НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування та спеціалізованих комп‘ютерних систем

**Лабораторна робота №2**

***З дисципліни: «Системне програмне забезпечення»***

«Аллокатор пам’яті загального призначення (2 частина)»

Студент КВ-42**:** Савицький Т.П.

Перевірив(ла):

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Київ-2017

**Завдання**

1. Області пам'яті можна виділяти будь-яким доступним способом.

2. Функції mem\_alloc (), mem\_realloc () і mem\_free () повинні відповідати наведеним вище прототипам.

3. Адреси пам'яті, які повертаються функціями mem\_alloc () і mem\_realloc (), повинні бути вирівняні на кордон в 4 байта.

4. Спробувати зменшити час пошуку вільного блоку пам'яті і час звільнення зайнятого блоку.

5. Спробувати зменшити фрагментацію пам'яті.

6. Написати функцію mem\_dump (), яка повинна виводити на консоль з-стояння областей пам'яті.

Всі блоки пам'яті діляться на дві групи. До першої групи входять блоки з роз-рами менше або рівними половині сторінки, у другу всі інші. Блоки першої групи діляться на класи. Блоки одного класу мають однаковий раз-мер. Наприклад, цей розмір може бути число 2x (x ≥ 4). Якщо додаток запрошує блок пам'яті деякого розміру меншого або рівного половині сторінки, то аллокатор призначає цього блоку найближчий за розміром клас. Блоки другої групи це блоки розміром в одну або кілька сторінок. Якщо додатку-ня запитує блок пам'яті деякого розміру більшого ніж половина сторінки, то аллокатор округлює цей розмір до найближчого цілого числа сторінок.

Оцінимо цей алгоритм. Якщо прийняти, що всі блоки вирівняні на кордон 4 байтів (тут 1 байт це 1 октет), то розмір заголовка на 32 бітної системі може бути дорівнює 8 байтам. Один біт заголовка показується зайнятість блоку, інші біти показують розмір поточного і наступного блоків в байтах. Для N блоків потрібно (N \* 8) байт додаткової інформації. Також буде потрібно пам'ять для зберігання статистики і покажчиків на області пам'яті. У гіршому випадку при пошуку вільного блоку буде потрібно перевірити N заголовків. Звільнення блоку відбувається швидко, для цього достатньо перевірити заголовки лівого і правого сусідніх блоків. Якщо виконати 1000 пар запитів на виділення 10 байт і 20 байт, а потім звільнити все блоки по 10 байт, то сумарна вільна пам'ять матиме розмір ≥18000 байт, але для виділення нового блоку в 20 байт можливо буде потрібно нова область пам'яті.

**Текст програми**

#include <iostream>

#include "stdio.h"

#include "pagememoryallocator.h"

using namespace std;

int main() {

printf("Work Started!\n");

PageMemoryAllocator al = PageMemoryAllocator();

cout << al.mem\_dump() << "\n\n";

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

void\* add1 = al.mem\_alloc(1500);

void\* add2 = al.mem\_alloc(1000);

void\* add3 = al.mem\_alloc(500);

printf("After first allocation.\n");

cout << al.mem\_dump() << "\n\n";

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

al.mem\_free(add2);

printf("After free.\n");

cout << al.mem\_dump() << "\n\n";

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

al.mem\_realloc(add1, 500);

printf("After first reallocation.\n");

cout << al.mem\_dump() << "\n\n";

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

printf("After second reallocation.\n");

al.mem\_realloc(add3, 1000);

cout << al.mem\_dump() << "\n";

std::getchar();

return 0;

}

/\*Містить константи для початкової ініціалізації аллокатора згідно значень за замовченням.\*/

class Constants {

public:

/\*Розмір сторінки за замовченням.\*/

static const unsigned int DEFAULT\_PAGE\_SIZE = 4096;

/\*Об'єм буферної пам'яті за замовченням.\*/

static const unsigned int DEFAULT\_MEMORY\_SIZE = 100 \* DEFAULT\_PAGE\_SIZE;

};

/\*Блок пам'яті.\*/

class MemoryBlock {

public:

MemoryBlock();

};

#include "memoryblock.h"

MemoryBlock::MemoryBlock() {}

#include "stdio.h"

#include <algorithm>

#include<vector>

using namespace std;

/\*Сторінка пам'яті.\*/

class MemoryPage {

public:

/\*Конструктор за замовченням.\*/

MemoryPage();

/\*Конструктор з параметрами.\*/

MemoryPage(void\* startAddress);

/\*Спроба розбити сторінку на блоки однакового розміру.\*/

void SplitPage(unsigned int blockSize, unsigned int countOfBlocks);

/\*

\* Спроба виділити перший вільний блок.

\* В разі успіху повертає адресу початку цього блоку.

\* В іншому випадку повертає NULL.

\* \*/

void\* AllocateFirstFreeBlock();

/\*

\* Спроба звільнити блок із заданою адресою.

\* \*/

void FreeBlock(void\* address);

/\*

\* Перевірка на заповненість сторінки.

\* \*/

bool IsFullFilled();

/\*

\* Перевірка порожність сторінки.

\* \*/

bool IsEmpty();

void \*GetMemory() const;

private:

/\*Пам'ять сторінки.\*/

void\* memory;

/\*Вільні блоки.\*/

vector<void\*> freeBlockAddresses;

/\*Зайняті блоки.\*/

vector<void\*> blocksInUseAddresses;

};

#include "stdio.h"

#include <algorithm>

#include<vector>

using namespace std;

/\*Сторінка пам'яті.\*/

class MemoryPage {

public:

/\*Конструктор за замовченням.\*/

MemoryPage();

/\*Конструктор з параметрами.\*/

MemoryPage(void\* startAddress);

/\*Спроба розбити сторінку на блоки однакового розміру.\*/

void SplitPage(unsigned int blockSize, unsigned int countOfBlocks);

/\*

\* Спроба виділити перший вільний блок.

