Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológii
7. marec 2021

Dátové štruktúry a algoritmy

Dokumentácia k zadaniu č. 1

Roland Vdovják
xvdovjak@stuba.sk
ID: 110912

Roland Vdovják
ID: 110912

1 Zadanie

Úlohou prvého zadania bolo implementovať štyri funkcie v programovacom jazyku C, riešenie dôkladne otestovať a zdokumentovať.

Funkcie:

- void *memory_alloc(unsigned int size);
- int memory_free(void *valid_ptr);
- int memory_check(void *ptr);
- void memory_init(void *ptr, unsigned int size);

Memory_alloc má poskytovať služby analogické štandardnému malloc. Teda, vstupné parametre sú veľkosť požadovaného súvislého bloku pamäte a funkcia mu vráti: ukazovateľ na úspešne alokovaný kus voľnej pamäte, ktorý sa vyhradil, alebo NULL, keď nie je možné súvislú pamäť požadovanej veľkosť vyhradiť.

Memory_free slúži na uvoľnenie vyhradeného bloku pamäti, podobne ako funkcia free. Funkcia vráti 0, ak sa podarilo (funkcia zbehla úspešne) uvoľniť blok pamäti, inak vráti 1. Môžete predpokladať, že parameter bude vždy platný ukazovateľ, vrátený z predchádzajúcich volaní funkcie **memory_alloc**, ktorý ešte nebol uvoľnený.

Memory_check slúži na skontrolovanie, či parameter (ukazovateľ) je platný ukazovateľ, ktorý bol v nejakom z predchádzajúcich volaní vrátený funkciou **memory_alloc** a zatiaľ nebol uvoľnený funkciou **memory_free**. Funkcia vráti 0, ak je ukazovateľ neplatný, inak vráti 1.

Memory_init slúži na inicializáciu spravovanej voľnej pamäte. Predpokladajte, že funkcia sa volá práve raz pred všetkými inými volaniami **memory_alloc**, **memory_free** a **memory_check**. Viď testovanie nižšie. Ako vstupný parameter funkcie príde blok pamäte, ktorú môžete použiť pre organizovanie a aj pridelenie voľnej pamäte. Vaše funkcie nemôžu používať globálne premenné okrem jednej globálnej premennej na zapamätanie ukazovateľa na pamäť, ktorá vstupuje do funkcie **memory_init**. Ukazovatele, ktoré prideľuje vaša funkcia **memory_alloc** musia byť výhradne z bloku pamäte, ktorá bola pridelená funkcii **memory_init.**

Testovacie scenáre:

- Prideľovanie rovnakých blokov malej veľkosti (veľkosti 8 až 24 bytov) pri použití malých celkových blokov pre správcu pamäte (do 50 bytov, do 100 bytov, do 200 bytov),
- Prideľovanie nerovnakých blokov malej veľkosti (náhodné veľkosti 8 až 24 bytov) pri použití malých celkových blokov pre správcu pamäte (do 50 bytov, do 100 bytov, do 200 bytov),
- Prideľovanie nerovnakých blokov väčšej veľkosti (veľkosti 500 až 5000 bytov) pri použití väčších celkových blokov pre správcu pamäte (aspoň veľkosti 1000 bytov),
- Prideľovanie nerovnakých blokov malých a veľkých veľkostí (veľkosti od 8 bytov do 50 000) pri použití väčších celkových blokov pre správcu pamäte (aspoň veľkosti 1000 bytov).
- Vyhodnotiť koľko % blokov sa podarilo alokovať oproti ideálnemu riešeniu

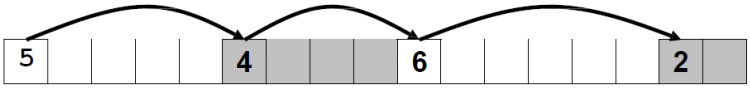
2 Teoretické východiská

Dynamická správa pamäte je neoddeliteľnou súčasťou pokročilých programov. Dynamická pamäť sa prideľuje v čase výpočtu, nie v čase prekladu, jej veľkosť nemusí byť známa až do okamihu pridelenia. Vyžiadanie jej pridelenia od procedúry *malloc* (a pod.) zahŕňa parameter veľkosti, *size*. (Kohútka, 2021, s. 11)

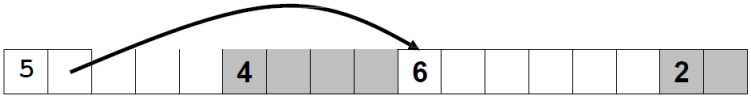
2.1 Metódy na udržiavanie voľnej pamäte

Poznáme štyri hlavné metódy na udržiavanie voľnej pamäte, ktoré nám hovoria, ako je zoznam alebo zoznamy blokov v pamäti usporiadané.

