Fakulta informatiky a informačných technológií STU v Bratislave Ilkovičova 2, 842 16 Bratislava 4

Tibor Dulovec

Zadanie 1 – Správca pamäti

DSA LS 2020/21
Cvičiaci P. Lehoczký
Pondelok 11:00 – 12:50

Obsah

Opis programu 3
Hlavička bloku 3
Zložitosť algoritmov 3
Časová náročnosť: O(n) 3
Pamäťová náročnosť: O(n) 3
Opis funkcií 3
memory_init 3
memory_alloc 3
memory_free 4
memory_check 5
Testovanie programu 6
Test1: Prideľovanie malých blokov rovnakej veľkosti 6
Test2: prideľovanie blokov rôznej malej veľkosti 7
Test3: pridávanie rôzny počet náhodne veľkých blokov 8
Test4: Uvoľňovanie a spájanie blokov 9
Test5: funkcia memory_check 10

Opis programu

Program slúži na správu pamäte. Je naprogramovaný spôsobom explicitného zoznamu. Voľné bloky som spojené jednosmerne v spájaným zozname. To z dôvodu, aby hlavičky boli o 8 bytov pamäte menšie. Časová náročnosť programu je kvôli tomu síce o niečo väčšia, ale získali sme tím viac pamäte. Obzvlášť pri malých blokov je menšia hlavička výrazne viac oceniteľnejšia ako o niečo malo lepšia časová efektivita. Voľné bloky pri sebe sa spájajú a tým sa docieľuje menší počet väčších blokov.

Hlavička bloku

```
struct HEAD {
    int size;
    struct HEAD *next;
};
```

Hlavička bloku je štruktúra, ktorá má veľkosť 16 bytov. Je v nej uložený ukazovateľ na ďalší voľný blok a jeho veľkosť.

Na rozoznanie voľného bloku od obsadeného sa používa znamienko. Ak je veľkosť bloku záporná, ide o voľný blok. Ak je kladná, blok je už alokovaný.

Zložitosť algoritmov

Časová náročnosť: O(n)

Celý zoznam prehľadávame lineárne a tak je časová náročnosť lineárna.

Pamäťová náročnosť: O(n)

Pamäťová náročnosť je taktiež lineárna.

Opis funkcií

memory_init

Funkcia inicializuje pamäť. Je potrebná aby táto funkcia bola zavolaná pred ostatnými funkciami určenými na správu pamäte. Funkcia začiatku poľa vytvorí prvý blok, ktorý bude vždy ukazovať na voľný blok, ktorý sa bude používať pre alokovanie. Veľkosť tohto prvého bloku je podľa veľkosti pamäte, ktorá je zadaná pre program.

```
memoryStart = ptr;
struct HEAD *memoryHead = (struct HEAD *) ptr;
struct HEAD *firstBlock = (struct HEAD *) ((char *) ptr + sizeof(struct HEAD));
memoryHead->next = firstBlock;
```

Obrázok 1 vytvorenie hlavičky, ktorá odkazuje na prvý voľný blok. Tento blok bude následne definovaný

memory_alloc

Funkcia začína vypočítanie potrebnej veľkosti. A to spočítaním vstupným parametrom, ktorý prezentuje požadovanú veľkosť, s veľkosťou hlavičky.

Túto potrebnú veľkosť použijeme ako argument pre funkciu **findBlock**. Ktorá nam za pomocou princípu **best fit** nájde v celom zozname najmenší, ale dostatočne veľký blok. Časová náročnosť je kvôli best fit vyššia(**O(n)**), ale vzhľadom na použitie explicitného zoznamu, ktorý uchováva iba voľné bloky to nemusí byť problém. Pretože voľných blokov nemusí byť veľa.

