Zadanie 1 – Správca pamäti

Filip Vida

Algoritmus

Rozhodol som sa pre explicitné zoznam voľných blokov pamäte.

Do hlavičky ukladám 2 údaje, údaj typu unsigned int kde sa uchováva veľkosť bloku pamäte a 4b adresu nasledujúceho voľného bloku pamäte. V hlavičke pre alokovanú pamäť 2. blok využívam ako overenie legitímnosti pointra na pamäť v memory_check().

Hlavička má veľkosť 8B a toto znamená že na alokovanie pamäte potrebujeme aspoň 9B(8 na hlavičku + 1 užívateľovi). Pokiaľ je voľný blok pamäti menší ako 9B tak sa pridáva k alokovanej pamäti aby nevznikol nevyužiteľný blok.

```
struct header {
    unsigned int size;
    struct header *nextFree;
};
```

memory_alloc()

Následne prejdeme zoznam voľných blokov a metódou best fit nájdeme vhodný blok kde miesto alokujeme. Hľadám najmenší rozdiel medzi veľkosťami alokovanej a voľnej pamäte.

```
while (current->nextFree != NULL) {
    if ((difference == -1 && current->size >= size + sizeof(struct headerAllocated))
        || (current->size >= size + sizeof(struct headerAllocated) && current->size - size < difference)) {
        difference = current->size - size;
        currentBestFit = current;
    }
    current = current->nextFree;
}
```

Následne sa dostaneme k dvom vetvám programu, ak po alokovaní blok voľnej pamäte ostane, a keď sa úplne zaplní.

```
//Ak po alokovani ostane volne miesto
if (newSize != size + sizeof(struct headerAllocated) && fragmentation == 0) {
    if (first == currentBestFit) {
        printf( _Format: "Po alokovani ostalo miesto a je to first\n");
        void *oldNext = first->nextFree;
        void *pointer = (void *) first;
        first = pointer + sizeof(struct headerAllocated) +
                size;
        first->size = newSize - size - sizeof(struct headerAllocated);
        first->nextFree = oldNext;
    } else {
        printf( _Format: "Po alokovani ostalo volne miesto a prvok nebol first");
        struct header *helper = first;
        while (helper->nextFree != current) {
            helper = helper->nextFree;
        }
        void *novyNext = current->nextFree;
        void *pointer = (void *) current;
        current = pointer + sizeof(struct headerAllocated) + size;
        current->size = newSize - size - sizeof(struct headerAllocated);
        current->nextFree = novyNext;
        helper->nextFree = current;
    }
```

Ukážka pre možnosti že sa blok celý nazaplní

Ak po alokovaní ešte v bloku voľnej pamäte miesto ostane, rozhodujeme či je tento blok first alebo nie. Ak blok po alokovaní pamäte celý vyplníme, je nutné ho celý vymazať a následne celý linked list upraviť.

```
//Ak vyplnime celu dostupnu pamet, nieje ziadna volna, vytvorime novy first aby sa nestratil ale dame ho na novu random adresu
if (current->nextFree == NULL && current == first) {...}
    //Ak je vyplnene miesto na konci linked listu ale nieje jeho zaciatok
else if (current->nextFree == NULL && current != first) {...}
    //Ak je vyplnene miesto na zaciatku linked listu a existuju este ine
else if (current->nextFree != NULL && current == first) {...}
    //Ak je vyplnene miesto v strede linked listu
else if (current->nextFree != NULL && current != first) {...}
printf( _Format: "Dostupna velkost je presne rovnako velka\n");
```

memory_free()

Na začiatku zavoláme momory_check() a zistíme či je daný ukazovateľ platný. Najjednoduchšia možnosť nastane ak je zoznam voľných oblastí prázdny, vtedy len vytvoríme nový first.

```
if (first->size == 0) {
    void *pointer = (void *) valid_ptr;
    int size = *(int *) (valid_ptr - 8);
    first = (void *) pointer - 8;
    if (first)
        memset(first, _Val: 17, _Size: size + 8);
    first->size = size + 8;
    first->nextFree = NULL;
    printf( _Format: "Neexistovalo volne miesto, uvolnil sa novy first\n");
    return 0;
```

