Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Dátové štruktúry a algoritmy

Zadanie 1 - Správca pamäti

Jakub Hrnčár
AIS ID: 110 800, ak. rok 2020/2021

1. Úvod do zadania

Mojou úlohou bolo naprogramovať vlastnú implementáciu funkcií malloc() a free(), ktoré sa v jazyku C používajú na správu dynamicky alokovanej pamäti. Nemali sme pracovať priamo s haldou voľnej pamäte v operačnom systéme, ale iba s nami vytvoreným polom, ktoré haldu reprezentuje.

Hlavné funkcie mali byť:

- `void memory_init(void *ptr, unsigned int size);` - funkcia pripraví pole pamäte do nami požadovaného formátu a nastaví globálny ukazovateľ na začiatok poľa. Argumenty sú pole a jeho veľkosť.
- `void *memory_alloc(unsigned int size);` - analogická funkcia ku funkcií malloc() v jazyku C. Vracia ukazovateľ na užívateľom vyhradenú pamäť. Argumentom je len veľkosť vyhradenej pamäte.
- `int memory_free(void *valid_ptr);` - analogická funkcia ku funkcií free() v jazyku C. Vráti 0 ak sa podarilo uvoľniť pamäť a 1 ak sa nepodarilo. Pri implementácií som vychádzal z predpokladu, že ukazovateľ, ktorý je jediný argument, bude vždy korektný - bol kedysi vrátený funkciou memory_alloc a ešte nebol uvoľnený.
- `int memory_check(void *ptr);` - kontrolná funkcia, jej argument je ukazovateľ. Funkcia musí vrátiť 1 len v prípade, že ukazovateľ v argumente ukazuje na začiatok tela alokovaného bloku. V každom inom prípade vráti 0.

Mal som na výber z viac spôsobov implementácie, ja som si vybral metódu explicitného spájaného zoznamu voľných blokov kombinovanú s metódou najlepší vhodný blok (angl. explicit list, best fit).

Súčasťou zadania je aj 6 testovacích scenárov, ktoré sa zameriavajú na rôzne hraničné a modelové situácie, ktoré môžu pri používaní nastať a všetky dokazujú správnosť implementácie.

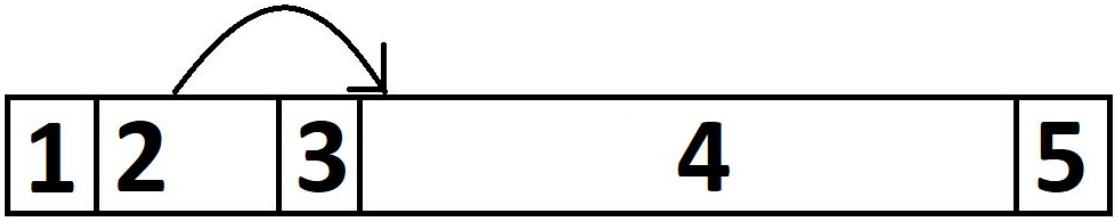
Pri implementovaní som sa často musel rozhodovať medzi lepšou pamäťovou zložitosťou (a teda menšou fragmentáciou), alebo lepšou časovou a algoritmickou zložitosťou. Vo väčšine prípadov som sa rozhodol pre lepšiu fragmentáciu a neskôr som sa snažil program optimalizovať na lepšiu časovú zložitosť ako najviac som vedel.

2. Stručný opis algoritmu

1. **Funkcia memory_init:**

Táto funkcia nie je veľmi komplexná, rieši len jednoduché základné formátovanie pamäte. Funkcia hneď po spustení nastaví globálny ukazovateľ na začiatok pamäti a nastaví pamäť na nuly, čo je veľmi dôležité pre správny chod programu. Vykoná to vďaka funkcií memset(), do ktorej vstupujú oba argumenty funkcie memory_init a to ukazovateľ na pamäť a jej veľkosť.

