Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут”

Кафедра АСОІУ

**ЗВІТ**

про виконання комп’ютерного практикуму № 1

з дисципліни

“Операційні системи”

|  |  |  |
| --- | --- | --- |
| Прийняв: |  | Виконав: |
| Проф. Сімоненко В. П. |  | студент 3-го курсу  гр. ІП-51 ФІОТ  Зарічковий Олександр Анатолійович |

Київ – 2017

**ЗМІСТ:**

[1 ОПИС АЛГОРИТМУ 3](#_Toc493839366)

[2 Оцінка складності роботи 5](#_Toc493839367)

[3 Оцінка розходів на службову інформацію 6](#_Toc493839368)

[4 Переваги та не доліки розробленого алгоритму 7](#_Toc493839369)

[5 Лістинг програми 8](#_Toc493839370)

[6 Приклад виконання програми 15](#_Toc493839371)

# ОПИС АЛГОРИТМУ

## **Алокація пам’яті**

1. Початок
2. Додати до запрошеного розміру пам’яті розмір дескриптора пам’яті.
3. Перевірити на наявність необхідного об’єму вільної пам’яті та повернути покажчик на NULL, якщо відсутня вільна пам'ять необхідного об’єму.
4. Обрати серед всіх вільних областей пам’яті в загальному пулі, таку що більша або рівна запрошеному розміру і при цьому має мінімально можливий розмір.
5. Створити дескриптор та зареєструвати обрану область, як зайняту.
6. Кінець

## **Деалокація памяті**

1. Початок
2. Перевірити чи заданий покажчик не NULL.
3. Перевірити чи заданий покажчик вказує на пам'ять з загального пула та чи зареєстрована дана пам'ять, як зайнята.
4. Видалити дескриптор та зареєструвати область пам’яті, як вільну.
5. Якщо наступна область після даної вільна, то злити ці дві області в одну.
6. Якщо попередня область перед даною вільна, то злити ці дві області в одну.
7. КІНЕЦЬ

## **Реалокація памяті**

1. ПОЧАТОК
2. Перевірити чи заданий покажчик не NULL.
3. Перевірити чи заданий покажчик вказує на пам'ять з загального пула та чи зареєстрована дана пам'ять, як зайнята.
4. Алокувати нову область пам’яті.
5. Скопіювати в нову область необхідну кількість байт.
6. Деалокувати стару область пам’яті.
7. КІНЕЦЬ

# Оцінка складності роботи

## **Алокація**

Оскільки для зберігання розмірів областей памяті та знаходження вільних областей памяті використовуються червоно-чорні дерева, то

* Перевірка на наявність вільної пам'ять має складність O(log(n))
* Обрання найкращої вільної області – O(log(n))
* Створення дескриптора та реєстація області – O(log(n))

Загальна складність: O(log(n))

1. **Деалокація**

Деалокація – це зворотній процес до алокації, але він залежить від порядку деалокації, тому

* Видалення дескриптора – O(log(n)) (в середньому) та O(n) (в найгіршому випадку)
* Злиття двох сусідніх вільних областей – O(log(n)) (в середньому) та O(n) (в найгіршому випадку)

Загальна складність: – O(log(n)) (в середньому) та O(n) (в найгіршому випадку)

1. **Реалокація**

Оскільки реалокація – це алокація нової та де алокація старої памяті то загальна вартість операції: O(log(n)) (в середньому) та O(n) (в найгіршому випадку).

# Оцінка розходів на службову інформацію

Службова інформація – дескриптори областей, карта розмірів областей та карта вільних областей.

1. Дескриптор – структура з 2 покажчиками та 1 булевим флагом. Структура вирівняна по 4 байти, тому в памяті вона займає 12 байт для 1 області.
2. Карта розмірів – карта з ключом у вигляді покажчика на область та значенням у вигляді цілого числа, тому загальна вартість складає 8 байт для 1 області.
3. Карта вільних областей – карта з ключом у розміру областей та вигляді та значенням у вигляді списку адресів областей, тому загальна вартість складає ~8 байт для 1 вільної області.

Отже, загальна службова інформація займає ~28 байт на 1 область.

# Переваги та не доліки розробленого алгоритму

Розроблений алгоритм дозволяє дуже швидко алакувати та деалакувати необхідні області пам’ятай при цьому досягаюся мінімального рівня фрагментація пам’яті та максимального раціонально використовує загальний пул пам’яті. До недоліків можна віднести залежність від порядку алокації та деалокації областей, які можуть призвести до деградації до лінійної складності операції деалокації.

