Oliver Leontiev
DSA, Zadanie 1
2020

# Zadanie 1 – Správca pamäti

Pri riešení zadania bola použitá metóda 2, teda jeden explicitný zoznam voľných blokov. Zoznam je jednosmerný (single-link) a je udržiavané poradie blokov tak, aby adresy išli vzostupne za sebou. Na začiatku zoznamu je hlavná hlavička (master_header), v ktorej je uložený smerník na prvý voľný blok (4B) a smerník na koniec pamäte (4B). Spôsob alokácie je best-fit, pre minimalizovanie fragmentácie. Štruktúra blokov je nasledovná:

**Voľný blok:**

**|** velkosť (4B) **|** smerník na další (4B) **|** voľný priestor **|**

**Alokovaný blok:**

**|** velkosť (2B) **|** priestor na uživateľove dáta **|**

**Štruktúri pre hlavičky blokov**: (* *header* v skutočnosti predstavuje *size*)

```
typedef struct freeblock {
	unsigned int header;
	struct freeblock *next;
}FREEBLOCK;

typedef struct fullblock {
	unsigned short header;
}FULLBLOCK;
```

Pri alokácii voľného bloku sa alokuje aj smerník na další a 2 bajty z velkosti. Na zápis veľkosti alokovaného bloku stačí unsigned short (2B) lebo zo zadania vyplýva, že najväčší možný blok je 50 000 bajtov. Pri voľných blokoch to už neplatí a potrebujeme unsigned int (4B).

Vďaka tomu, že zoznam je zoradený podľa adries nepotrebujeme pätu pre spájanie susedných blokov. Vysvetlené pri funkcii memory_free.

## Funkcie:

### void memory_init(void *ptr, unsigned int size)

Vytvorí hlavnú hlavičku (master_header) na začiatku pamäte a zapíše do nej smerník na nasledujúci voľný blok (na začiatku je hneď za hlavnou hlavičkou) a smerník na koniec pamäte (podľa zadanej veľkosti). Do prvého voľného bloku zapíše jeho velkosť (celková pamäť – 8 bajtov pre hlavnú hlavičku) a smerník na NULL, čo značí, že je zatiaľ posledný voľný blok.

Časová zložitosť je konštantná – O(1).

### void *memory_alloc(unsigned int size)

Voľný blok, ktorý ukazuje na NULL je posledný. Ak ukazuje na NULL master header, tak je pamäť plná a vrátim 0. Následne, ak master header ukazuje na nejaký voľný blok, zistím či je dostatočne veľký na alokovanie požadovanej pamäte aj s hlavičkou. Ak áno, označím ho zatiaľ ako najvhodnejší (best) a master header ako predok najvhodnejšieho (best_prev). Následne prejdem všetky voľné bloky a hľadám najvhodnejší pre alokáciu. Najvhodnejší je najmenší blok do ktorého sa zmestí požadovaná pamäť s hlavičkou.

```
while (p->next != NULL) {
		prev = p;
```

Oliver Leontiev
DSA, Zadanie 1
2020

```
            p = p->next;
            if ((p->header >= size + sizeof(FULLBLOCK))
                  && ( (best == NULL) || (best->header > p->header ) ))
            {
                  best = p;
                  best_prev = prev;
            }
      }
```

Následne sa s vybraním voľným blokom, jeho predchodcom a požadovanou pamäťou zavolá funkcia:

```
void addblock(FREEBLOCK *p, FREEBLOCK* prev, unsigned short size)
```

Táto funkcia prepíše hlavičku a presmeruje smerník predchádzajúceho voľného bloku. Ak by po alokácií zostalo vo voľnom bloku miesto aspoň na uloženie hlavičky voľného bloku (8B), tak sa voľný blok rozdelí a predchádzajúci sa nasmeruje na tento zvyšok (remainder). Zvyšok potom bude ukazovať na nasledujúci voľný blok. Inak sa blok alokuje celý a predchádzajúci bude ukazovať na ďalší voľný blok.