Ak požadovaná veľkosť pre alokovaný blok je presne taká, aká je veľkosť voľného bloku, ktorý nám vrátila funkcia findBlock, tak sa tento blok priamo vyberie z listu. V opačnom prípade, ktorý bude určite častejší, sa voľný blok rozdelí. A to na požadovanú veľkosť a zbytok voľnej časti, ktorá sa môže použiť neskôr. Nový vytvorený alokovaný blok sa vráti ako ukazovateľ na začiatok payloadu.

```
// create new allocated block
struct HEAD *allocatedBlock = foundFreeBlock;
allocatedBlock->size = (int) allocateSize;
allocatedBlock->next = NULL;

// return pointer
return (char *) allocatedBlock + sizeof(*allocatedBlock);
```

Obrázok 2 vracanie novo vytvoreného alokovaného bloku

memory_free

Funkcia uvoľňuje už alokovanú časť pamäte. Vstupom pre túto funkciu je ukazovateľ na alokovaný blok. Veľkosť bloku sa najprv zmení na záporné číslo, čo symbolizuje to, že blok je znova voľný. Na konci funkcie sa uvoľnený pridá na začiatok zoznamu voľných blokov.

Pred tým sa ale overí, čí je blok pred ním alebo za ním voľný. Ak áno, bloky sa spoja a tým vytvoria väčší blok pre budúcu alokáciu. Týmto výrazne zlepšíme pamäťovú efektívnosť a zmenšíme fragmentáciu, pretože vytvárame menej väčších blokov miesto veľa menších blokov.

```
// Check if next block is free. If block is free, merge it to this block
struct HEAD *blockAfter = (struct HEAD *) ((char *) blockToFree + blockToFree->size * -1 + sizeof(struct HEAD));

if (blockAfter->size < 0) { // if block is free
    blockToFree->next = blockAfter->next;
    blockToFree->size += blockAfter->size - sizeof(struct HEAD);

    // If blockAfter is first in memory.. We move blockToFree to beginning of memory
    if (((struct HEAD *) memoryStart)->next == blockAfter) {
        ((struct HEAD *) memoryStart)->next = blockToFree;
    }
}
```

Obrázok 3 spájanie bloku v prípade, že nasledujúci blok je voľný

memory_check

Funkcia, ktorá kontroluje, či sa ukazovateľ nachádza v alokovanom poli. Prechádza každým blokom od začiatku pamäte pokiaľ sa nenájde ukazovateľ, ktorý je ako vstupný parameter, vo voľnom bloku. V opačnom prípade predpokladáme, že ukazovateľ je v alokovanom poli.

```
actualBlock = (struct HEAD *) ((char *) actualBlock + sizeof(struct HEAD) + positiveSize);
```

Obrázok 4 koniec while cyklu zabezpečujúci posúvanie sa na ďalší blok

Testovanie programu

Test1: Prideľovanie malých blokov rovnakej veľkosti

Funkcia testuje prideľovanie blokov rovnakej malej veľkosti(16) pre pamäť o veľkosti 200bitov.

Veľkosť neje dostatočná pre šiesty blok a tak sa alokuje iba 5 zo 6tich blokov. To je 83.33%.

```
void test1() {
    int memorySize = 200;
    char region[memorySize];
    char *pointers[1000];
    float allocatedPart = 0;
    float mallocatedPart = 0;
    memory_init(&region, memorySize);

    for (int i = 0; i < 6; ++i) {
        allocatedPart++;
        pointers[i] = memory_alloc( size: 16);
        if (pointers[i]) {
            mallocatedPart++;
        }
    }

    printBlockUsage(memorySize, mallocatedPart, allocatedPart);
}
```

```
Size of memory: 200 bytes
Allocated blocks: 83.33%
```

Test2: prideľovanie blokov rôznej malej veľkosti

Funkcia testuje prideľovanie blokov náhodnej veľkosti medzi 8 až 24 bytov pre pamäť o veľkosti 200bitov. Výsledok je vždy rozdielna, vzhľadom na to, že sa nedá predpovedať aká veľkosť bytov bude. Ale môžeme predpokladať, že nikdy nebude 100%, keďže ak by mali všetky bloky minimálnu veľkosť, tak sa do pamäte aj tak nezmestia.

```
void test2() {
    srand( _Seed: time( _Time: 0));
    int memorySize = 200;
    char region[memorySize];
    char *pointers[1000];
    float allocatedPart = 0;
    float mallocatedPart = 0;
    memory_init(&region, memorySize);

    for (int i = 0; i < 11; ++i) {
        allocatedPart++;
        int randomSize = (rand() % (24 - 8 + 1)) + 8;;
        pointers[i] = memory_alloc(randomSize);
        if (pointers[i]) {
            mallocatedPart++;
        }
    }

    printBlockUsage(memorySize, mallocatedPart, allocatedPart);
}
```

```
Size of memory: 200 bytes
Allocated blocks: 54.55%
```