Dizajn pamäte po zavolaní memory_init je takýto:



1 - Prvé 4 bajty obsadí veľkosť celej pamäte
2 - 8 bajtov pre ukazovateľ na prvé voľné miesto dostupné užívateľovi
3 - 4 bajty pre hlavičku, kde je zapísaná veľkosť tela bloku. Obsahuje príznak zaplnenosti pamäte, ktorým je znamienko pred hodnotou veľkosti. V tomto prípade je kladné, pretože blok je nepriradený.
4 - telo bloku, užívateľovi dostupná voľná pamäť, obsahuje nuly, pretože ešte nebol vytvorený spájaný zoznam voľných blokov
5 - 4 bajty pre pätičku bloku, ktorá je identická s hlavičkou (3)
Body 3, 4 a 5 predstavujú prvý voľný blok. Réžia celej pamäte zaberá 12 bajtov, réžia bloku zaberá ďalších 8 bajtov. Užívateľ má teda k dispozící o 20 bajtov menej ako reálne požiadal.

2. **Funkcia memory_alloc:**

Nosná funkcia celého zadania, vykonáva analogickú funkciu ku funkcií malloc(). Má jediný argument, ktorým je veľkosť pamäte, ktorú užívateľ vyžaduje. Funkcia po úspešnom pridelení pamäte vráti ukazovateľ na začiatok tela priradeného bloku, v opačnom prípade vráti NULL.

Funkcia musí byť robustná, pretože môže nastať veľmi veľa hraničných situácií, ktoré musí vedieť konzistentne vyriešiť.

Príznak zaplnenosti pamäte som vložil do hlavičky/pätičky bloku vo forme znamienka pred hodnotou veľkosti. Ak je znamienko záporné (a teda celá hodnota veľkosti je menšia ako 0), blok je priradený. Ak je kladné (a teda celá hodnota je väčšia ako 0), blok je voľný a je zaradený v spájanom zozname voľných blokov.

Telo bloku je zaplnené jednotkami pomocou funkcie memset().

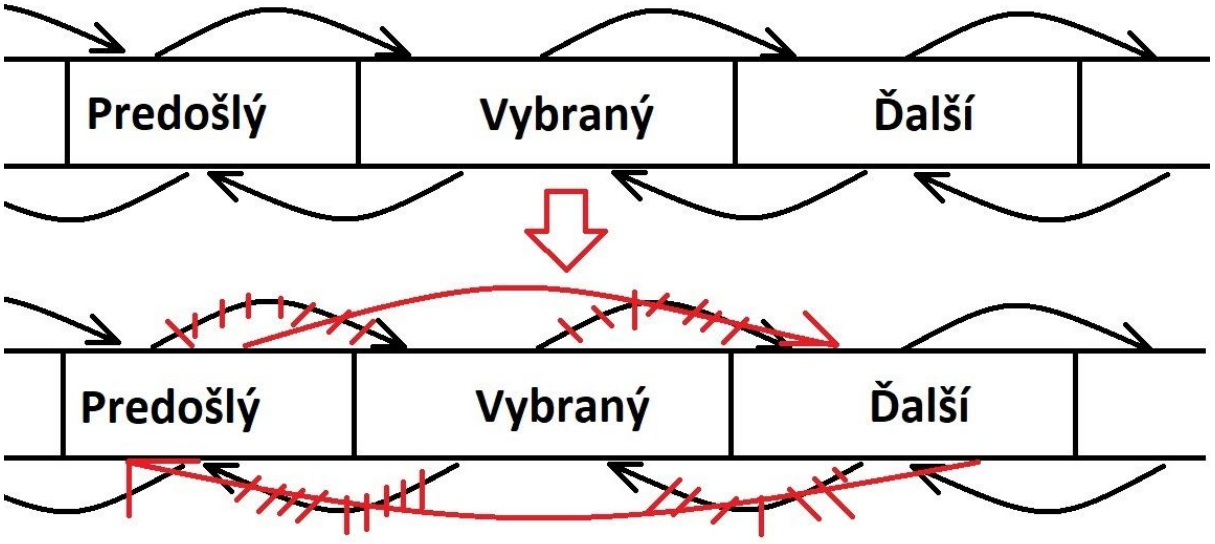
Dizajn priradeného bloku:

1 - 4 bajty pre hlavičku, kde je uložená veľkosť tela bloku s príznakom zaplnenia (teda veľkosť je prenásobená číslom -1).
2 - telo bloku, počet bajtov je taký, aký si užívateľ vyžiada. Musí mať aspoň 16 bajtov.
3 - 4 bajty pre pätičku, ktorá je identická s hlavičkou.

