Zadanie 1 – Správca pamäte

Opis algoritmu

1. Funkcia memory_init

V tejto funkcii vložíme do vyhradenej pamäte z funkcie main hlavičku (na ktorú bude odkazovať globálna premenná v programe), pričom bude obsahovať veľkosť dostupnej pamäte a smerník na začiatok spájaného zoznamu blokov s voľnou pamäťou.

Obrázok 1

Následne je vložená aj hlavička pre prvý voľný blok v pamäti ako na Obrázok 1. Hlavička bloku obsahuje veľkosť dostupnej pamäte pre používateľa, smerník na predchádzajúci voľný blok a smerník na nasledujúci voľný blok. Hlavičky sú reprezentované ako štruktúry a sú používané na réžiu pamäte.

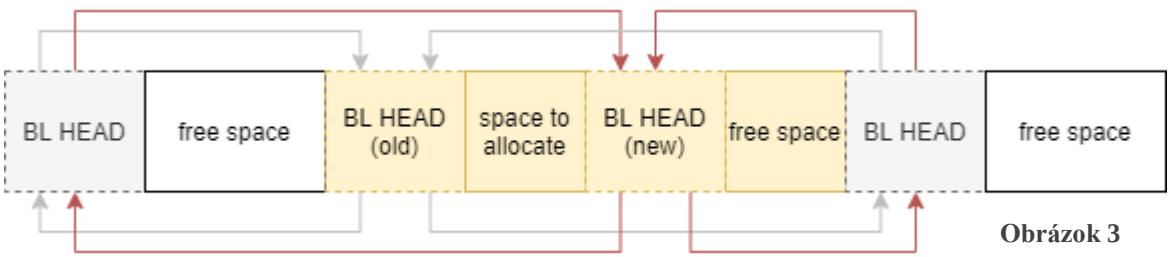
2. Funkcia memory_alloc

Prideľovanie blokov sa realizuje štýlom First fit, čiže funkcia najprv prechádza zoznam voľných blokov dokým nenájde prvý vyhovujúci blok s dostatočnou veľkosťou. Ďalej je vhodné blok rozdeliť na dve časti, aby sme zbytočne neplytvali pamäťou.

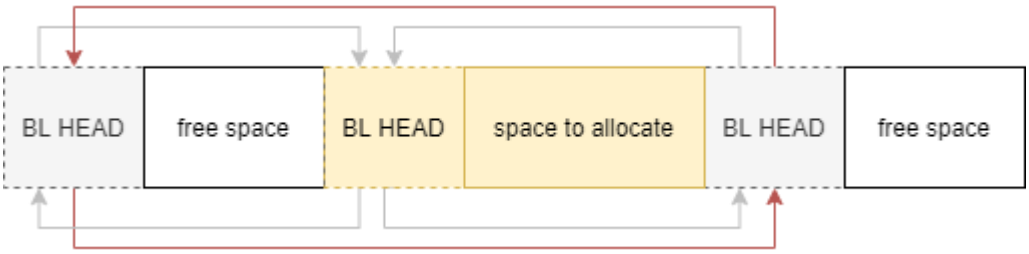
```
int remainingSize = fittable->size - size - sizeof(BL_HEAD);
```

Obrázok 2

Na to aby sme vedeli, či sa blok oplatí rozdeliť, je vhodné si to vypočítať. Ako zobrazuje Obrázok 2, od veľkosti bloku, ktorý sme našli odpočítame požadovanú veľkosť na alokovanie a veľkosť hlavičky bloku. Výsledkom je číslo, ktoré predstavuje veľkosť druhého bloku, čiže ak je 1 a väčšie tak sa to určite oplatí. Pre označenie obsadeného bloku bola zvolená metóda, kedy veľkosť daného bloku označíme ako zápornú. **Chyba! Nenašiel sa žiaden zdroj odkazov.** zobrazuje žltou farbou blok, ktorý sa ide deliť. Teraz je potrebné presmerovať predchádzajúce voľné bloky na nový blok a zároveň skopírovať odkazy na susedov zo starého bloku na nový blok. Na obrázku zobrazené šedé šípky predstavujú odkazy, ktoré sa zmenia na červené po rozdelení.



