Національний Технічний Університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет Інформатики та Обчислювальної Техніки

Кафедра АСОІУ

**Лабораторна робота №2**

Виконав:

студент 3 курсу ФІОТ

групи ІС-72

Корнієнко В.

Перевірив:

Дифучин А.

Київ 2020

**Мета роботи**

Аллокатор пам'яті загального призначення повинен виконувати, принаймні, три завдання: виділення блоку пам'яті заданого розміру, звільнення виділеного блоку пам'яті і зміна розміру виділеного блоку пам'яті. Ці функції складають інтерфейс аллокатора пам'яті загального призначення:

* void \*mem\_alloc(size\_t size);
* void \*mem\_realloc(void \*addr, size\_t size);
* void mem\_free(void \*addr);
* mem\_dump()

**Оцінка витрат для збереження службових даних**

Char state – 1 byte;

size\_t size – 4 byte;

size\_t address – 4 byte;

**Оцінка часу**

O(N) - Оцінка часу пошуку вільної комірки пам’яті і видалення комірки.

N – кількість сторінок.

**Лістинг програми**

**allocator.h**

#ifndef ALLOCATOR

# define ALLOCATOR

#include <iostream>

#include <cstring>

#define MEMORY\_SIZE 1340

#define PAGE\_SIZE 256

#define FREE 0

#define DIVIDE 1

#define BLOCK 2

#define MULTBLOCK 3

typedef struct s\_page{

void\* address;

char state;

size\_t size;

} mem\_page;

void init\_memory(size\_t size);

void\* mem\_alloc(size\_t size);

void\* mem\_realloc(void\* address, size\_t size);

void mem\_free(void\* address);

void mem\_dump();

extern mem\_page\* global\_memory;

extern size\_t page\_count;

#endif

**main.cpp**

#include "allocator.h"

int main()

{

void\* data;

void\* data1;

void\* data2;

void\* data3;

init\_memory(MEMORY\_SIZE);

data = mem\_alloc(100);

memmove(data, "Hello World", 11);

data1 = mem\_alloc(200);

memmove(data1, "My Lab", 6);

data2 = mem\_alloc(100);

memmove(data2, "cmp", 3);

data3 = mem\_alloc(500);

memmove(data3, "aaaaaaaaaaaaaaaaaaaaaaaaaaaaa\

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa", 500);

mem\_dump();

mem\_realloc(data2,100);

mem\_free(data);

mem\_dump();

return 0;

}

**allocator.cpp**

#include "allocator.h"

mem\_page\* global\_memory;

size\_t page\_count;

void mem\_dump()

{

printf("------------Memory Dump------------\n");

printf("Global Memory Address: %p\n", global\_memory);

for (size\_t i = 0; i < page\_count; i++)

{

printf("-----------------------------\n");

printf("Page: %i State: %3i Address: %p\n", i, global\_memory[i].state, global\_memory[i].address);

printf("Data: %s\n", (char\*)global\_memory[i].address);

if (global\_memory[i].state == BLOCK || global\_memory[i].state == DIVIDE)

printf("Data: %s\n", (char\*)((size\_t)global\_memory[i].address + PAGE\_SIZE / 2));

printf("-----------------------------\n");

}

}

void init\_memory(size\_t size)

{

global\_memory = (mem\_page\*)malloc(size);

global\_memory = (mem\_page\*)calloc(MEMORY\_SIZE, 1);

page\_count = size / PAGE\_SIZE;

for (size\_t i = 0; i < page\_count; i++)

{

global\_memory[i].address = (void\*)((unsigned long int)global\_memory + page\_count + i \* PAGE\_SIZE);

global\_memory[i].size = 0;

global\_memory[i].state = FREE;

}

}

void\* mem\_alloc(size\_t size)

{

void\* address;

size\_t counter = 0;

size\_t firstpage = 0;

size\_t number\_pages\_to\_allocate = (double)size / PAGE\_SIZE + 0.5;

size\_t i;

if (size <= PAGE\_SIZE)

{

for (i = 0; i < page\_count; i++)

{

if (global\_memory[i].state == FREE || (global\_memory[i].state == DIVIDE && size <= PAGE\_SIZE / 2))

{

address = (void\*)((size\_t)(global\_memory[i].address) + (global\_memory[i].state == DIVIDE ? PAGE\_SIZE / 2 : 0));

if (size > PAGE\_SIZE / 2)

{

global\_memory[i].state = BLOCK;

global\_memory[i].size = 1;

}

else

{

global\_memory[i].state = global\_memory[i].state == DIVIDE ? BLOCK : DIVIDE;

global\_memory[i].size = 0;

}

return address;

}

}

}

else

{

for (i = 0; i < page\_count; i++)

{

if (global\_memory[i].state == FREE)

{

counter += 1;

firstpage = i - counter + 1;

if (counter == number\_pages\_to\_allocate)

{

for (size\_t j = firstpage; j < firstpage + counter; j++)

{

global\_memory[j].state = MULTBLOCK;

}

global\_memory[firstpage].size = number\_pages\_to\_allocate \* PAGE\_SIZE;

address = global\_memory[firstpage].address;

return address;

}

}

else

{

counter = 0;

firstpage = 0;

}

}

}

return NULL;

}

void\* mem\_realloc(void\* address, size\_t size)

{

void\* new\_address;

size\_t size\_of\_block = (size\_t)address - (size\_t)global\_memory - page\_count;

size\_t started\_block\_page = size\_of\_block / PAGE\_SIZE;

if (global\_memory[started\_block\_page].state == MULTBLOCK)

size\_of\_block = global\_memory[started\_block\_page].size;

else if (global\_memory[started\_block\_page].size != 0)

size\_of\_block = global\_memory[started\_block\_page].size \* PAGE\_SIZE;

else

size\_of\_block = PAGE\_SIZE / 2;

new\_address = mem\_alloc(size);

if (new\_address)

{

memmove(new\_address, address, size\_of\_block > size ? size : size\_of\_block);

mem\_free(address);

return new\_address;

}

return NULL;

}

void mem\_free(void\* address)

{

size\_t size\_of\_block = (size\_t)address - (size\_t)global\_memory - PAGE\_INFO \* page\_count;

size\_t started\_block\_page = size\_of\_block / PAGE\_SIZE;

size\_t page\_state = global\_memory[started\_block\_page].state;

if (page\_state == DIVIDE)

global\_memory[started\_block\_page].state = FREE;

else if (page\_state == BLOCK)

{

if (global\_memory[started\_block\_page].size == 0)

global\_memory[started\_block\_page].state = DIVIDE;

else

global\_memory[started\_block\_page].state = FREE;

}

else if (page\_state == MULTBLOCK)

for (size\_t i = started\_block\_page; i <global\_memory[started\_block\_page].size / PAGE\_SIZE; i++)

global\_memory[i].state = FREE;

std::memset(address, 0, global\_memory[started\_block\_page].size == 0 ? PAGE\_SIZE / 2: global\_memory[started\_block\_page].size);

global\_memory[started\_block\_page].size = 0;

address = NULL;

}