Kvôli podmienkam réžie je minimálna užívateľom zadaná veľkosť vždy aspoň 16 bajtov, pretože do tela bloku sa musia zmestiť dva ukazovatele veľké 8 bajtov, teda dokopy 16 bajtov. Ak užívateľ vyžiada menšiu pamäť ako 16 bajtov, program ju zaokrúhli na 16 bajtov.

Na nájdenie voľného bloku pamäte sa používa algoritmus najlepší vhodný (ang. best fit), ktorý nájde najmenší dostatočne veľký voľný blok a určí ho na priradenie. Ak nájde bok ktorý je ideálne veľký, zastaví sa a vyberie ho a ak nájde viac rovnako veľkých, tak vyberie ktorýkoľvek.

Poslednou časťou základnej funkcionality je vynechanie vybraného bloku zo spájaného zoznamu a to zmenením ukazovateľov predošlého a ďalšieho voľného bloku. Predošlí bude ukazovať na ďalší a ďalší bude ukazovať na predošlí.



Základná funkcionalita je jasná, avšak môže tu nastať niekoľko situácií, ktoré si vyžadujú pozornosť a opatrenie. Z definície best fit algoritmu vyplýva, že môže nájsť blok, ktorý väčší ako potrebujeme. Na základe veľkosti pamäte, ktorá je navyše, môžu nastať 2 situácie:

- Navyše miesto je väčšie alebo rovné ako 24 bajtov - V tejto situácií sa navyše miesto odčlení a vytvorí sa nový plnohodnotný prázdny blok, ktorý sa zároveň aj zapojí do spájaného zoznamu voľných blokov. Ukazovatele na predošlí a ďalší blok si zobral z pôvodného bloku a zapíše ich do nového bloku.
- Navyše miesto je menšie ako 24 bajtov - Toto je komplikovaná situácia, ktorá vedie ku internej fragmentácií, preto som sa snažil túto fragmentáciu odstrániť najlepšie ako to išlo. Program toto navyše miesto označí a prepíše na písmená ‘V’. V ďalších krokoch teda musí program byť dostatočne inteligentný na to, aby rozoznal túto označenú pamäť a dokázal s ňou pracovať.

Poslednou funkciou je aktualizovanie ukazovateľa na začiatku celej pamäte, ktorý ukazuje na prvé voľné miesto v pamäti, na nové prvé voľné miesto v pamäti.

3. **Funkcia memory_check:**

Overovacia funkcia, jej jediná funkcionalita je skontrolovať ukazovateľ z argumentu, či ukazuje na začiatok tela pridelenej pamäte, t. j. či bol niekedy v minulosti vrátený funkciou memory_alloc. V prípade, že takto vrátený bol, vráti 1, inak 0. Veľmi dôležitá podmienka pri tejto funkcií je, že 1 musí vrátiť len v prípade, že ukazovateľ ukazuje na začiatok tela pridelenej pamäte a nikde inde, teda nemôže ukazovať do stredu tela, ani na začiatok voľnej pamäte a ani na žiadnu hlavičku alebo pätičku.

Na overenie úplnej korektnosti tejto funkcie som navrhol jednoduchý test, ktorý sa nachádza v sekcií testy.

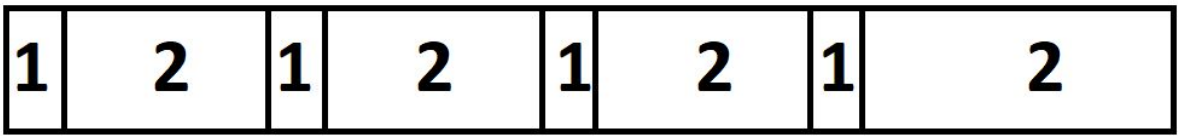
4. **Funkcia memory_free:**

Funkcia memory_free poskytuje užívateľovi uvoľňovanie už pridelenej pamäte. Táto funkcia má niekoľko komponentov, ktoré sa starajú o jej korektné implementovanie v každej situácií, ktorá môže nastať. Samotný koncept tejto funkcie je veľmi jednoduchý, avšak veľmi rýchlo sa dokáže značne skomplikovať.