Obrázok 3



Obrázok 4

Ak sa blok neoplatí rozdeliť, stačí len presunúť odkazy z alokovaného bloku na predchádzajúci a nasledovný blok ako to ilustruje situácia na Obrázok 4.

3. Funkcia memory_check

Funkcia dostane ako argument pointer, ktorý treba overiť či je validný. Najprv treba overiť, či je pointer v rámci dostupnej pamäte. Hornú a dolnú hranicu možno identifikovať ako:

```
void* lower_limit = (void*)memory_head_ptr + sizeof(MEM_HEAD);
void* upper_limit = (void*)memory_head_ptr + memory_head_ptr->size + sizeof(MEM_HEAD);
```

Obrázok 5

V ďalšom rade treba overiť, či náhodou pointer neukazuje na hlavičku niektorej zo zoznamu voľných blokov. Na to nám poslúži nasledovná funkcia na **Chyba! Nenašiel sa žiaden zdroj odkazov.**, ktorá navyše kontroluje, či v hlavičke v položke size nie je náhodou kladné číslo, čo by znamenalo, že je blok voľný a teda nemôže sa uvoľniť.

```
int not_in_linkedlist(BL_HEAD* block_head) {
    BL_HEAD* temp = memory_head_ptr->linked_list;

    if(temp->size > 0)
        return 0;
    while(temp != NULL) {
        if(temp == block_head)
            return 0;
        temp = temp->next;
    }
    return 1;
}
```

Obrázok 6

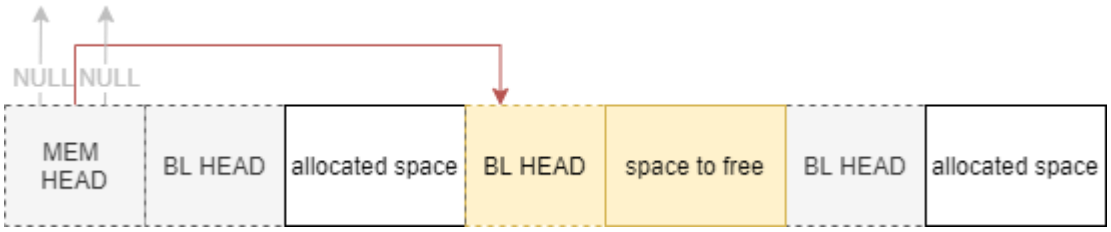
Ak všetky podmienky sedia môžeme prehlásiť pointer za platný.

4. Funkcia memory_free

Najskôr si veľkosť bloku ktorý chceme uvoľniť prenásobíme -1, teda bude označený kladným číslom. Štandardne môžu nastať 4 situácie:

*Šedé šípky zobrazujú predchádzajúci stav

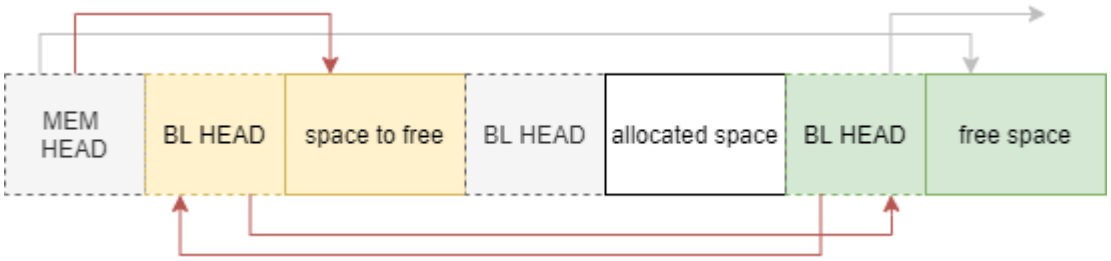
a) Chceme uvoľniť blok, pričom zoznam voľných blokov je prázdny:



Obrázok 7

Do hlavičky pre pamäť vložíme odkaz na tento blok.

b) Chceme uvoľniť blok, ktorý je pred prvým blokom v spájanom zozname:



Obrázok 8

Do bloku na uvoľnenie pridáme adresu ďalšieho (zeleného bloku), nastavíme aby hlavička pamäte ukazovala na žltý blok a zelenému bloku dáme adresu prechádzajúceho na žltý.

Môže nastať aj situácia, kedy budú 2 voľné bloky vedľa seba, potom prichádza na rad funkcia merge_block, ktorá dostane dva voľné bloky a spojí ich do jedného.

```
void merge_block(BL_HEAD* block_to_free, BL_HEAD* temp) {
```

Obrázok 9

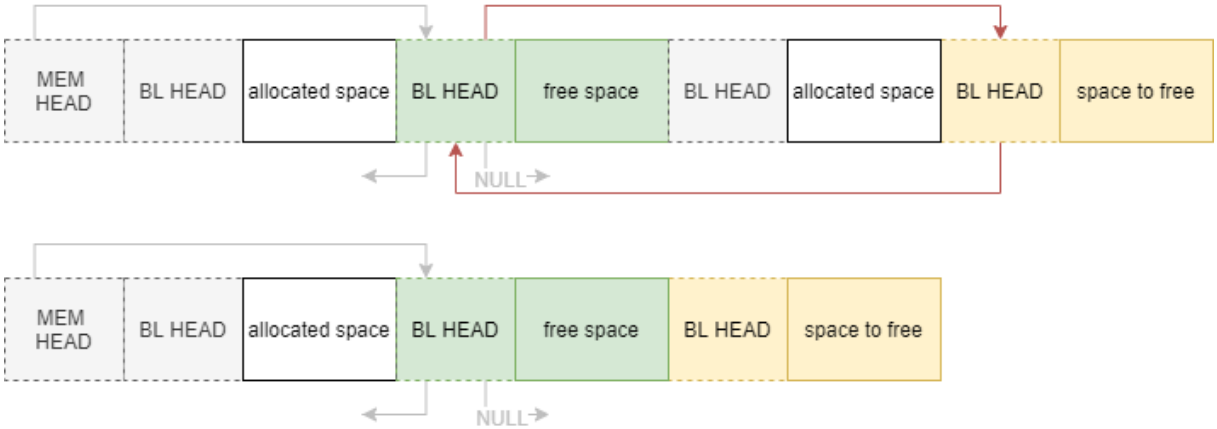
Keď bude blok, ktorý chceme uvoľniť naľavo:

```
if(block_to_free < temp) {
    block_to_free->size += temp->size + sizeof(BL_HEAD);
    temp->size = 0;
    //copy links to new merged block
    block_to_free->next = temp->next;
    if(temp->prev != block_to_free)
        block_to_free->prev = temp->prev;
    //set links of temp neighbors to new merged block
    if(temp->prev != NULL)
        temp->prev->next = block_to_free;
    /*if(block_to_free->next == temp)
        temp->prev->next = NULL;*/
    if(temp->next != NULL)
        temp->next->prev = block_to_free;
}
```

Obrázok 10

Spájanie je realizované, tak že údaje z bloku pravého sa presunú do ľavého, ktorý vlastne zväčší svoju veľkosť a nadobudne odkaz ďalšieho a predchádzajúceho voľného bloku z bloku napravo. Ešte samozrejme treba upraviť predchádzajúci a nasledovný blok pravého bloku, aby smerovali na náš zväčšený blok, ktorý je už naľavo.

c) Chceme uvoľniť blok, ktorý je už za posledným prvkom spájaného zoznamu:



Obrázok 11

Tu sa nám takisto môžu vyskytnúť dve situácie. Ak sa sú bloky vedľa seba, spojíme ich opäť funkciou merge:

```
//block_to_free is after linked list
if(temp->next == NULL && block_to_free > temp) {
    //found left neighbor of block_to_free, merge them
    if((void*)temp + sizeof(BL_HEAD) + temp->size == block_to_free) {
        merge_block(block_to_free, temp);
        printf("Block %p with size %d was released\n", valid_ptr, abs(size_to_free));
    }
```

Obrázok 12

Ak nie sú vedľa seba, tak blok len pridáme na koniec zoznamu:

```
//if they aren't neighbors
else {
    temp->next = block_to_free;
    block_to_free->prev = temp;
    block_to_free->next = NULL;
    printf("Block %p with size %d was released\n", valid_ptr, abs(size_to_free));
}
```

Obrázok 13

d) Chceme uvoľniť blok, ktorý je medzi dvomi blokmi spájaného zoznamu:

```
//if block_to_free is before some free block & after another free
else if(block_to_free < temp) {
    int merged = 0, freeSpace = block_to_free->size;
    temp = temp->prev;
    //find if there is left neighbor of block_to_free and then merge them
    if((void*)temp + sizeof(BL_HEAD) + temp->size == block_to_free) {
        merge_block(block_to_free, temp);
        printf("Block %p with size %d was released\n", valid_ptr, abs(size_to_free));
        merged++;
    }
    //find if there is right neighbor of block_to_free
    temp = temp->next;
    if((void*)temp - freeSpace - sizeof(BL_HEAD) == block_to_free) {
        //if block_to_free was already merged with left neighbor
        if(merged) {
            merge_block(temp->prev, temp);
        }
        //if block_to_free wasn't merged with left neighbor
        else {
            merge_block(block_to_free, temp);
            printf("Block %p with size %d was released\n", block_to_free, abs(size_to_free));
        }
    }
}
```

Obrázok 14

Vyhodnotenie testovania

1. Testovanie rovnakých blokov

```
int test_size = 118, block_size = 15;
int i, n_of_allocated = 0, n_of_deallocated = 0;
char region[test_size];
memory_init(region, test_size);
show_Ll();

char* array[test_size];
for(i = 0; (test_size - i*block_size) > block_size ; i++) {
    array[i] = memory_alloc(block_size);
    if(array[i]) {
        memset(array[i], 0, block_size);
        n_of_allocated++;
    }
}

double success_rate = ((double)n_of_allocated/(double)i) * 100.0;
printf("\nNumber of allocated %d, ideal case: %d, success rate %.2lf%%\n",
n_of_allocated, i, success_rate);

for(i = n_of_allocated-1; i >= 0; i--) {
    if(array[i])
        if(memory_free(array[i]))
           n_of_deallocated++;
}
if(n_of_allocated == n_of_deallocated)
    printf("Deallocation was successful\n");

return 0;
```

Výsledok:
Number of allocated 4, ideal case: 7, success rate 57.14%
Deallocation was succesful

Odhad zložitosti

Časová zložitosť

Pri vybranej metóde hľadania voľných blokov – first fit – časová zložitosť závisí od veľkostných rozpoložení voľných blokov, v priemere to býva konštantná časová zložitosť, avšak môže sa stať najhorší scenár, že blok s vyhovujúcou veľkosťou bude až na konci, vtedy by bola zložitosť N – pričom N je množstvo blokov v pamäti.

Priemerná zložitosť O(1)
Najhoršia zložitosť O(n)

Pamäťová zložitosť

Pri zvolenom algoritme sú využívané štruktúry, ktoré obsahujú smerníky, čiže na úrovni réžie pamäte vzniká fragmentácia, s ktorou je nutné počítať, čiže oproti ideálnemu riešeniu nealokujeme také množstvo pamäte. Ideálnejšie by bolo používať vhodný dátový typ, v ktorom by bolo číslo predstavujúce offsety namiesto smerníkov.