int), pretože je nutné odpočítať aj veľkosť, ktorú zaberie novo vytvorený hlavička nového bloku a päta alokovaného.

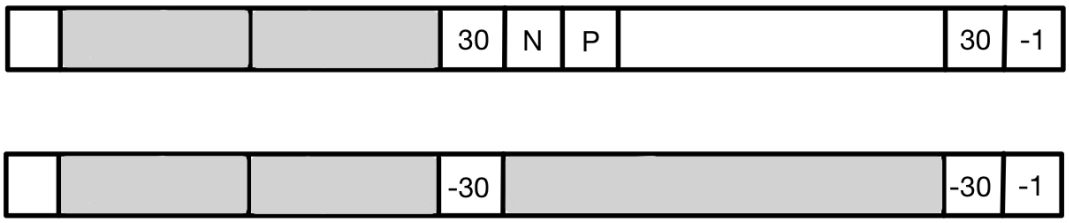
Príklad rozdelenia bloku pri alokovaní 8 bajtov do bloku s veľkosťou 30 bajtov



2.3. Alokovanie celého bloku

Ak je veľkosť bloku rovnaká ako požadovaná veľkosť, blok sa vymaže zo zoznamu pre voľné bloky a hlavička s pätičkou sa nastavia na zápornú hodnotu. Celý blok je alokovaný aj v prípade, že sa do zvyšnej časti nezmestí nová hlavička, päta a offset pre ďalší a predchádzajúci blok, aby v pamäti nevznikali medzery, ku ktorým už nebude možné sa dostať, keďže program si drží informáciu iba o všetkých voľných blokoch.

Príklad rozdelenia bloku pri alokovaní 30 bajtov do bloku s veľkosťou 30 bajtov



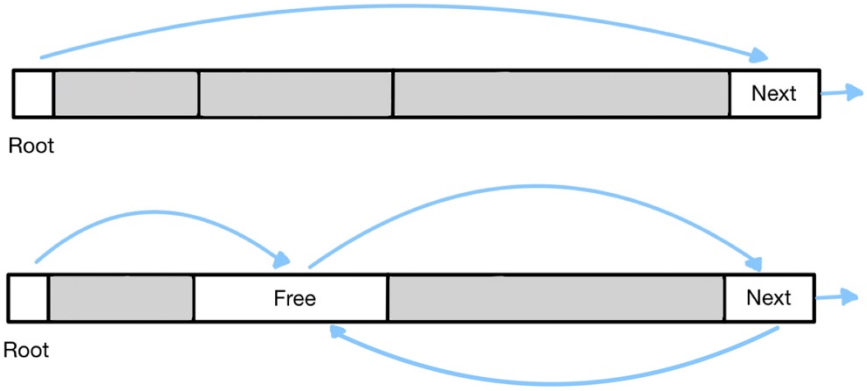
3. Uvoľňovanie pamäte

3.1. Uvoľnenie bloku

Funkcia uvoľňovanie funguje na princípe algoritmu LIFO (Last in first out), kde sa každý nový voľný blok uloží na začiatok zoznamu. Aby nebola pamäť trvalo rozkúskovaná, je potrebné riešiť spájanie blokov podľa toho, či sa vedľa uvoľneného bloku nachádza ešte iný voľný. Vnútro nie je potrebné prepisovať a upravovať, keďže sa predpokladá, že táto pamäť, ktorá je dostupná, bude prepísaná používateľom.

Poradie voľných blokov v zozname nemusí zodpovedať fyzickému usporiadaniu blokov, preto nie je možné iba zlúčiť bloky na základe zoznamu. Preto funkcia používa pätičky a hlavičky na nájdenie bloku zľava a sprava. Najskôr sa uvoľní požadovaný blok a potom sa rieši spájanie pre jednoduchší prístup k informáciám o bloku.

Časová zložitosť je konštantná $O(n)$. keďže kontrolujeme iba susedné bloky a nie je potrebné prehľadávať celý zoznam.