# Лістинг програми

“MemoryHeader.h”

#pragma once

#define ALIGIN\_TYPE long int // 4 bytes

union MemoryHeader {

struct Header{

MemoryHeader\* NextBlock;

MemoryHeader\* PrevBlock;

bool isFree;

} sheader;

ALIGIN\_TYPE aligin[sizeof(Header) / sizeof(ALIGIN\_TYPE) + (sizeof(Header) % sizeof(ALIGIN\_TYPE) ? 1 : 0)];

};

“MemoryAllocator.h”

#pragma once

#include <map>

#include <vector>

#include <cstdio>

#include <algorithm>

#include "MemoryHeader.h"

using std::map;

using std::vector;

class MemoryAllocator {

private:

size\_t \_allocated\_memory;

char\* \_memory\_pool;

map <void\*, size\_t> \_sizes\_map;

map <size\_t, vector<char \*>> \_free\_map;

void \_merge(MemoryHeader\* first, MemoryHeader\* second);

public:

MemoryAllocator(size\_t size);

void\* allocate(size\_t size);

void deallocate(void\* pointer);

void\* reallocate(void\* pointer, size\_t size);

void mem\_dump(void);

};

“MemoryAllocator.cpp”

#include "MemoryAllocator.h"

//#define DEBUG

#ifdef DEBUG

#include <iostream>

using std::cout;

#endif // DEBUG

MemoryAllocator::MemoryAllocator(size\_t size) {

if (size < sizeof(MemoryHeader)) {

throw std::bad\_alloc();

}

\_memory\_pool = new char[size];

\_allocated\_memory = 0;

\_free\_map[size].push\_back((char \*)\_memory\_pool);

\_sizes\_map[\_memory\_pool] = size;

MemoryHeader\* loc\_header = new (\_memory\_pool) MemoryHeader();

loc\_header->sheader.NextBlock = nullptr;

loc\_header->sheader.PrevBlock = nullptr;

loc\_header->sheader.isFree = true;

#ifdef DEBUG

printf("Created pool for %d bytes\n", size);

#endif // DEBUG

}

void\* MemoryAllocator::allocate(size\_t size) {

size = (size / sizeof(ALIGIN\_TYPE) + (size % sizeof(ALIGIN\_TYPE) ? 1 : 0)) \* sizeof(ALIGIN\_TYPE);

size += sizeof(MemoryHeader);

auto iter = \_free\_map.lower\_bound(size);

if (iter == \_free\_map.end()) {

return (void \*) nullptr;

}

char\* location = (iter->second).back();

(iter->second).pop\_back();

MemoryHeader\* prev = ((MemoryHeader\*)(location))->sheader.PrevBlock;

MemoryHeader\* next = ((MemoryHeader\*)(location))->sheader.NextBlock;

if ((iter->second).size() == 0) {

\_free\_map.erase(iter);

}

auto size\_iter = \_sizes\_map.find(location);

size\_t loc\_size = size\_iter->second;

\_sizes\_map.erase(size\_iter);

size\_t left\_mem\_size = loc\_size - size;

MemoryHeader\* loc\_header = new (location) MemoryHeader();

loc\_header->sheader.PrevBlock = prev;

loc\_header->sheader.NextBlock = next;

loc\_header->sheader.isFree = false;

if (left\_mem\_size >= sizeof(MemoryHeader) + sizeof(ALIGIN\_TYPE)) {

char\* next\_block = location + size;

MemoryHeader\* next\_header = new (next\_block) MemoryHeader();

next\_header->sheader.PrevBlock = ((MemoryHeader\*)(location));

next\_header->sheader.NextBlock = ((MemoryHeader\*)(next));

next\_header->sheader.isFree = true;

loc\_header->sheader.NextBlock = next\_header;

\_sizes\_map[next\_block] = left\_mem\_size;

\_free\_map[left\_mem\_size].push\_back(next\_block);

}

else {

size += left\_mem\_size;

}

\_sizes\_map[location] = size;

\_allocated\_memory += size;

#ifdef DEBUG

printf("Allocated %d in %p\n", size, location);

std::cout << "Sizes map:\n";

for (auto const &iter : \_sizes\_map) {

printf("%p - %d\n", iter.first, iter.second);

}

std::cout << "Free map:\n";

for (auto const &iter : \_free\_map) {

cout << iter.first << " :";

for (auto const &v\_iter : iter.second) {

printf(" %p", v\_iter);

}

printf("\n");

}

system("pause");

#endif // !DEBUG

return (void \*)(location + sizeof(MemoryHeader));

}

void MemoryAllocator::\_merge(MemoryHeader\* first, MemoryHeader\* second) {

auto iter\_size = \_sizes\_map.find(second);

for (auto iter\_free = \_free\_map[iter\_size->second].begin();

iter\_free < \_free\_map[iter\_size->second].end();

iter\_free++) {

if ((char \*)second == \*iter\_free) {

\_free\_map[iter\_size->second].erase(iter\_free);

break;