Test3: pridávanie rôzny počet náhodne veľkých blokov

Pridávanie náhodne veľkých blokov do pamäte. Počet blokov je náhodný, pridáva sa kým je dostupná veľkosť. Obsadenosť blokov oproti ideálnemu počtu tu môže byť nižšia. To hlavne z dôvodu, že posledné pridané bloky môžu byť obrovské vzhľadom na celkovú veľkosť pamäte.

```
Memory size: 30000
Size of block is between 100 to 1000
Block with size 977 was allocated on address 6383936
Block with size 406 was allocated on address 6384929
Block with size 829 was allocated on address 6385351
Block with size 300 was allocated on address 6386196
Block with size 344 was allocated on address 6386512
```

```
Block with size 251 was allocated on address 6411178
Block with size 716 was allocated on address 6411445
Block with size 484 was allocated on address 6412177
Block with size 433 was allocated on address 6412677
Block with size 643 was allocated on address 6413126
Allocated blocks: 93.75%
```

(Snímok so všetkými vypísanými blokmi som nezobrazil z dôvodu že ich je veľké množstvo)

Test4: Uvoľňovanie a spájanie blokov

Test4 kontroluje konkrétny scenár. Testuje spájanie voľných blokov, ktoré sú vedľa seba. To z dôvodu, nech sa do pamäte zmestí viac väčších blokov. 11

```
void test4() {
    int memorySize = 400;
    char region[memorySize];
    memory_init(&region, memorySize);

    char *pointer1 = (char *) memory_alloc( size: 15);
    char *pointer2 = (char *) memory_alloc( size: 40);
    char *pointer3 = (char *) memory_alloc( size: 20);
    printf( _Format: "Pointer3 have address %d\n", pointer3);
    char *pointer4 = (char *) memory_alloc( size: 20);
    printf( _Format: "Pointer4 have address %d\n", pointer4);
    char *pointer5 = (char *) memory_alloc( size: 15);
    printf( _Format: "Pointer5 have address %d\n", pointer5);

    if (memory_check(pointer3)) {
        memory_free(pointer3);
        if (!memory_check(pointer3)) printf( _Format: "Block 3 is now free\n");
    }
    if (memory_check(pointer4)) {
        memory_free(pointer4);
        if (memory_check(pointer4)) printf( _Format: "Block doesn't exist\n");
    }
    char *pointer6 = (char *) memory_alloc( size: 25);
    printf( _Format: "Pointer6 have address %d\n", pointer6);

}
```

V tomto prípade alokujeme 5 ukazovateľov a potom dva z nich o veľkosti 20 uvoľníme. Funkciou memory_check ešte overíme, že či ten blok, ktorý sa mal spojiť náhodou neexistuje.

Spájanie blokov spôsobí to, že blok na ukazovateli 3 bude o veľkosti 40 a nový ukazovateľ 6 sa zmesti do daného bloku. Keby sa bloky nespájali, pamäť by mohla obsahovať veľa voľných malých blokov, čo by tiež spôsobovalo väčšiu fragmentáciu.

```
Pointer3 have address 6421623
Pointer4 have address 6421659
Pointer5 have address 6421695
Block 3 is now free
Block 4 doesn't exist
Pointer6 have address 6421623
```

Test5: funkcia memory_check

Test kontroluje scenáre s overovaním pamäte. Je otestované aj náhodne umiestnenie mimo hlavičku v poli. Či už v alokovanej oblasti alebo voľnej oblasti.

```
void isPointerValid(void *ptr) {
    if (memory_check(ptr))
        printf( _Format: "Pointer is valid\n");
    else
        printf( _Format: "Pointer is not valid\n");
}

void test5() {
    int memorySize = 400;
    char region[memorySize];
    memory_init(&region, memorySize);

    char *pointer = (char *) memory_alloc( size: 25);
    isPointerValid(pointer);
    isPointerValid( ptr: pointer + 5);

    memory_free(pointer);
    isPointerValid(pointer);
    isPointerValid( ptr: pointer + 5);
}
```

```
Pointer is valid
Pointer is valid
Pointer is not valid
Pointer is not valid
```