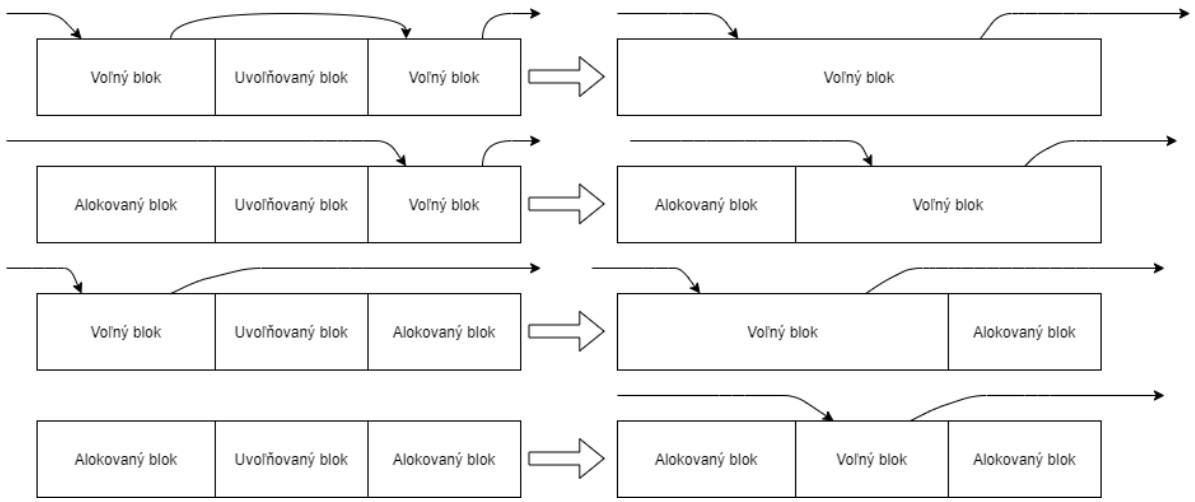
Vráti sa smerník na začiatok alokovaného bloku. Časová zložitosť je lineárna, O(n) kde n je počet voľných blokov. Best case je O(1), ak je pamäť plná.

**int memory_free(void *valid_ptr)**

Hľadáme najbližší voľný blok pred alokovaným blokom, ktorý chceme uvoľniť. Vďaka tomu vieme zistiť aj nasledujúci (predošlý na neho ukazuje). Teda poznáme voľné bloky, medzi ktorými sa náš alokovaný blok nachádza. Vieme veľkosti všetkých troch blokov, a teda vieme zistiť či sú vedľa seba pre potenciálne spájanie. Vďaka tomu, že sme zachovali poradie adries sme nepotrebovali päty ani double-link, a ušetrili sme miesto. S predchádzajúcim voľným blokom (alebo master headerom ak žiadny nie je) a smerníkom na hlavičku uvoľnovaného bloku zavoláme funkciu:

```
void freeblock(FULLBLOCK* p, FREEBLOCK* prev)
```

Táto funkcia má veľa podmienok, ale vlastne iba vyhodnocuje, ktorý prípad (case) platí pre uvoľňovaný blok a podľa toho spája bloky a presmeruváva smerníky. Musíme rozlišovať, či je predchádzajúci blok master header alebo voľný blok, lebo master header má inú štruktúru. Okrem prípadov, kedy pred blokom je master header a po bloku je koniec pamäte rozlišujeme štyri základné situácie:



Oliver Leontiev
DSA, Zadanie 1
2020

Časová zložitosť je worst case O(n), kde n je počet voľných blokov, ak je uvoľňovaný blok za posledným volným blokom. Best case je O(1) ak je uvoľňovaný blok pred prvým volným blokom. Spájanie blokov prebieha v konštantnom čase.