}

}

if (\_free\_map[iter\_size->second].size() == 0) {

\_free\_map.erase(iter\_size->second);

}

size\_t second\_size = iter\_size->second;

\_sizes\_map.erase(iter\_size);

iter\_size = \_sizes\_map.find(first);

for (auto iter\_free = \_free\_map[iter\_size->second].begin();

iter\_free < \_free\_map[iter\_size->second].end();

iter\_free++) {

if ((char \*)first == \*iter\_free) {

\_free\_map[iter\_size->second].erase(iter\_free);

break;

}

}

if (\_free\_map[iter\_size->second].size() == 0) {

\_free\_map.erase(iter\_size->second);

}

\_sizes\_map[first] += second\_size;

\_free\_map[\_sizes\_map[first]].push\_back((char \*)first);

first->sheader.NextBlock = second->sheader.NextBlock;

if (first->sheader.NextBlock) {

first->sheader.NextBlock->sheader.PrevBlock = first;

}

return;

}

void MemoryAllocator::deallocate(void\* pointer) {

char\* loc\_pointer = (char \*)pointer;

size\_t loc\_size;

loc\_pointer -= sizeof(MemoryHeader);

if (\_sizes\_map.find(loc\_pointer) == \_sizes\_map.end()) {

return;

}

auto iter = \_sizes\_map.find(loc\_pointer);

loc\_size = iter->second;

if (((MemoryHeader \*)(iter->first))->sheader.isFree) {

return;

}

MemoryHeader\* loc\_header = (MemoryHeader\*)loc\_pointer;

loc\_header->sheader.isFree = true;

\_free\_map[iter->second].push\_back((char \*)iter->first);

if (loc\_header->sheader.NextBlock && loc\_header->sheader.NextBlock->sheader.isFree) {

\_merge(loc\_header, loc\_header->sheader.NextBlock);

}

if (loc\_header->sheader.PrevBlock && loc\_header->sheader.PrevBlock->sheader.isFree) {

\_merge(loc\_header->sheader.PrevBlock, loc\_header);

}

#ifdef DEBUG

printf("Deallocated %d in %p\n", loc\_size, loc\_pointer);

std::cout << "Sizes map:\n";

for (auto const &iter : \_sizes\_map) {

printf("%p - %d\n", iter.first, iter.second);

}

std::cout << "Free map:\n";

for (auto const &iter : \_free\_map) {

cout << iter.first << " :";

for (auto const &v\_iter : iter.second) {

printf(" %p", v\_iter);

}

printf("\n");

}

system("pause");

#endif // !DEBUG

}

void\* MemoryAllocator::reallocate(void\* pointer, size\_t size) {

char\* loc\_pointer = (char \*)pointer - sizeof(MemoryHeader);

if (\_sizes\_map.find(loc\_pointer) == \_sizes\_map.end()) {

return nullptr;

}

void\* new\_memory = allocate(size);

memcpy(new\_memory, pointer, std::min(size, \_sizes\_map[loc\_pointer]));

deallocate(pointer);

}

void MemoryAllocator::mem\_dump(void) {

printf("Memory dump\nGlobal pool %p\n", \_memory\_pool);

for (auto const &iter : \_sizes\_map) {

printf("Location: %p - Size: %09d bytes - IsFree: %1d\n", iter.first, iter.second, ((MemoryHeader\*)iter.first)->sheader.isFree);

}

}

“main.cpp”

#include "MemoryAllocator.h"

#define SYSTEM\_POLL\_SIZE\_BYTES 1048576 // 2^20 = 1 Mb

MemoryAllocator GLOBAL\_MEMORY\_ALLOCATOR = MemoryAllocator(SYSTEM\_POLL\_SIZE\_BYTES);

//-------------------------------------------------

// Labwork functionality

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

return GLOBAL\_MEMORY\_ALLOCATOR.allocate(size);

}

void \*mem\_realloc(void \*addr, size\_t size) {

return GLOBAL\_MEMORY\_ALLOCATOR.reallocate(addr, size);

}

void mem\_free(void \*addr) {

return GLOBAL\_MEMORY\_ALLOCATOR.deallocate(addr);

}

//---------------------------------------------------

// Tests coverage

// Warning: for more debug information enable DEBUG flag in MemoryAllocator.cpp

bool test\_alloc() {

void\* pool = nullptr;

pool = mem\_alloc(SYSTEM\_POLL\_SIZE\_BYTES + 1);

if (pool != nullptr) {

return false;

}

pool = mem\_alloc(sizeof(ALIGIN\_TYPE));

ALIGIN\_TYPE\* value = new (pool) (ALIGIN\_TYPE)(1);

if (\*value != (ALIGIN\_TYPE)(1)) {

return false;