Prvým komponentnom je jednoduché uvoľnenie bloku. Táto časť je úplne opačná k priradeniu pamäte vo funkcií memory_alloc(). Telo bloku je treba vynulovať a hlavičku s pätičkou prenásobiť číslom -1, aby sa stratil príznak zaplnenosti.

Druhý komponent je uvoľnenie pamäte, ktorá má označenie V. Tento komponent sa pozrie za a pred vybraný blok (samozrejme iba ak sa nenachádzajú na konci alebo začiatku pamäte) a zistí, či sa tam náhodou nenachádzajú voľné písmená ‘V’. Ak sa nachádzajú, všetky tieto písmená program zakomponuje do vybraného uvoľneného bloku a zväčší ho o počet týchto písmen. Je veľmi podstatné tieto písmená uvoľniť čo najskôr, aby potom nevytvárali problémy v iných častiach programu.

Ďalší, veľmi dôležitý a obsiahly komponent je spájanie voľných blokov. Toto spojenie je vitálne pre správny chod programu, pretože eliminuje tzv. falošnú fragmentáciu, ktorej príklad je na nasledujúcom obrázku:

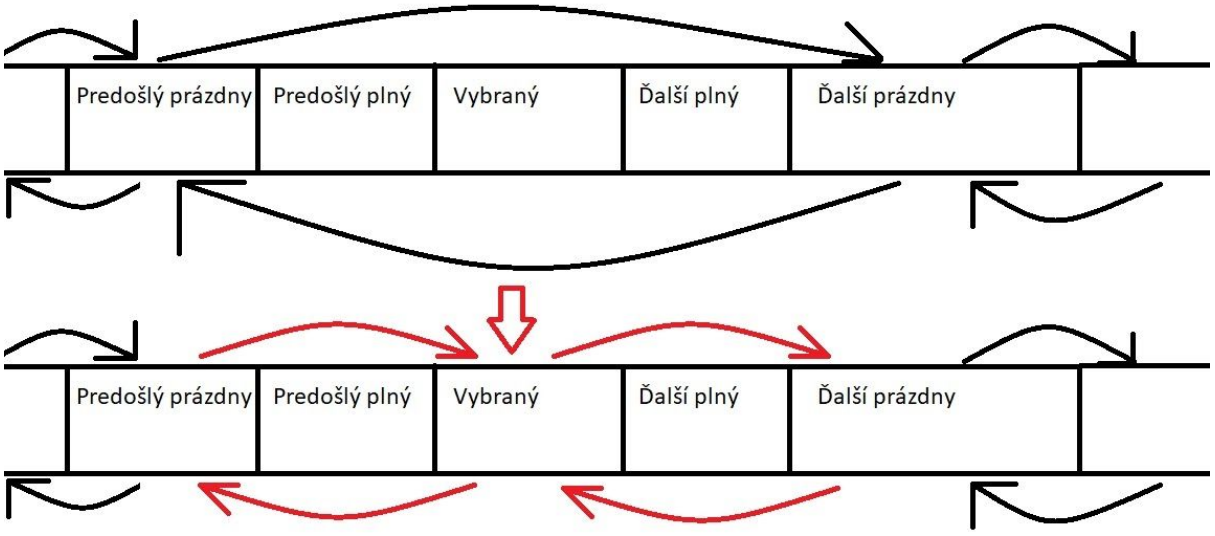


1 - hlavičky prázdnych blokov (pre jednoduchosť je pätička vynechaná)
2 - telo prázdnych blokov