**int memory_check(void *ptr)**

Najprv zistíme, či pointer ukazuje do našej pamäte. Iba kvôli tejto funkcii máme uložený smerník na koniec pamäti v hlavnej hlavičke (inak by sme ho nepotrebovali). Ak by sme totiž mali smerník, ktorý ukazuje za našu pamäť, prehľadávali by sme alokované bloky za posledným voľným blokom a nevedeli by sme kedy sme vyšli von z pamäte. Toto je jediná funkcia, ktorá prehľadáva alokované bloky a preto nám nestačí vedieť, ktorý voľný blok je posledný, ale musíme vedieť kde končí celková pamäť.

Ak smerník ukazuje do pamäti presunieme ho o 2 bajty dozadu na potenciálnu hlavičku a prechádzame voľné bloky aby sme zistili, ktorý voľný blok je najbližší pred ním. Ak počas toho zistíme, že smerník ukazuje na voľný blok rovno vrátime 0. Do premennej full potom uložíme prvý alokovaný blok za najbližším predchádzajúcim voľným blokom (alebo za master headerom, ak sme na začiatku pamäti).

```
if (master_header->next == NULL || master_header->next > (FREEBLOCK*)ptr)
		full = (FULLBLOCK*)(master_header+1);

	else {
		free = master_header->next;
		while (free->next != NULL && free->next <= (FREEBLOCK*)ptr) {
			free = free->next;
			if (free == (FREEBLOCK*)ptr) //
				return 0;
		}
		full = (FULLBLOCK*)((char*)free + free->header);
	}
```

Následovne prechádzame alokované bloky počínajúc full, kým nenájdeme pointer (vrátime 1) alebo ho neprekročíme (vrátime 0).

Časová zložitosť je best case O(1), keď smerník neukazuje do pamäti a worst case O(n), kde n predstavuje počet alokovaných blokov, ak je plná pamäť a smerník ukazuje do posledného alokovaného bloku. Väčšinou sa prejde niekoľko voľných blokov a niekoľko alokovaných blokov.

## Fragmentácia

Vnútorná fragmentácia je celkom malá, keďže réžia pre alokovaný blok je len 2 bajty. Najmenší voľný blok, ktorý tento spôsob požaduje je 8 bajtov, aby sa zmestila hlavička. Najmenší blok, ktorý sa alokuje podľa zadania je 8 bajtov (10 bajtov s hlavičkou). Teda dá sa povedať že voľné bloky sú bez réžie navyše. Pri malých veľkostiach celkovej pamäte zaváži hlavne 8 bajtov, ktoré potrebujeme na hlavnú hlavičku.

Vonkajšia fragmentácia je mierne vyššia, keďže voľné bloky vieme spájať iba ak sú vedľa seba. Môžeme napríklad alokovať, uvoľňovať a znova alokovať bloky tak, že nám zostane zvyšok o 1 bajt menší ako najmenší uvoľniteľný blok (7 bajtov). Tento priestor ostane nevyužitý, aj ak sa susedný (nasledujúci) blok uvoľní, lebo nebolo možné označiť ho ako voľný blok.

## Testovanie

Oliver Leontiev
DSA, Zadanie 1
2020

**Test 1** – rovnaké bloky malej veľkosti (8 – 24 bajtov), malá celová pamäť (50, 100, 200 bajtov)

Pri prvom teste sme zaplnili pamäť rovnako veľkými blokmi pre každú veľkosť bloku (8 – 24 bajtov), pri každej veľkosti, ak sa už rovnaký blok nezmestil, zmenšili sme bloky, ktoré alokujeme o 1 bajt (najmenej 8 bajtov). Po zaplnení pamäti sme 2 bloky náhodne uvoľnili a skúsili alokovať 8 bajtov, kým sa nezaplnila pamäť znova. Ideálny prípad sme popritom simulovali na čísle, kde alokáciu rezprezentovalo pripočítavanie veľkosti bloku, kým sme presiahli veľkosť celkovej pamäti. Pre každú veľkosť pamäte sme test vyhodnotili osobitne.

```
TEST 1
PAMAT 50 BYTOV:
Percento alokovanej pamate oproti idealnemu pripadu:    66.33%
PAMAT 100 BYTOV:
Percento alokovanej pamate oproti idealnemu pripadu:    73.70%
PAMAT 200 BYTOV:
Percento alokovanej pamate oproti idealnemu pripadu:    80.76%
```