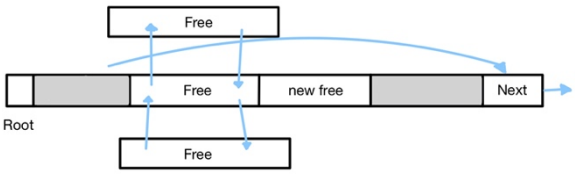
3.2. Spájanie s voľným blokom sprava

Na spájanie sprava je potrebné zistiť, či ďalší blok ma kladnú alebo zápornú hlavičku. V prípade, že je kladná – blok je voľný, sa upravia informácie v práve uvoľnenom bloku. Nie je potrebné meniť offset pre ďalšie bloky, keďže toto bolo spravené v prvom kroku, iba zmeniť veľkosť hlavičky a pätičky. Keďže sa kontroluje, či je hodnota kladná alebo záporná, špecifický prípad predstavuje začiatok pamäte, kde môže označiť offset k prvému voľnému bloku ako nealokovaný blok, preto je tam potrebná podmienka navyše.

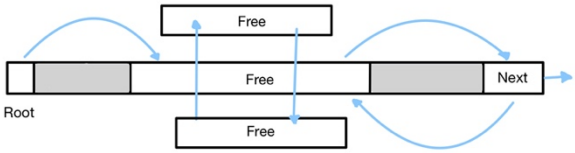
```
freed->head += merge_next->head+2*sizeof(int);                    //hlavička nového bloku
    *((int*)(valid_ptr+freed->head)) = freed->head;               //nová pätička
    first_free() = freed_offset;                                  //presunutie bloku na začiatok zoznamu
```

Upravenie smerníkov:

pred



potom



3.3. Spájanie s voľným blokom zľava

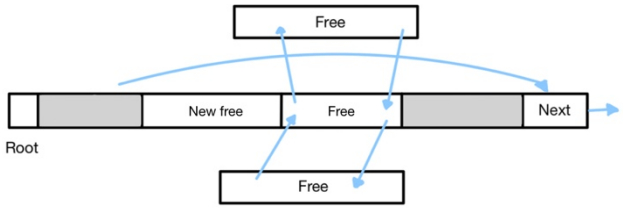
Pri spájaní s voľným blokom zľava je tiež potrebná podmienka, ktorá zistí, či je to voľný blok alebo len začiatok pamäte, ktorý tiež obsahu

je kladné číslo – offset. Tu je potrebné zmeniť okrem veľkosti a hlavičky aj offset pre ďalší a predchádzajúci blok, keďže nová hlavička sa nachádza v predchádzajúcom voľnom bloku.

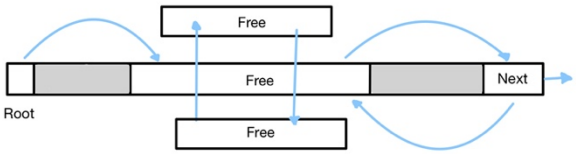
```
merge_prev->head += freed->head + 2*sizeof(int);                  //nastavenie novej hlavičky
*((int*)(valid_ptr+free_head(valid_ptr))) = merge_prev->head;     //nastavenie novej pätičky
first_free() = freed_offset - 2*sizeof(int)-prev_block(valid_ptr); //nový offset pre prvý voľný
```

Upravenie smerníkov:

pred



potom



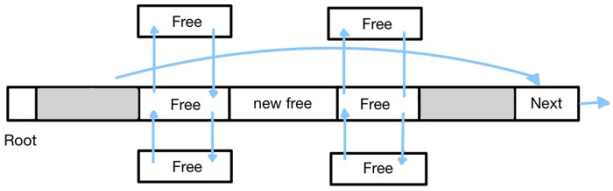
3.4. Spájanie s blokmi z oboch strán

Pri spájaní z oboch strán je nutné riešiť rovnakú podmienku ako pri spájaní zľava a sprava – či voľný blok naľavo nie je len offset ku prvému voľnému bloku. Ako nová hlavička sa nastaví v predchádzajúcom voľnom bloku a pätička v nasledujúcom voľnom bloku. Následne sa oba spojené bloky vymažú z pôvodného miesta v zozname a zaznačí sa ich nová veľkosť do hlavičky a päty.

