Matej Delinčák, ID: 102895

Zadanie 1 – Správca pamäti

Použitie metódy oddelených zoznamov voľných blokov

Pre réžiu na spravovanie pamäti som vybral metódy segregovaných zoznamov voľných blokov. Tieto zoznamy držia offset na ďalší voľný blok konkrétnej veľkosti. Prvý zoznam drží bloky veľkosti 8 až 64 bajtov (kvôli ušetreniu réžie pre malé pamäte), druhý bloky 64 až 128, atď. Tie veľkosti sa menia s mocninou čísla dva. Následne som ešte do implementoval dynamické hlavičky. To znamená, že na základe veľkosti pamäti, sa vo funkcii *memory_init* rozhodne, či sa budú používať hlavičky veľkosti char, short, int prípadne long. Táto implementácia znížila réžiu pamäte razantne najmä pri pamätiach s malou veľkosťou.

Pre nájdenie voľného bloku som použil algoritmus best fit, ktorý prejde príslušný zoznam a vyberie z neho taký blok, ktorý najlepšie pasuje na požadovanú veľkosť. Týmto som zabezpečil, aby som nerozbíjal zbytočne veľké bloky, ktoré môžu byť použité neskôr. Ak sa v danom zozname nenájde vhodný blok požadovanej veľkosti, prehľadám najbližší zoznam väčších blokov.

Stav pamäti po memory_init

Na začiatku poľa(1) si ukladám aktuálny mód (aké veľké hlavičky používam):

1 – **char** (pamäte od 50 do 127)

2 – **short** (pamäte od 128 do 32766)

4 – **int** (pamäte od 32767 do veľkosti intu)

8 – **long long** (pre pamäte väčšie ako veľkosť intu)

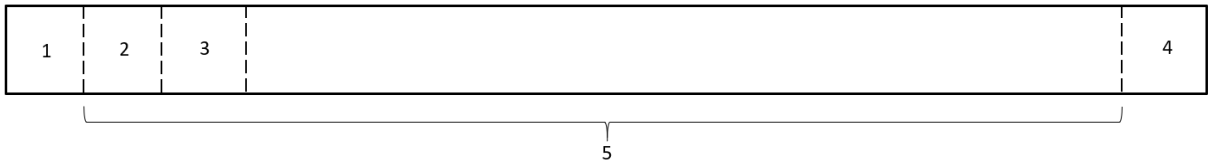
Na pozícií dva si ukladám počet oddelených zoznamov. Tento počet vyrátam ako (vo funkcií *blockNumber*):

$$ak\ je\ pamäť\ väčšia\ ako\ 63B\colon\ \lfloor \log_2(veľkosť) \rfloor - 2,$$

$$ak\ nie\ tak\ používam\ len\ jeden\ zoznam$$

Na čísle 3 sa nachádzajú začiatky samotných zoznamov a na čísle 6 sa nachádza nula, ktorá symbolizuje koniec pamäte, ktorú používam. Bloky označené číslami 4 a 5 sú hlavička a päta prvého voľného bloku.

Všeobecná schéma voľného bloku



Ako bolo vyššie spomínané, bloky s číslami 1 a 2 sú hlavička a päta. Tieto dva bloky majú veľkosť podľa nastaveného módu a držia informáciu, koľko miesta je medzi nimi (5). Toto miesto bude sprístupnené užívateľovi pri alokovaní.

V bloku 2 sa nachádza smerník, resp. offset, na ďalší voľný blok. Ak taký už nie je, je tam hodnota -1. Podobne v bloku 3 sa nachádza smerník na predchádzajúci voľný blok, resp. na začiatok príslušného zoznamu. Tieto smerníky budú prepísané užívateľom, pretože pri alokovaní tohto bloku sú už zbytočné. Týmto som znížil réžiu o 8 bajtov, keďže tieto offsety nemusím držať v hlavičke.