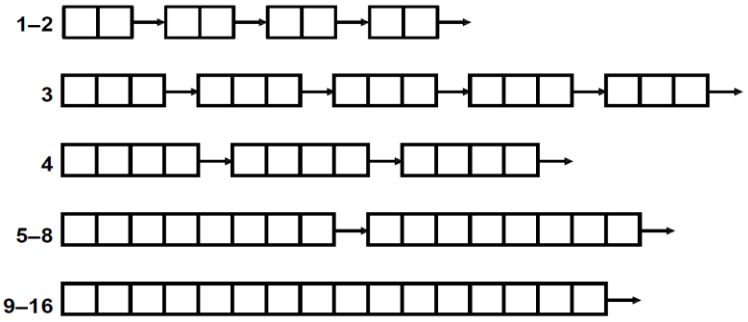
1. **Implicitný** zoznam voľnej pamäti s použitím dĺžok, ktorý spája všetky bloky



2. **Explicitný** zoznam voľnej pamäti s použitím ukazovateľov zapísaných vo voľných blokoch



3. Oddelené zoznamy blokov voľnej pamäti



4. Bloky usporiadané podľa veľkosti

(Kohútka, 2021, s. 22)

2.2 Spôsoby na nájdenie voľného bloku

Spôsoby hľadania voľného bloku nám hovoria o tom, ako sa nájde blok pamäte ktorý spĺňa nami požadovanú veľkosť a je ho možné alokovať.

- **Prvý vhodný** (first fit)
  Zoznam sa prehľadá od začiatku, vyberie sa prvý voľný blok.
- **Nasledujúci vhodný** (next fit)
  Zoznam sa prehľadáva od miesta, kde skončilo predchádzajúce hľadanie.
- **Najlepšie vhodný** (best fit)
  Vyberie blok s veľkosťou najbližšou k požadovanej veľkosti, prechádza celý zoznam.
- **Najhoršie vhodný** (worst fit)
  Vyberie blok s najväčšou veľkosťou k požadovanej veľkosti, prechádza celý zoznam.

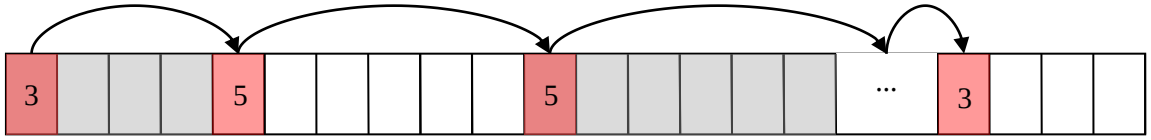
(Kohútka, 2021, s. 24)

3 Riešenie

V našom riešení sme sa rozhodli pre implicitný zoznam s metódou hľadania voľného bloku „najlepší vhodný“ (best fit).

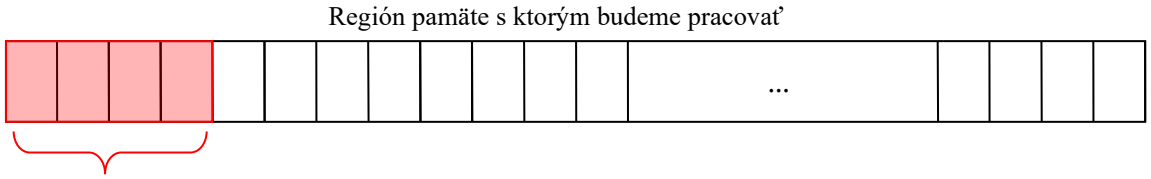
3.1 Implicitný zoznam

Implicitný zoznam spočíva v spojení každého bloku pamäti, voľného alebo obsadeného, s nasledujúcim blokom. Nasledujúci diagram obsahuje grafické znázornenie takéhoto zoznamu. Červeným podfarbením sú označené hlavičky blokov, sivo podfarbené sú označené obsadené slová a bez podfarbenia sú označené voľné slová.