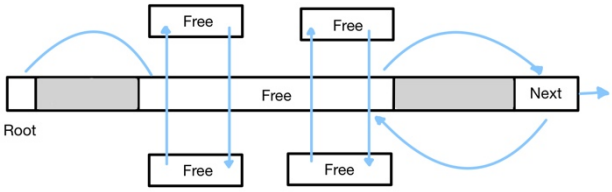
```
//hlavička nového bloku
merge_prev->head = merge_prev->head + merge_next->head + freed->head + 4*(sizeof(int));
//pätička nového bloku
*((int*)(valid_ptr+merge_next->head+freed->head+2*sizeof(int))) = merge_prev->head;
//nový offset
first_free() = freed_offset - 2*sizeof(int)-prev_block(valid_ptr);
```

Upravenie smerníkov:

pred



potom



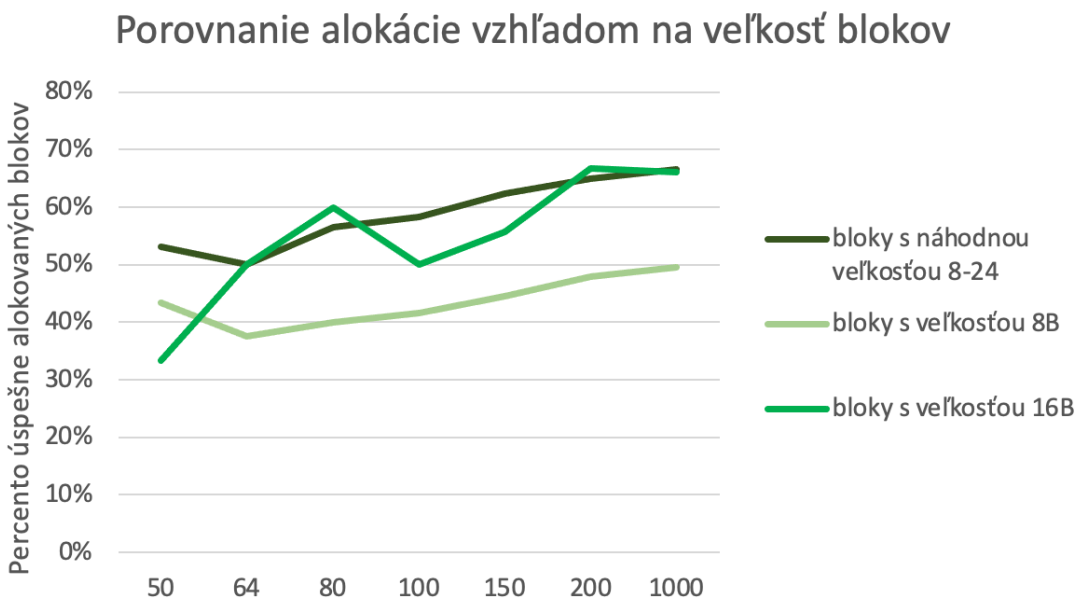
4. Testovanie

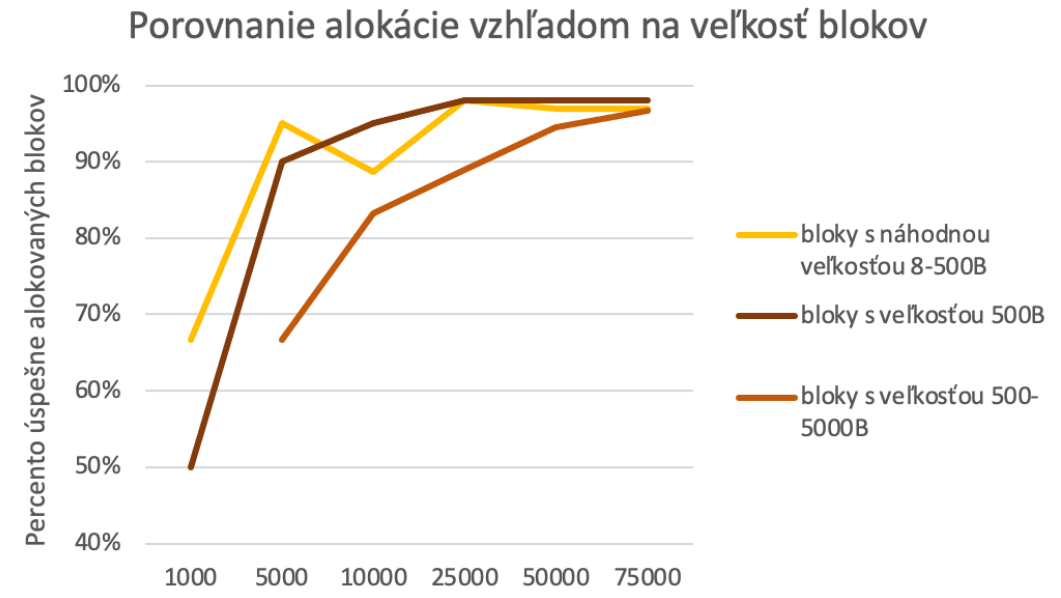
Testovanie funkčnosti prebiehalo zvlášť pre alokovanie a zvlášť pre uvoľňovanie pamäte. Vo funkcii pre testovanie alokovania sa zisťovala efektivita programu, kde sa porovnávali výsledky ideálneho riešenia bez vnútornej a vonkajšej fragmentácie s mojou implementáciou.

4.1. Testovanie alokovanie pamäte

Test skúša rôzne scenáre pre alokovanie rovnaké veľké bloky alebo náhodne zvolenú veľkosť v určitom intervale, tiež testuje aj alokovanie pamäte do malých celkových blokov ako je 50, 100 a 200 bajtov alebo do väčších pamätí ako je 100 000 bajtov. Alokovanie blokov prebieha v cykle dovtedy, kým v ideálnom riešení nebude veľkosť voľnej pamäte menšia ako najmenší možný blok podľa scenára.