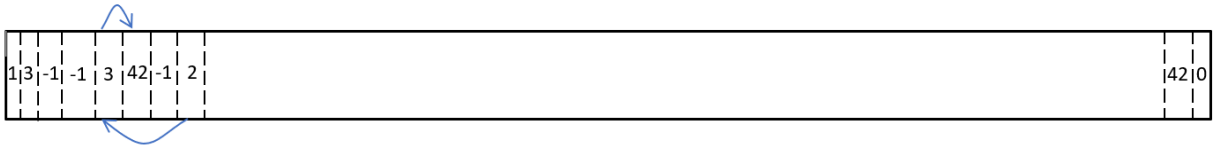
Funkcia memory_alloc

Funkcia zavolá algoritmus *bestFit*, ktorý vráti blok, s ktorým sa má pracovať. Ak je tento blok možné rozdeliť, tak tato funkcia zavolá funkciu *split*. Ak ho nie je možné rozdeliť, tak ho alokuje celý. To znamená, že užívateľ môže dostáť aj viac pamäte ako požadoval. Toto sa nazýva vnútorná fragmentácia a v mojej implementácii môže nadobúdať hodnotu z intervalu (0 ; 2*MÓD + 8).

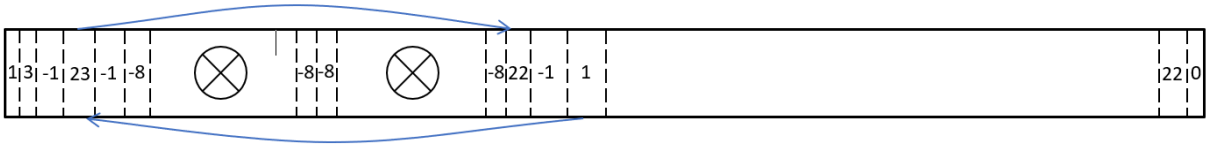
```
if (readFromArr(act) >= size + 2* TYPE + 8)
```

Podmienka, na základe ktorej sa rozhoduje, či sa blok rozdelí alebo nie

Príklad stavu pamäte veľkosti 50 pred a po alokovaní dvoch blokov veľkosti 8

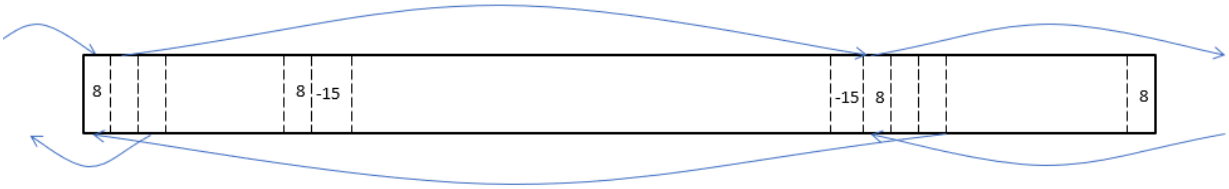


Na obrázku hore môžeme vidieť pamäť v stave po *memory_init.* Kde tretí zoznam ukazuje na jediný voľný blok. Následne zavoláme dva-krát *memory_alloc(8).* Pôvodný blok s veľkosťou 42 najprv odstránim z príslušného zoznamu a potom z neho vyčlením blok o veľkosti 8. Tento nový blok vrátim užívateľovi a zo zvyšnej pamäti vytvorím nový blok, ktorý pridám znova do jemu príslušnému zoznamu. Toto isté zopakuj aj druhý-krát. Po týchto dvoch volaniach pamäť vyzerá nasledovne (obrázok dole).