Naša implementácia je založená na tom, že blok obsahuje hlavičku v ktorej je informácia o veľkosti bloku (vo forme *unsigned int*) a bajt rozhodujúci o obsadenosti bloku. Pre prácu s regiónom pamäti potrebujeme globálny smerník (*void* memstart*), ktorý ukazuje na začiatok regiónu.

Na základe týchto informácii program vie z každého bloku posunúť smerník na nasledujúci blok. Aby sme zabránil presunutiu smerníka na pamäť mimo región pamäti s ktorou pracujeme, globálny smerník *memstart* po pretypovaní na *(int*)memstart* obsahuje počiatočnú voľnú veľkosť pamäti, t.j. pamäť, ktorú chceme inicializovať mínus veľkosť hlavičky regiónu.



3.2 Inicializácia pamäte

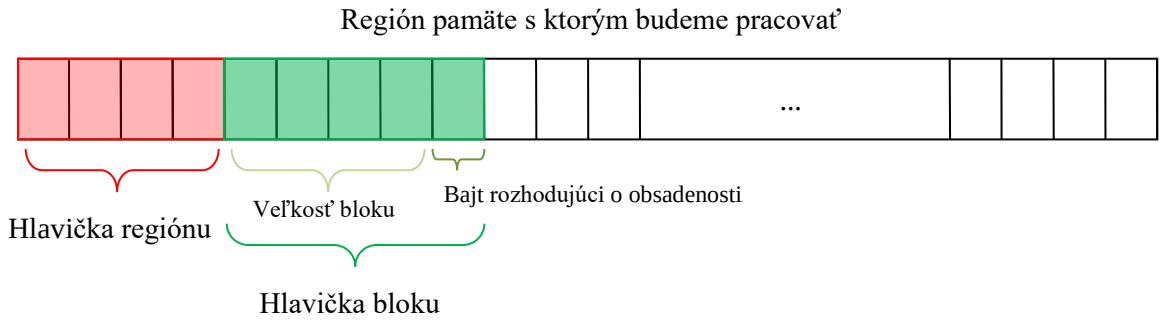
Zo zadanie vieme, že funkcia *memory_init* má obsahovať dva parametre, smerník na región pamäte, s ktorou budeme pracovať a veľkosť, ktorú chceme alokovať.

Súčasťou našej inicializácie je aj podmienka, za ktorej je možné alokovať. Podmienka hovorí o tom, aby vstupný parameter veľkosti bol väčší ako súčet bajtov potrebných na hlavičku regiónu, hlavičku prvého bloku a minimálnu možnú alokovateľnú pamäť, t.j. zo zadania osem bajtov.
Ukážka kódu:

```
if (size < 2*sizeof(int) + 1 + 8){ ... }
```

Ďalej v inicializácií program priradí smerník z parametra funkcie globálnemu smerníku. Vypočíta počiatočnú voľnú pamäť, ktorú priradí pamäti globálneho smerníka. Program prvýkrát fragmentuje zostávajúcu pamäť. Hlavičke bloku priradí veľkosť pamäti, ktorú bude možno alokovať. Rozhodovaciemu bajtu v pamäti hodnotu ‘1’, ktorá znamená, že blok je voľný.

Po funkcii *memory_init* vyzerá pamäť nasledovne:



3.3 Alokácia pamäte

Zo zadania vieme, že funkcia *memory_alloc* má obsahovať jeden parameter a to veľkosť pamäte, ktorú chceme alokovať. Funkcia má vrátiť buď smerník na alokovanú pamäť alebo *NULL*.

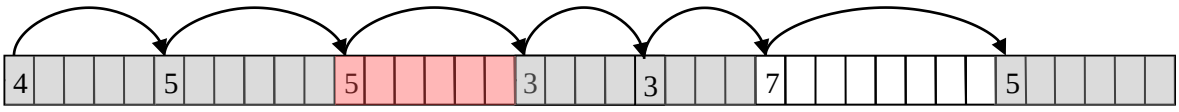
Po počiatočnej kontrole vstupu program zavolá pomocnú funkciu *bestFit*, ktorá má ako parameter číslo, ktoré reprezentuje veľkosť pamäte, ktorú chceme alokovať.

3.3.1 Najlepší vhodný (best fit)

Spôsob na nájdenie voľného bloku metódou „najlepší vhodný“ funguje na princípe prechádzania celého zoznamu blokov. Počas tohto prechádzania hľadá voľný blok o veľkosti najbližšej ku hľadanej alebo presne hľadanú veľkosť. Nižšie uvedené príklady zjednodušene ukazujú, ako by mala fungovať naša funkcia *bestFit*.