}

pool = mem\_alloc(sizeof(short) \* 100);

short\* arr = new (pool) short[100];

for (int i = 0; i < 100; i++) {

arr[i] = (short) i;

}

return true;

}

bool test\_dealloc() {

void \* pool = nullptr;

mem\_free(pool);

pool = mem\_alloc(sizeof(short) \* 100);

short\* arr = new (pool) short[100];

for (int i = 0; i < 100; i++) {

arr[i] = (short)i;

}

mem\_free(pool);

return true;

}

bool test\_performance() {

void\* arr[10];

for (int i = 0; i < 10; i++) {

if (i % 2) {

arr[i] = mem\_alloc(sizeof(short) \* 10);

} else {

arr[i] = mem\_alloc(sizeof(short) \* 20);

}

}

for (int i = 0; i < 10; i++) {

if (i % 2) {

mem\_free(arr[i]);

}

}

mem\_free(arr[0]);

void\* pool = mem\_alloc(sizeof(short) \* 10);

mem\_free(pool);

pool = mem\_alloc(sizeof(short) \* 20);

mem\_free(pool);

for (int i = 0; i < 10; i++) {

mem\_free(arr[i]);

}

return true;

}

//---------------------------------------------------

int main(void) {

printf("Start testing....\n");

printf("Allocation test.... %s\n", (test\_alloc() ? "Ok" : "Failed"));

printf("Deallocation test.... %s\n", (test\_dealloc() ? "Ok" : "Failed"));

printf("Complex performance test.... %s\n", (test\_performance() ? "Ok" : "Failed"));

printf("Dumping info...\n");

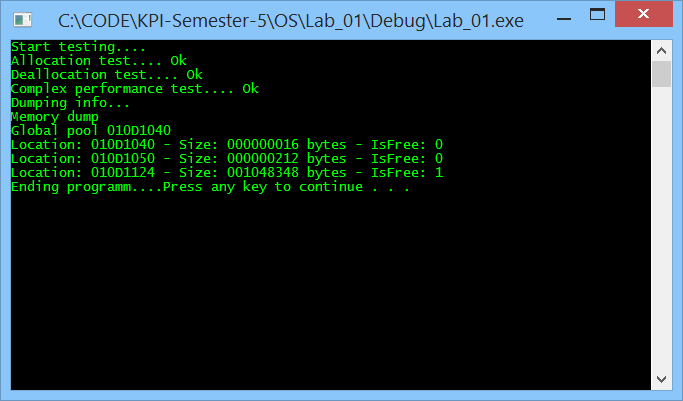
GLOBAL\_MEMORY\_ALLOCATOR.mem\_dump();

printf("Ending programm....");

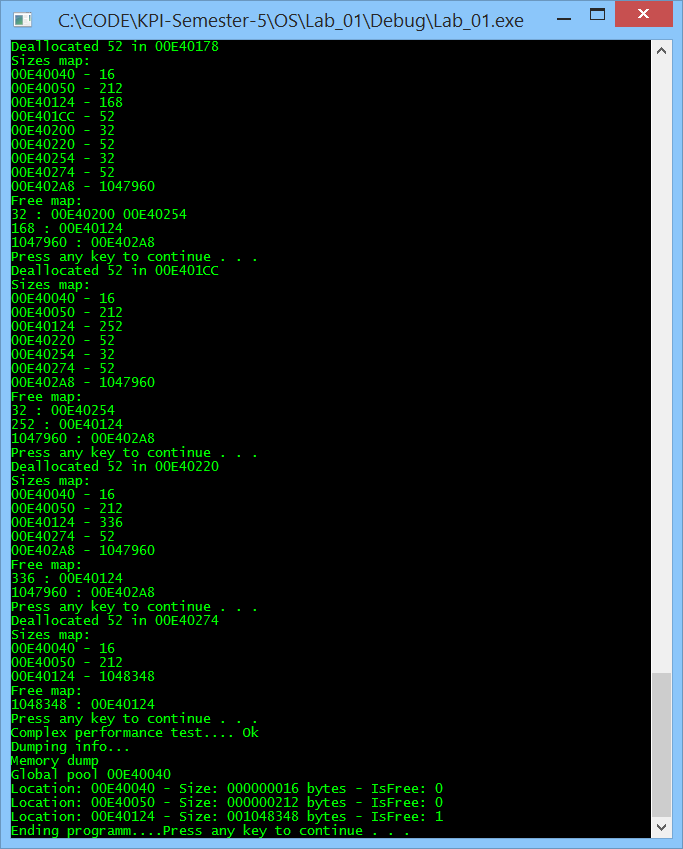
system("pause");

}

# Приклад виконання програми



Робота програми без DEBUG mode



Робота програми з DEBUG mode