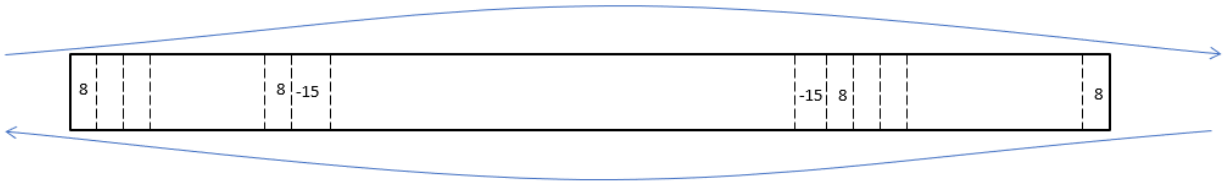


Funkcia memory_free

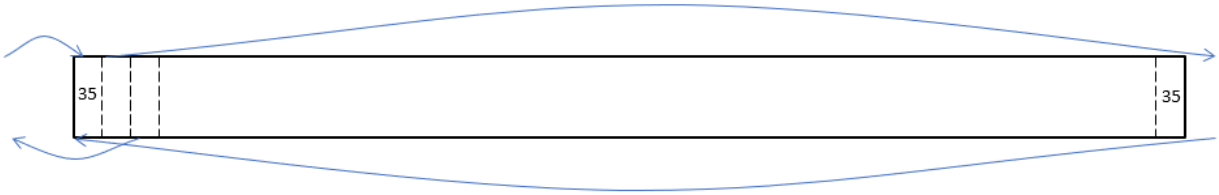
Funkcia *memory_*free dostane ako parameter smerník od užívateľa, ktorý bol vrátený funkciou *memory_alloc*. Moja funkcia nastaví tento blok na znova použiteľný (veľkosť jeho hlavičky bude kladná) a predtým, než ho zapíše do správneho zoznamu, pozrie sa vpravo a vľavo od tohto bloku. Ak sa tam nachádza ďalší voľný blok, tieto dva bloky sa spoja do jedného. Najskôr všetky bloky, ktoré idem spájať ,odstránim z ich zoznamov. Potom ich spojím a až nakoniec ako jeden celý blok, ich pridám do príslušného zoznamu. Malá ukážka, kde vpravo aj vľavo od uvoľňovaného bloku sú voľné bloky.



Teraz odstránim oba bloky z ich zoznamov.



Spojím ako jeden blok a vložím do príslušného zoznamu.



Rozhodovacia podmienka, ktoré dva bloky spojiť:

```
if ((readFromArr( paOffset: act - TYPE) != 0)&&(act > *(memory-1)* TYPE))
{
    if (readFromArr( paOffset: PREVIOUS_BLOCK(act)) > 0)
        act = mergeBlocks( first: PREVIOUS_BLOCK(act), act);
}
//ak je blok, ZA uvolnovanym blokom, volny, tak ich mergni
if (readFromArr(next) != 0)
{
    if (readFromArr(next) > 0){
        deleteBlock(next);
        act = mergeBlocks(act, next);
    }

}
```

Funkcia memory_check

Funkcia sa pozrie na najbližšiu hlavičku a pätu. Ak sa čísla zhodujú a sú kladne, tak to znamená, že tento blok bol uvoľnený. Je veľmi malá pravdepodobnosť, že niekde v pamäti, by sa nachádzali dve rovnaké čísla, práve v tomto rozostupe.

Pomocné funkcie pri implementácií:

void writeToArr(long long paOffset, long long paVal)

- Zápis do pamäte na základe módu. Táto funkcia bola vytvorená kvôli dynamickým hlavičkám.

long long readFromArr(long long paOffset)

- Podobná funkcia ako *writeToArr,* len pre čítanie z pamäte

char blockNumber(long long paSize)

- Funkcia vráti poradové číslo príslušného zoznamu

void deleteBlock(long long act)

- Funkcia vymaže blok z jeho prideleného zoznamu

void insertBlock(long long act)

- Funkcia vloží blok do jeho správneho zoznamu

long long mergeBlocks(long long first, long long second)

- Funkcia vymaže spojí dva prázdne bloky, ale nepridá ich nikam

long long bestFit(long long size)

- Funkcia nájde najlepší blok podľa veľkosti tak, aby sa použili menšie bloky ako väčšie. Najskôr prejde príslušný zoznam a ak nenájde vhodný blok, prejde aj zoznam s väčšími blokmi.

```
while (readFromArr( paOffset: NEXT_P(act)) != -1)
{
    //chod na dalsi blok v zozname
    act = readFromArr( paOffset: NEXT_P(act));
    if (readFromArr(act) < size) continue;
    if (readFromArr(best) < size) best = act;
    if (readFromArr(act) == size) return act;
    if (readFromArr(act) - size < readFromArr(best) - size) best = act;
}
```

long long split(long long act, unsigned int size)