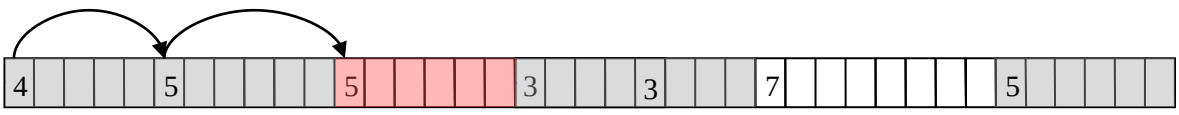
Príklad 1: chceme alokovať pamäť o veľkosti štyri bajty

Program prejde celý zoznam blokov, v pomocnej premennej si uloží smerník na najlepší blok. Po skončení vráti smerník na blok, ktorý je najbližšie požadovanej veľkosti. V tomto príklade ide o tretí blok (podfarbený na červeno).



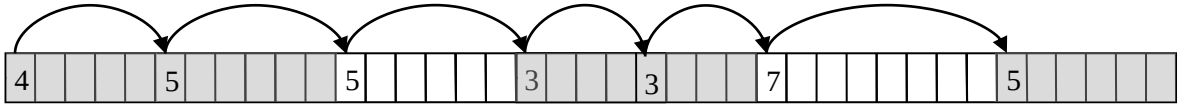
Príklad 2: chceme alokovať pamäť o veľkosti päť bajtov

Program počas prechádzania zoznamom narazí na blok o rovnakej veľkosti, ako chceme alokovať. Prestane prechádzať a vráti smerník na daný blok. V tomto príklade ide tiež o tretí blok (podfarbený na červeno).



Príklad 3: chceme pamäť o veľkosti desať bajtov

Program prejde celý zoznam. Po skončení sa v pomocnej premennej nenachádza nič, pretože tak veľký blok nebolo možné nájsť. Program vráti *NULL*.



Funkcia obsahuje cyklus, ktorý sa opakuje dokým neprejde celý región pamäte alebo nenarazí na blok o rovnakej veľkosti, ako je hľadaná veľkosť. Ak blok, na ktorom je smerník pri iterácii cyklu, je voľný, porovnáva veľkosť bloku so zatiaľ najlepšou (*best*). Ak narazí na voľný blok o rovnakej veľkosti, cyklus sa ukončí.

Ukážka kódu:

```
while ((ptr - memstart) * sizeof(int) < *(int*)memstart) {
	volny = (char*)((int*)ptr + 1);   // Bajt rozhodujuci o obsadenosti
	if (*volny == 1 && i > size) {
		if (best == NULL) {
			best = ptr;
		}
		else if (*ptr <= *best) {
			best = ptr;
		}
		if (*best == size) break;
	}
	if ((ptr - memstart) * sizeof(int) < *(int*)memstart) {
		ptr = ((char*)ptr + i + sizeof(int) + 1);
		i = *ptr;
	}
}
```

3.3.2 Zmena bloku na alokovaný

Ak funkcia *bestFit* vrátila smerník rôzny od *NULL*, alokovanie pokračuje, inak sa funkcia *memory_alloc* skončí a vráti *NULL*.

Program pokračuje tak, že bajt rozhodujúci o obsadenosti zmení na hodnotu ‘2’, čo znamená obsadený. Ak je možné rozdeliť blok, ktorý chceme alokovať, fragmentuje sa. Na záver sa posunie smerník z hlavičky na voľnú časť bloku pamäte, ktorý sme chceli alokovať.

Ukážka kódu:

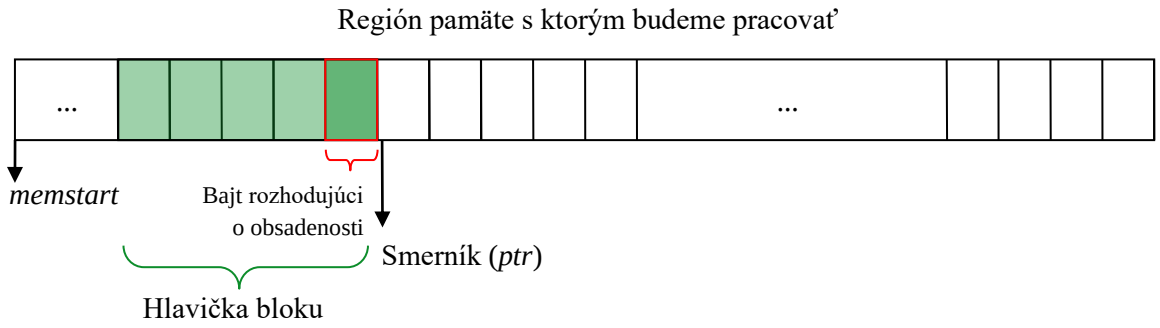
```
if (ptr == NULL) return NULL;

	long int poc = *(int*)ptr;		// Hodnota smernika
	*(char*)(ptr + sizeof(int)) = 2;  // Bajt rozhodujuci o obsadenosti
	if (poc > size + sizeof(int) + 1 + 8){
	    *(int*)ptr = size;
	    *(int*)(ptr + sizeof(int) + size + 1) = poc - size - sizeof(int) - 1;
	    *(char*)(ptr + sizeof(int)*2 + size + 1) = 1;
	}
```

3.4 Kontrola platnosti ukazovateľa

Zo zadanie vieme, že funkcia *memory_check* má obsahovať jeden parameter a to smerník. Máme zistiť, či daný smerník bol alokovaný a vrátiť hodnotu 1 alebo *NULL*.

Vychádzame z toho, že v hlavičke sa nachádza informácia o obsadenosti. Program posunie smerník na bajt, ktorý rozhoduje o obsadenosti a podľa hodnoty v ňom rozhodne, či je blok voľný alebo obsadený. Zároveň tento smerník musí byť v rozsahu inicializovanej pamäte.



Ukážka kódu:

```
if ((ptr > memstart) && ((char*)ptr - *(int*)memstart < memstart)) {
        char* ptrtype = (char*)ptr - 1;
        if (*(char*)ptrtype == 2) return 1;
        else return 0;
}
return 0;
```

3.5 Uvoľňovanie pamäte

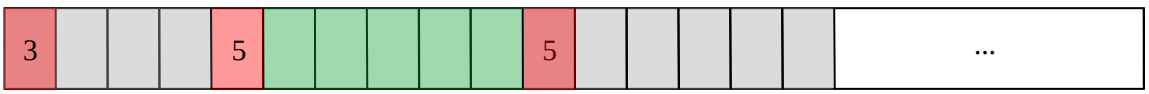
Zo zadania vieme, že funkcia *memory_free* má obsahovať jeden parameter a to smerník na pamäť bloku, ktorý chceme uvoľniť. Ak sa podarí uvoľniť, funkcia vráti hodnotu ‘1’, inak ‘0’.

3.5.1 Rôzne situácie defragmentácii

Pri uvoľňovaní pamäte musíme myslieť aj na defragmentáciu. V našom prípade existujú štyri situácie, ktoré môžu nastať. V nižšie uvedených príkladoch uvoľňujeme v poradí druhý blok (podfarbený zeleno, obsadené bloky sú podfarbené sivo, voľné bez podfarbenia).

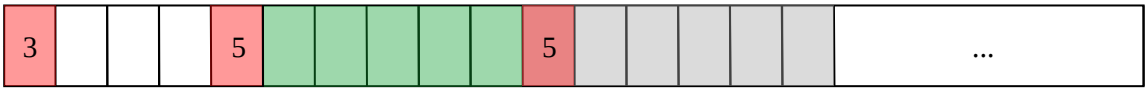
1. Predchádzajúci blok aj nasledujúci blok sú obsadené

   Nie je potrebná defragmentácia, keďže nie je čo spájať.



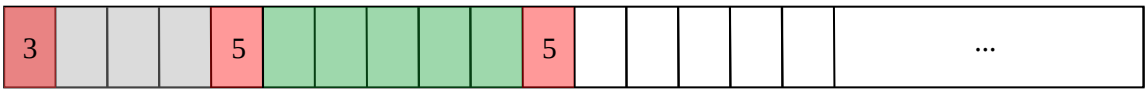
2. Predchádzajúci blok je voľný a nasledujúci blok je obsadený

   Defragmentácia je potrebná spolu s predchádzajúcim blokom. V tomto prípade je potrebné dať pozor na to, aby existoval predchádzajúci blok, t.j. aby blok, ktorý uvoľňujeme nebol prvým blokom v regióne.