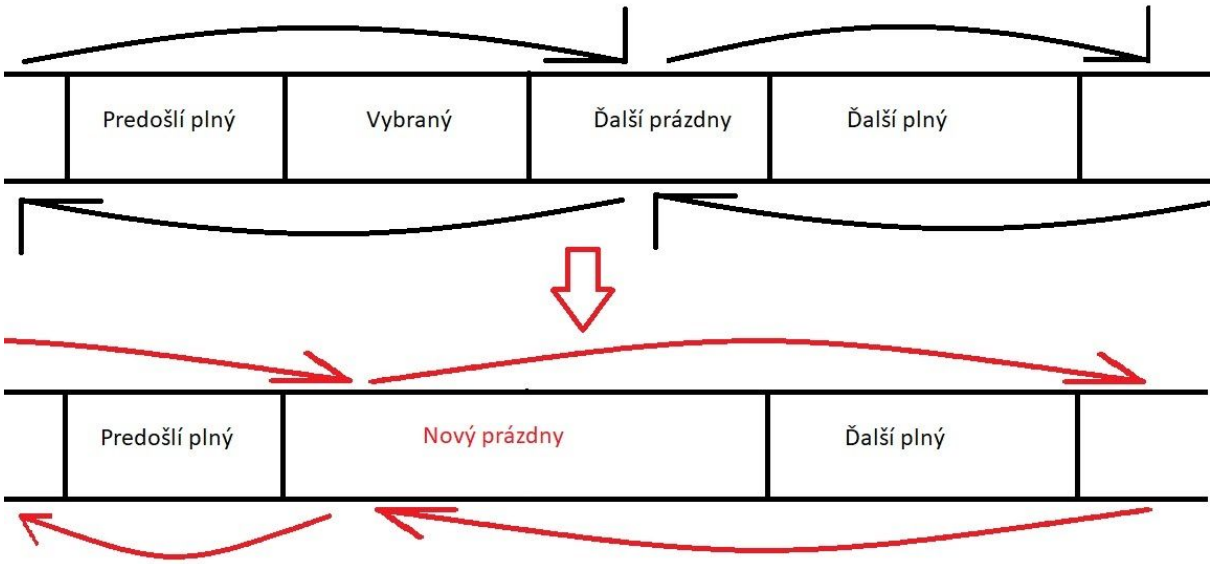
Ako vidíme na obrázku, táto celá pamäť je prázdna, avšak je rozdelená na veľa prázdnych blokov, ktoré nie sú spolu spojené a teda nedá sa nedá do tejto pamäte vložiť nejaký väčší blok - toto sa nazýva falošná fragmentácia. Ak by sa však tieto bloky pospájali do jedného veľkého bloku, bude možné vložiť aj nejaký väčší blok.

Existujú 4 prípady uvoľnenia, ktoré môžu nastať a ktoré je potrebné implementovať do programu. To o ktorý prípad ide viem zistiť veľmi jednoducho: program sa pozrie na predošlí a ďalší blok. Ak tam nájde koniec/začiatok pamäte, priradí do premennej 0. Ak nájde blok, priradí jeho veľkosť aj so znamienkom. Na základe porovnávania tejto premennej s nulou program vie veľmi jednoducho zistiť daný prípad. Prípady sú nasledovné:

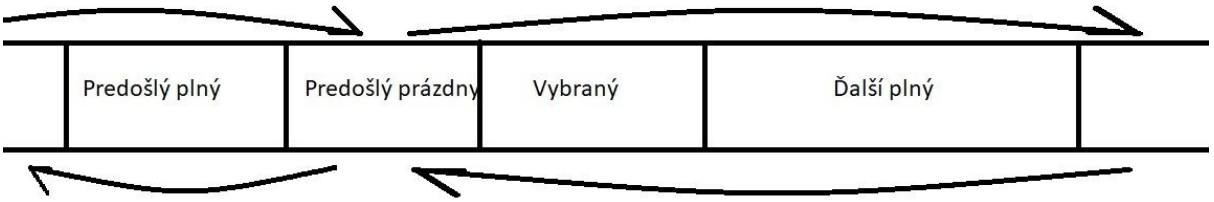
1. **plný - vybraný - plný**
   Táto situácia sa na prvý pohľad javí ako štandardná a najľahšia situácia, avšak opak je pravdou. Treba si totiž uvedomiť, že pod plným blokom rozumieme aj koniec alebo začiatok pamäte. Je veľmi dôležité, aby toto bolo ošetrené, pretože program nesmie pristupovať mimo vyhradenú pamäť. Implementácia je vykonaná cez aritmetiku ukazovateľov a a porovnávanie koncovej a začiatočnej adresy. Ak sa ale nenachádza na začiatku alebo na konci pamäte, nájdu sa najbližšie voľné bloky cez cyklus prehľadávania pamäte. Pri nájdení najbližších blokov sa nastavia všetky ukazovatele na správne miesta podľa obrázka:

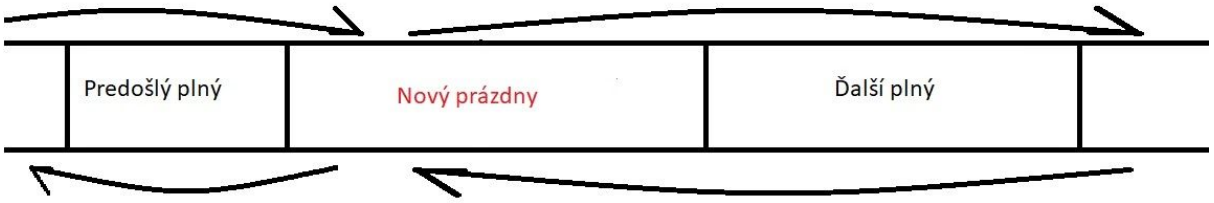


2. **plný - vybraný - prázdny**
   V tejto situácií je nutné skopírovať ukazovatele z nasledujúceho voľného bloku, zlúčiť oba bloky a vložiť ukazovatele na začiatok tela nového bloku. Potom treba prispôsobiť ukazovatele v ďalších voľných blokoch aby správne ukazovali na nový veľký blok.

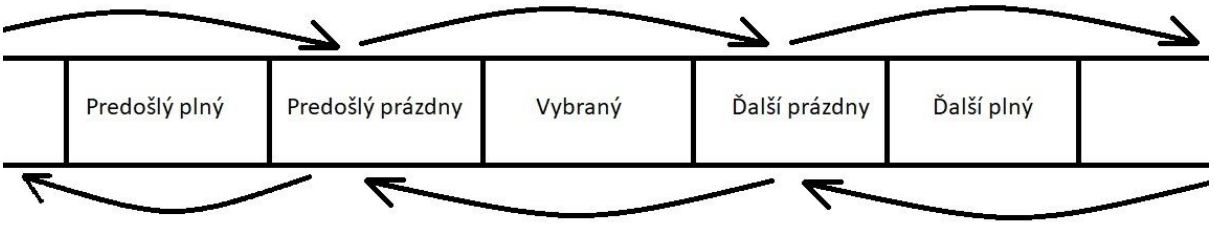


3. **prázdny - vybraný - plný**
   Relatívne najjednoduchšia situácia, pretože nie je treba nastavovať ukazovatele v iných blokoch okrem predošlého voľného.





4. **prázdny - vybraný - prázdny**
   V tomto kroku je potrebné veľké množstvo aritmetiky ukazovateľov, pretože sa musia zlúčiť dokopy až 3 bloky a je potrebné správne nastaviť všetky ukazovatele, ktoré s nimi súvisia.



Ako posledné je nutné opäť aktualizovať ukazovateľ na začiatku pamäte, aby ukazoval na nové prvé voľné miesto v pamäti.

3. Testovanie programu

1. Test 1 - Uvoľňovanie a alokovanie blokov striedavo

```
char memory[200];
memory_init(memory,200);
void *p0 = memory_alloc(15);
void *p1 = memory_alloc(15);
void *p2 = memory_alloc(15);
void *p3 = memory_alloc(15);
void *p4 = memory_alloc(15);

memory_free(p0);
memory_free(p1);
memory_free(p2);

void* p5 = memory_alloc(15);
void* p6 = memory_alloc(15);
memory_free(p5);
p5 = memory_alloc(15);
void* p7 = memory_alloc(15);
void* p8 = memory_alloc(15);
memory_free(p8);
memory_free(p6);
```

1. Inicializovanie a zaplnenie pamäte - zaokrúhlenie veľkostí na 16 bajtov.
2. **Hraničná situácia** - uvoľnenie prvého bloku.
3. Uvoľnenie ďalších blokov a spojenie podľa prípadu 3 a 2.
4. **Hraničná situácia** - p5 vloží do pamäte písmená V, následne sa riadne uvoľnia.