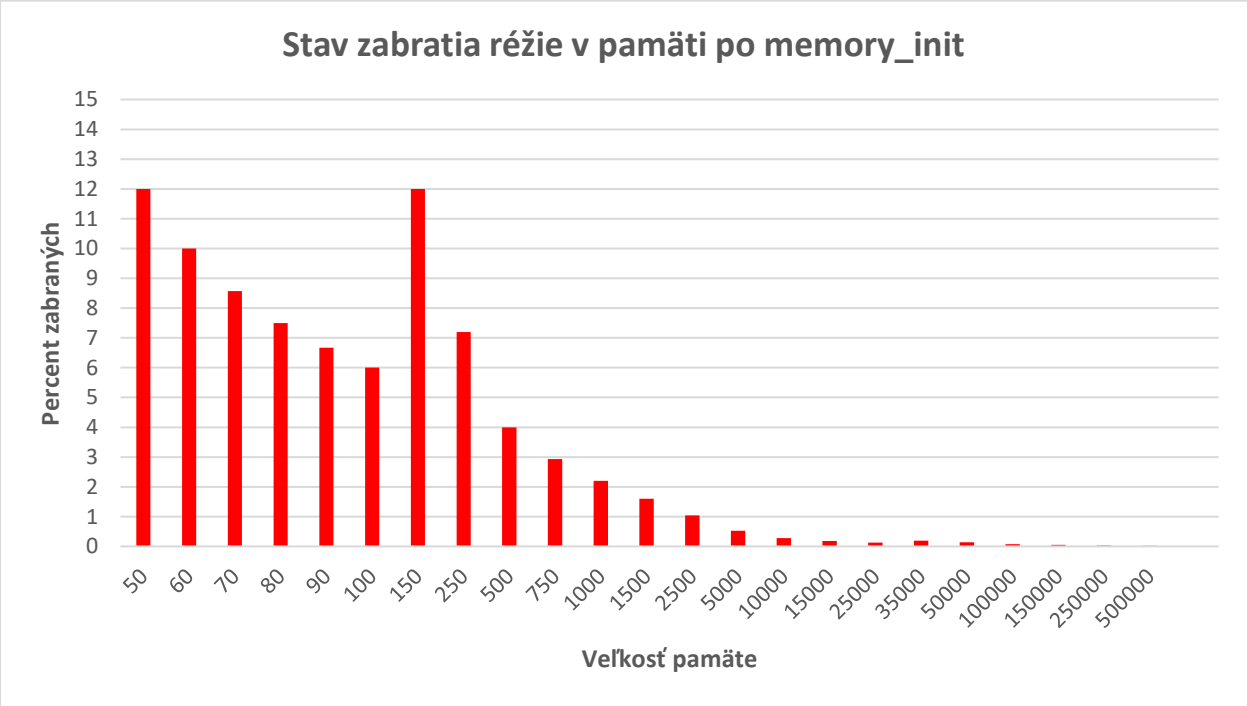
- Funkcia rozdelí pridelený blok na veľkosť, ktorú požaduje uživateľ a na blok o velkosti, ktorá ostane po rozdelení.