Ak sa táto podmienka nesplní, hľadáme v poli voľných oblastí také, ktoré sa v pamäti nachádzajú presne pred alebo za oblasťou, ktorú uvoľnujeme.

```
while (current != NULL) {
    pointer1 = current;
    if (current == pointer + size) {
        //printf("Naslo sa pred pointrom");
        nasloPred = 1;
        prvokPred = current;
    } else if (pointer1 + current->size == pointer - 8) {
        //printf("Naslo sa za pointrom");
        prvokPo = current;
        nasloZa = 1;
    }
    current = current->nextFree;
}
```

Následne pomocou týchto 2 premenných zistíme v ktorej situácii sa nachádzame a vykonáme potrebné akcie.

```
    //Pripady pre najdene moznosti
    //Alokovana pamet je medzi 2 prvkami linked listu
    if (nasloPred == 1 && nasloZa == 1) {...}
        //Alokovana pamet je pred prvkom v linked listu
    else if (nasloPred == 1 && nasloZa == 0) {...}
        //Alokovana pamet je presne za prvkom v linked liste
    else if (nasloPred == 0 && nasloZa == 1) {...}
        //Alokovana pamet nieje spojena s volnou pametou
    else if (nasloPred == 0 && nasloZa == 0) {...}
```

Odhad výpočtovej (časovej) a priestorovej (pamäťovej) zložitosti

Funkcia memory_alloc()

Na začiatku každého memory alloc si prejdeme celý linked list – lineárna zložitosť O(N)
Podľa prípadu môžu následne nastať 2 možnosti :

- Operácia si vyžaduje nájsť prvok pred najdeným best fitom - lineárna zložitosť O(N)
- Operácia nevyžaduje nové prejdenie linked listu

Pri možnosti č. 1 je zložitosť O(N) + O(N), pri možnosti č. 2 je to O(N)

Funkcia memory_free()

Najlepší prípad nastane ak memory_free() zavoláme po tom, ako sme alokovali celú dostupnú pamäť, vytvoríme nový prvok v linked liste a funkcia končí

V iných prípadoch prechádzame celý linked list - lineárna zložitosť O(N)

Nasledujú 2 prípady

- Operácia si vyžaduje ešte raz prejsť linked list - lineárna zložitosť O(N)
- Operácia si to nevyžaduje

Funckia v najlepšom prípade žiadne cykly nevyužíva, môže byť O(N) alebo O(N) + O(N)

Pamäťová efektívnosť

Máme pamäť o veľkosti 100B

Alokujeme zóny 8B

Z 8 allocov prešlo úspešne 6, na 2 nebolo dosť miesta

48 B bolo alokovaných pre používateľov

Efektívne sme využili 96 B, po prirátaní hlavičky(6*8B)

Posledné 4 B boli priradené do pole6 aby nevznikla vonkajšia fragmentácia

Využitie pamäte 96%

```
char *region;
unsigned int initialSize = 100; //Velko
region = (char*) malloc(initialSize); /
memory_init(region, initialSize);
char *pole1 = memory_alloc( size: 8);
char *pole2 = memory_alloc( size: 8);
char *pole3 = memory_alloc( size: 8);
char *pole4 = memory_alloc( size: 8);
char *pole5 = memory_alloc( size: 8);
char *pole6 = memory_alloc( size: 8);
char *pole7 = memory_alloc( size: 8);
char *pole8 = memory_alloc( size: 8);
memory_free(pole5);
memory_free(pole1);
memory_free(pole3);
memory_free(pole2);
memory_free(pole4);
memory_free(pole6);
memory_free(pole7);
memory_free(pole8);
```

Vykonané testy

Test1 – Malé hodnoty, alokovanie po sebe rovnaké bloky

Test2 – Malé hodnoty, alokovanie striedame s free, rôzne hodnoty a vyznačené špecifické prípady funckie malloc(ukázané v kóde ako komentáre)

Test3 – väčšie hodnoty, alokované podobne velké hodnoty

Test4- väčšie hodnoty, alokované aj malé aj veľké hodnoty

Testy sa nachádzajú v programe ako funckie.