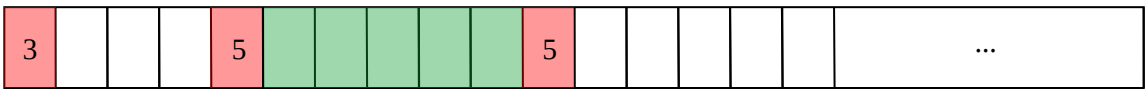
3. Predchádzajúci blok je obsadený a nasledujúci blok je voľný

Defragmentácia je potrebná spolu s nasledujúcim blokom. V tomto prípade je potrebné dať pozor na to, aby existoval nasledujúci blok, t.j. aby blok, ktorý uvoľňujeme nebol posledným blokom v regióne.



4. Predchádzajúci blok aj nasledujúci blok je voľný

Je potrebná defragmentácia z obidvoch strán, rovnako je potrebné dať pozor, aby existoval prechádzajúci aj nasledujúci blok.



Pre lepšiu prácu sme inicializovali nové smerníky.

Ukážka kódu:

```
        // Smernik ukazuje na hlavicku bloku, ktory uvolnujeme
valid_ptr = (char*)((int*)valid_ptr - 1) - 1;
        // Smernik na povodny blok
int* a = (int*)valid_ptr;
        // Smernik na nasledujuci blok
int* b = (int*)((char*)valid_ptr + *a + sizeof(int) + 1);
        // Smernik na bajt rozhodujuci o obsadenosti nasledujuceho bloku
char* c = (char*)(((int*)b) + 1);
```

3.5.3 Defragmenácia s nasledujúcim blokom

Ak je nasledujúci blok voľný a zároveň nie je mimo región pamäte, hlavička pôvodného bloku sa zväčší a nasledujúci znuluje.

Ukážka kódu:

```
if ( (*c == 1) && ((b - memstart) * sizeof(int) < *(int*)memstart) ) {
        *(int*)a = *b + *(int*)a + 2 + sizeof(int)*2;
        *(int*)b = 0;
        *(char*)(((int*)b) + 1) = 0;
}
```

3.5.3 Defragmenácia s predchádzajúcim blokom

Ak pôvodný blok nie je prvý v regióne funkcia *findPred* nájde predchádzajúci blok a vráti smerník ukazujúci naň, ak je predchádzajúci blok voľný, zväčší sa hlavičke hodnota o hodnotu pôvodného, ktorý sa neskôr znuluje.

Ukážka kódu:

```
if (f > 4) {
        char* prev;
        prev = findPrev(a);
        char* d = (char*)prev + sizeof(int);// Smernik rozhodujuci o obsadenosti
        if (*d == 1) {
                *(int*)prev += *a + sizeof(int) + 1;
                *a = 0;
                *((char*)a + sizeof(int)) = 0;
        }
}
```

4 Testovanie

Na testovanie všetkých potrebných scenárov sme vytvorili pomocnú funkciu, ktorej parametrami sú rôzne premenné, ktoré ovplyvňujú situácie, ktoré chceme testovať.

Funkcia *memory_test* má ako parametre región, smerník na pole, znak pre rovnako veľké bloky, minimálnu a maximálnu veľkosť bloku, minimálnu a maximálnu veľkosť regiónu.

Príklad: `memory_test(region, pointer, 0, 8, 24, 180, 200);`

Program najskôr postupne alokuje región až kým nie je možné ďalej alokovať v ideálnom riešení, potom testuje správnosť každého smerníka, ak je správny, tak ho uvoľní. Na záver program vypočíta a vypíše percento obsadenosti oproti ideálnemu riešeniu bez vnútornej alebo vonkajšej fragmentácie.

Ukážka kódu:

```
while (allocated <= random_memory - block_min) {
	random_blk = (rand() % (block_max - block_min + 1)) + block_min;
	if (allocated + random_blk > random_memory)
		continue;
	allocated += random_blk;
	allo_count++;
	pointer[i] = memory_alloc(random_blk);
	if (pointer[i]) {
		i++;
		mallo_count++;
		mallocated += random_blk;
	}
}
for (int j = 0; j < i; j++) {
	if (memory_check(pointer[j])) {
		memory_free(pointer[j]);
	}
	else {
		printf("ERROR: Smernik bol poskodeny.\n");
	}
}
```