2. Test 2 - Best fit test

```
char memory[500];
memory_init(memory, 500);

void* p0 = memory_alloc(20);
void* p1 = memory_alloc(30);
void* p2 = memory_alloc(40);
void* p3 = memory_alloc(50);
void* p4 = memory_alloc(35);
void* p5 = memory_alloc(45);
void* p6 = memory_alloc(20);
void* p7 = memory_alloc(25);
```

```
void* p8 = memory_alloc(33);

memory_free(p0);
memory_free(p3);
memory_free(p5);
memory_free(p7);

void* p9 = memory_alloc(40);
void* p10 = memory_alloc(20);

memory_free(p6);
```

1. Inicializovanie a zaplnenie pamäte.
2. Uvoľnenie niektorých blokov.
3. **Hraničná situácia** - p9 a p10 sú ukazovatele, ktorým musí best fit vybrať miesto. Pre p9 vyberie miesto ukazovateľa p5 a pre p10 vyberie miesto ukazovateľa p0.
4. **Hraničná situácia** - uvoľnenie ukazovateľa p6, pred ktorým je písmeno ‘V’ následné spojenie s ďalším blokom podľa prípadu 3.

3. Test 3 - Práca s väčšími číslami

```
char memory[5000];

memory_init(memory,5000);
void *p0 = memory_alloc(1500);
void *p1 = memory_alloc(2700);
void *p2 = memory_alloc(1200);

memory_free(p1);
void *p3 = memory_alloc(2000);
void *p4 = memory_alloc(800);
memory_free(p0);
```

1. Inicializovanie a zaplnenie pamäte.
2. **Hraničná situácia** - Ukazovateľ p2 nie je možné vytvoriť a priradiť mu pamäť, pretože nie je dostatok miesta v pamäti.

4. Test 4 - memory_check test

```
char memory[170];
memory_init(memory,170);
void *p0 = memory_alloc(15);
void *p1 = memory_alloc(15);
void *p2 = memory_alloc(15);
void *p3 = memory_alloc(15);
```

```
for (int i = 0; i<170; i++){
    int k=memory_check(((char *)heap + i));
    if (k){
       printf("%d\n",k);
        }
    }
```

Tento test otestuje na každom bajte pamäte funkciu memory_check. Ak vytvoríme 4 plné bloky tak memory check musí vrátiť práve 4 - krát číslo 1.

5. **Test 5 - Základný test s veľkými číslami**

```
char memory[60000];
memory_init(memory,60000);
void *p0 = memory_alloc(50000);
void *p1 = memory_alloc(421);
void *p2 = memory_alloc(6900);
memory_free(p1);
void *p3 = memory_alloc(11);

memory_free(p2);
void *p4 = memory_alloc(8000);
memory_free(p0);
```

1. Inicializovanie a zaplnenie pamäte.
2. **Hraničná situácia** - Ukazovateľ p0 má takmer maximálnu možnú veľkosť.
3. Alokovanie a uvoľňovanie ďalších blokov.

6. **Test 6 - Len jeden blok v pamäti**

```
char memory[100];
memory_init(memory,100);
void *p0 = memory_alloc(80);
memory_free(p0);
```

1. Inicializovanie pamäte.
2. **Hraničná situácia** - Alokovanie bloku, ktorý bude mať presnú veľkosť voľného miesta. Pri jeho následnom uvoľnení sa nesmie pristúpiť mimo pamäť.

4. Záver

Program je funkčný a predstavuje obstojnú simuláciu reálnej funkcie malloc() a free(), avšak je určite menej efektívny ako reálne implementácie týchto funkcií.

Interná fragmentácia programu predstavuje veľkosť využitej réžie v bloku - každý blok musí mať hlavičku, miesto na 2 ukazovatele a pätičku, čo dokopy predstavuje minimálne 24 bajtov na 1 blok. Užívateľ si v tomto programe nedokáže vyžiadať menší blok ako 16 bajtov, čo je veľká limitácia. Čím bude väčšia celková pamäť, tým bude aj vyššia percentuálna úspešnosť efektívneho zaplnenia pamäte. Na druhej strane, vďaka best fit algoritmu a označovaniu moc malej voľnej pamäte písmenom ‘V’ sa mi podarilo minimalizovať externú fragmentáciu a vlastne všetko voľné miesto sa dá na niečo využiť.

Časová zložitosť programu je lineárna - O(n), každá funkcia používa aspoň v 1 časti lineárne zložitý algoritmus. V istých častiach programu sa mi podarilo dosiahnuť konštantná zložitosť, avšak častým používaním funkcie memset() sa tento efekt stráca, pretože memset() má lineárnu zložitosť.