Časová zložitosť

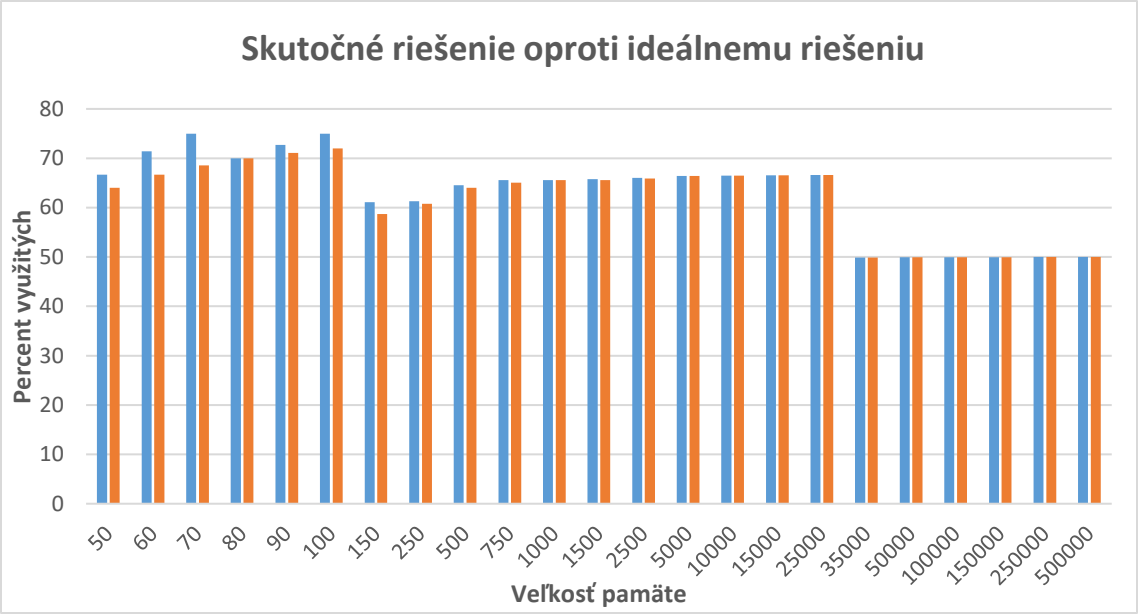
Keďže ide o prehľadávanie lineárne zreťazeného zoznamu, ide o časovú zložitosť O(n), kde n je počet blokov. V najhoršom prípade musím prejsť celý zoznam. V najlepšom sa alokuje hneď prvý blok. Moja implementácia ale bude rýchlejšia, ako iba jeden explicitný zoznam. Naopak bude pomalšia ako metóda štyri – binárny strom. Pre malé pamäte do 63 B som použil iba jeden zoznam z dôvodu úspory už aj tak malej pamäte.

Pamäťová zložitosť

Najprv ukážem ako veľa zaberá moja réžia pamäti po *memory_init.* Veľkú časť bude zaberať pre malé pamäte, ale čím bude pamäť väčšia, tým menej percent bude zaberať. Túto hodnotu som zlepšil aj tým, že som použil dynamické hlavičky. To znamená, že na všetky smerníky, hlavičky a päty som použil jeden typ premennej. A to char pri malých pamätiach, neskôr short pri stredných pamätiach a nakoniec int alebo long long pri veľkých veľkostiach. Nižšie je uvedený graf, kde je graficky ukázaný, počet percent pamäte minutých na réžiu.



Ďalej ukazujem koľko percent blokov(modrý stĺpec) a bajtov(oranžový stĺpec) pamäte sa mi podarilo alokovať oproti ideálnemu riešeniu. Graf nižšie reprezentuje percentuálne zastúpenie týchto dvoch vlastností pre rôzne veľkosti pamäti.



Testovacie scenáre

Vytvoril som 6 testovacích funkcií a zameral som sa na:

- prideľovanie rovnakých blokov malej veľkosti (veľkosti 8 bytov) pri použití malej pamäte 50 bytov
- prideľovanie nerovnakých blokov malej veľkosti (náhodné veľkosti 8 až 24 bytov) pri použití malej pamäte 100 bytov
- prideľovanie nerovnakých blokov väčšej veľkosti (veľkosti 500 až 5000 bytov) pri použití strednej pamäte 20000 bytov
- prideľovanie nerovnakých blokov malých a veľkých veľkostí (veľkosti od 8 bytov do 50 000) pri použití veľkej pamäte 30000 bytov
- náhodné prideľovanie a uvoľňovanie blokov náhodných veľkostí 8 až 50000 bytov pri použití malej pamäte 300000 až 1000000 bytov a nakoniec sledovanie ich uvoľnenia