Z testu vyplýva, že s menšou celkovou pamäťou klesá priestorová efektivita. Najmä kvôli tomu, že hlavná hlavička a hlavičky alokovaných blokov, zaberajú pri 50 bajtoch pamäte veľkú časť. Test dáva rôzne výsledky, kvôli nahodnému uvoľňovaniu ale rozptyl je malý.

**Test 2** – nerovnaké bloky malej veľkosti (8 – 24 bajtov), malá celová pamäť (50, 100, 200 bajtov)

Druhý test používa podobný princíp ako prvý, ale neprechádzame postupne všetkými veľkosťami blokov. Namiesto toho ich náhodne generujeme, a pre každú veľkosť pamäte zaplníme pamät len raz. Taktiež bloky postupne nezmenšujeme, namiesto toho ak sa náhodný blok nezmestí, alokujeme 8 bajtové bloky do zaplnenia pamäte. Dva alokované bloky opäť náhodne uvoľníme a následne znova alokujeme 8 bajtové bloky do zaplnenia pamäte.

```
TEST 2
PAMAT 50 BYTOV:
Percento alokovanej pamate oproti idealnemu pripadu:    66.67%
PAMAT 100 BYTOV:
Percento alokovanej pamate oproti idealnemu pripadu:    74.74%
PAMAT 200 BYTOV:
Percento alokovanej pamate oproti idealnemu pripadu:    75.63%
```

Výsledok je veľmi podobný testu 1, ale s oveľa vačším rozptylom medzi spusteniami. Taktiež tu môže využitie pamäte klesnúť veľmi nízko (50 % pre 50 bajtovú pamäť), kvôli neštastnému uvoľneniu a alokovaniu (napr. uvoľní sa 15 bajtov, alokuje 8 – stratili sme 7 bajtov).

**Test 3** – nerovnaké bloky väčšej veľkosti (500 - 5 000 bajtov), veľká celová pamäť (10 000 a 100 000 bajtov)

Tretí test prebieha analogicky ako test 2, ale náhodne generujeme veľkosti z intervalu 500 – 5 000 bajtov a uvoľnujeme 5 blokov. Keď sa už náhodný blok nezmestí alebo po uvoľňovaní, alokujeme bloky 500 bajtov do zaplnenia pamäte.

Oliver Leontiev
DSA, Zadanie 1
2020

```
TEST 3
PAMAT 10000 BYTOV:
Percento alokovanej pamate oproti idealnemu pripadu:    94.85%
PAMAT 100000 BYTOV:
Percento alokovanej pamate oproti idealnemu pripadu:    99.00%
```

Priestorová efektivita je veľmi vysoká, väčšie bloky znamená menej celkovej réžie a menšia pravdepodobnosť vonkajšej fragmentácie.

**Test 4** – nerovnaké bloky rôznej veľkosti (8 - 50 000 bajtov), veľká celová pamäť (100 000 bajtov)

Vo štvrtom teste alokujeme náhodné bloky z intervalu 8 - 50 000 bajtov. Ak sa blok nezmestil, vyplníme pamäť 8 bajtovými blokmi. Následne 5 náhodných blokov uvoľníme a znova vyplníme pamäť 8 bajtovými blokmi.

```
TEST 4
PAMAT 100000 BYTOV:
Percento alokovanej pamate oproti idealnemu pripadu:    99.46%
```

Tu už je využitie pamäte veľmi vysoké, hlavne kvôli veľkej celkovej pamäti (hlavná hlavička zaberá mizivé percento) a veľkým blokom (menej blokov, menej celkovej réžie).