Testovacie scenáre prevedené do volania funkcie *memory_test* vyzerajú nasledovne:

```
printf("Testovanie 1:\n");
memory_test(region, pointer, 1, 8, 8, 40, 50);
memory_test(region, pointer, 1, 12, 12, 80, 100);
memory_test(region, pointer, 1, 24, 24, 180, 200);

printf("\nTestovanie 2:\n");
memory_test(region, pointer, 0, 8, 24, 40, 50);
memory_test(region, pointer, 0, 8, 24, 80, 100);
memory_test(region, pointer, 0, 8, 24, 180, 200);

printf("\nTestovanie 3:\n");
memory_test(region, pointer, 0, 500, 5000, 1000, 5000);
memory_test(region, pointer, 0, 500, 5000, 5000, 50000);
memory_test(region, pointer, 0, 500, 5000, 50000, 100000);

printf("\nTestovanie 4:\n");
memory_test(region, pointer, 0, 8, 50000, 1000, 5000);
memory_test(region, pointer, 0, 8, 50000, 5000, 80000);
memory_test(region, pointer, 0, 8, 50000, 80000, 100000);
```

5 Výsledky

5.1 Bod 1

V tomto bode testujeme rovnako veľké bloky malej veľkosti. Výpis z konzoly vyzerá nasledovne:

Testovanie 1:

Pamat o velkosti: 43 bajtov

Rovnake bloky o velkosti: 8 B

Alokovane bloky: 60.000%

Alokovane bajty: 24 (55.814%)

Pamat o velkosti: 81 bajtov

Rovnake bloky o velkosti: 12 B

Alokovane bloky: 66.667%

Alokovane bajty: 48 (59.259%)

Pamat o velkosti: 199 bajtov

Rovnake bloky o velkosti: 24 B

Alokovane bloky: 75.000%

Alokovane bajty: 144 (72.362%)

Zhodnotenie:

Výsledky sú ovplyvnené náhodnou veľkosťou poľa. Tým, že hlavička regiónu má konštantnú veľkosť, program alokuje rovnako veľké bloky, iba veľkosť poľa rozhodne o percentách. (Príklad na architektúre, kde *sizeof(int) == 4B*, hlavička regiónu je 4B, hlavička bloku je 5B, blok je 8B. Ideálne percentá dostaneme ak je región veľký (13*n+4)B, $n \in N$.

5.2 Bod 2

Podobné testovanie ako v bode jeden, len bloky sú nerovnakých veľkostí. Výpis z konzoly vyzerá nasledovne:

Testovanie 2:

Pamat o velkosti: 45 bajtov

Alokovane bloky: 66.667%

Alokovane bajty: 24 (53.333%)

Pamat o velkosti: 99 bajtov

Alokovane bloky: 71.429%

Alokovane bajty: 64 (64.646%)

Pamat o velkosti: 196 bajtov

Alokovane bloky: 75.000%

Alokovane bajty: 146 (74.490%)

Zhodnotenie:

Výsledky sú znovu ovplyvnené náhodnou veľkosťou poľa. Bloky sú rôzne, to znamená, že cyklus alokuje dokým je miesto aspoň na minimálny blok. Môže sa stať, že sa zaplní miesto presne na bajt presne, je to však ovplyvnené náhodnosťou veľkosti blokov. Čím väčšie sú bloky, tým menšia je hlavička, zároveň je menej hlavičiek, z toho vyplýva, že percentá budú väčšie.

5.3 Bod 3

Znovu testujeme nerovnako veľké bloky, ktoré sú väčšie ako v predošlých prípadoch. Výpis z konzoly vyzerá nasledovne:

Testovanie 3:

Pamat o velkosti: 1963 bajtov

Alokovane bloky: 100.000%

Alokovane bajty: 1710 (87.112%)

Pamat o velkosti: 30180 bajtov

Alokovane bloky: 90.909%

Alokovane bajty: 29344 (97.230%)

Pamat o velkosti: 92706 bajtov

Alokovane bloky: 100.000%

Alokovane bajty: 92295 (99.557%)

Zhodnotenie:

Na tomto teste môžeme pozorovať ako s zväčšujúcou sa veľkosťou regiónu stúpa počet percent. Hlavičky sú rovnako malé a bloky sú veľké najmenej 500B, toto kritérium pôsobí aj negatívne. Ak v pamäti vznikne miesto menšie ako 505B, program nemôže alokovať a percentá klesajú.