Efektivita sa zvyšuje s narastajúcou veľkosťou celkovej pamäte a veľkosťou blokov, keďže tam hlavička a päta predstavujú menšie percento z celkovej pamäte. Implementáciu by bolo možné zefektívniť zaznačovaním voľných blokov, ktoré majú menšiu veľkosť ako minimálna veľkosť pre voľný blok - 20 bajtov, ktoré vznikajú pri rozdelení bloku alebo použiť implementáciu iba s hlavičkou.





4.2. Testovanie uvoľňovania pamäte

Vo funkcii na testovanie pamäte sa najskôr alokuje pamäť, pokým nebude plná a v rôznych scenároch sa testuje, či funguje spájanie voľných blokov zľava – pamäť sa postupne uvoľňuje od začiatku až po koniec, ďalší test overuje testovanie spájania blokov sprava – pamäť sa uvoľňuje za sebou od konca po začiatok, čiže v opačnom poradí ako bola alokovaná, posledný test skúša uvoľňovanie v náhodnom poradí na otestovanie všetkých možných scenároch.

Na konci funkcie sa porovnáva hlavička, ktorá vznikla pri inicializácii pamäte, kedy bol len jeden voľný blok s hlavičkou po skončení uvoľňovania. V prípade, že uvoľňovanie prebehlo správne, mali by byť rovnaké.