5.4 Bod 4

Testujeme alokáciu nerovnako veľkých blokov, ktorých veľkosti sú od najmenších možných po veľmi veľké. Výpis z konzoly vyzerá nasledovne:

Testovanie 4:

Pamat o velkosti: 2701 bajtov

Alokovane bloky: 83.333%

Alokovane bajty: 2627 (97.260%)

Pamat o velkosti: 23561 bajtov

Alokovane bloky: 81.818%

Alokovane bajty: 23506 (99.767%)

Pamat o velkosti: 90258 bajtov

Alokovane bloky: 72.727%

Alokovane bajty: 90212 (99.949%)

Zhodnotenie:

Posledné testovanie je hlavne ovplyvnené počtom alokovaných blokov. Nemôže sa stať, že vznikne menšie miesto, ako minimálny blok, t.j. 8B. Pokles percent je spôsobený množstvom hlavičiek, čím viac hlavičiek, tým menej percent.

Najhorší scenár môže nastať, ak alokujeme rovnako veľké 8B bloky, percentá budú na hodnote 61,538% alebo menej, záleží od počtu voľných bajtov na konci poľa (0B-7B). V našom prípade nezávisle od veľkosti regiónu, máme hlavičku o veľkosti *sizeof(int)*, ktorá tieto percentá ešte zníži.

5.5 Časová a pamäťová náročnosť

Najhorší scenár (worst case): nech $O_t(x)$ je časová náročnosť a $O_m(y)$ je pamäťová náročnosť

memory_init

$O_t(1)$

$O_m(1)$

findBest

$O_t(n)$

$O_m(1)$

memory_alloc

$O_t(1)$ + náročnosť *bestFit*

$O_m(1)$ + náročnosť *bestFit*

memory_check

$O_t(1)$

$O_m(0)$

findPrev

$O_t(n)$

$O_m(1)$

memory_free

$O_t(1)$ + náročnosť *findPrev*

$O_m(1)$ + náročnosť *findPrev*

Celková zložitosť:

Časová náročnosť celého programu - $O_t(n)$ (lineárna)

Pamäťová náročnosť celého programu - $O_m(1)$ (konštantná)

Záver

Naše metódy použité pri vytváraní funkcií *inti, alloc, check* a *free* neboli ideálne vo všetkých testovaných scenároch. Pri väčších veľkostiach pamäte regiónu by bol optimálnejší explicitný list. Pri malých veľkostiach (do 200B) je implicitný vhodnejší, keďže hlavičky nezaberajú toľko pamäte.

Ako spôsob nájdenia voľného bloku sme použili najlepší vhodný, týmto spôsobom síce zväčšujeme časovú náročnosť, ale program je efektívnejší v alokácii pamäte. Zároveň v menej prípadoch dochádza ku fragmentácii.

Hlavička bloku sa mohla naprogramovať aj inak. Premiestnenie informácie o obsadenosti, t.j. jedného bajtu, do časti hlavičky, ktorá hovorí o veľkosti mohlo byť realizovateľné pomocou negatívneho znamienka. Síce ušetrili bajt, ale veľkosť hlavičiek by bola obmedzená na polovicu.

Výsledky v percentách alokovaných blokov alebo bajtov boli blízke predpokladaným. Pri náhodnom priraďovaní veľkosti blokov boli výsledky lepšie, ako najhoršie možné. Tie sa odvíjajú od veľkosti minimálneho bloku. Najhorší možný scenár je nasledovný.

$$(X - a) * \frac{n}{n + a + 1}$$

Kde *X* predstavuje veľkosť regiónu, ktorý chceme inicializovať, *a* predstavuje *sizeof(int)*, čo boli v našom prípade 4 bajty, *n* predstavuje veľkosť minimálneho bloku.

Situácie, kde sa podarilo alokovať sto percent blokov oproti ideálnemu riešeniu mohli vzniknúť len pri testovaní, kde veľkosť minimálneho bloku bola veľká (Bod 3).

Prípady, kde sa veľkosť percent využitej pamäte blížila ku sto percentám, boli spôsobené alokovaním malého počtu veľkých blokov, t.j. vzniklo málo bajtov potrebných na uloženie údajov hlavičky bloku.

Bibliografia

Kohútka, L. (24. február 2021). Správa pamäte pri vykonávaní programu. Dostupné na Internete: dokumentový sterver is.stuba.sk, DSA_02