\* В разі успіху повертає адресу початку цього блоку.

\* В іншому випадку повертає NULL.

\* \*/

void\* AllocateFirstFreeBlock();

/\*

\* Спроба звільнити блок із заданою адресою.

\* \*/

void FreeBlock(void\* address);

/\*

\* Перевірка на заповненість сторінки.

\* \*/

bool IsFullFilled();

/\*

\* Перевірка порожність сторінки.

\* \*/

bool IsEmpty();

void \*GetMemory() const;

private:

/\*Пам'ять сторінки.\*/

void\* memory;

/\*Вільні блоки.\*/

vector<void\*> freeBlockAddresses;

/\*Зайняті блоки.\*/

vector<void\*> blocksInUseAddresses;

};

#include "memorypage.h"

MemoryPage::MemoryPage() {

memory = NULL;

freeBlockAddresses = vector<void\*>();

blocksInUseAddresses = vector<void\*>();

}

/\*Конструктор з параметрами.\*/

MemoryPage::MemoryPage(void \*startAddress) {

memory = startAddress;

freeBlockAddresses = vector<void\*>();

blocksInUseAddresses = vector<void\*>();

}

/\*Спроба розбити сторінку на блоки однакового розміру.\*/

void MemoryPage::SplitPage(unsigned int blockSize, unsigned int countOfBlocks) {

freeBlockAddresses.clear();

blocksInUseAddresses.clear();

for(unsigned int i = 0; i < countOfBlocks; i++){

freeBlockAddresses.push\_back((void \*)((unsigned int)memory + blockSize \* i));

}

}

/\*

\* Спроба виділити перший вільний блок.

\* В разі успіху повертає адресу початку цього блоку.

\* В іншому випадку повертає NULL.

\* \*/

void \*MemoryPage::AllocateFirstFreeBlock() {

if(freeBlockAddresses.empty()){

return NULL;

}else{

void\* ret = freeBlockAddresses[0];

freeBlockAddresses.erase(freeBlockAddresses.begin());

blocksInUseAddresses.push\_back(ret);

return ret;

}

}

/\*

\* Спроба звільнити блок із заданою адресою.

\* \*/

void MemoryPage::FreeBlock(void \*address) {

vector<void\*>::iterator it = find(blocksInUseAddresses.begin(), blocksInUseAddresses.end(), address);

if(blocksInUseAddresses.end() == it){

return;

}

freeBlockAddresses.push\_back(\*it);

sort(freeBlockAddresses.begin(), freeBlockAddresses.end());

blocksInUseAddresses.erase(it);

}

/\*

\* Перевірка на заповненість сторінки.

\* \*/

bool MemoryPage::IsFullFilled() {

return freeBlockAddresses.empty();

}

/\*

\* Перевірка порожність сторінки.

\* \*/

bool MemoryPage::IsEmpty() {

return blocksInUseAddresses.empty();

}

void \*MemoryPage::GetMemory() const {

return memory;

}

/\*Звідси беремо константи для налаштувань за замовченням.\*/

#include "constants.h"

/\*Звідси беремо клас "MemoryPage".\*/

#include "memorypage.h"

/\*Звідси беремо команду "malloc".\*/

#include "malloc.h"

/\*Звідси беремо контейнер "vector".\*/

#include <vector>

#include <unordered\_map>

#include <string>

using namespace std;

/\*Аллокатор пам'яті, який діє за сторінковим принципом.\*/

class PageMemoryAllocator {

public:

/\*Конструктор за замовченням.\*/

PageMemoryAllocator();

/\*Конструктор з параметрами.\*/

PageMemoryAllocator(unsigned int newMemorySize, unsigned int newPageSize);

/\*

\* Спроба виділити пам'ять заданого розміру.

\* При успіху повертає адресу початку виділеної області пам'яті.

\* Інакше повертає NULL.

\* \*/

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

/\*Спроба звільнити блок пам'яті із заданою адресою початку.\*/

void mem\_free(void\* addr);

/\*Спроба змінити розмір блоку пам'яті із заданою адресою початку.\*/

void \*mem\_realloc(void\* addr, size\_t newSize);

/\*Повертає текстову інформацію про поточний стан буферної пам'яті.\*/

string mem\_dump();

private:

/\*Спроба розділити наявну пам'ять між сторінками.\*/

void ShareMemoryBetweenPages();

/\*Обчислення можливих розмірів блоків.\*/

void CalculatePossibleBlockSizes();

/\*Ініціалізація векторів номерів сторінок.\*/

void PageNumberVectorsInitializtion();

/\*Спроба визначити клас блоку, який потрібний для розміщення пам'яті заданого розміру.\*/

unsigned int FindBlockClassWithRequiredSize(size\_t size);

/\*

\* Спроба виділити блок пам'яті, що складається з визначеної кількості сторінок.

\* В разі успіху повертає true.

\* Інакше повертає false.

\* \*/

void\* AllocateBigMemoryBlock(unsigned int countOfPages);

/\*Визначає, чи потрапляє у буферну пам'ять задана адреса.\*/

bool IsInMemoryRange(void\* addr);

/\*Буферна пам'ять.\*/

void\* memory;

/\*Загальний розмір буферної пам'яті у байтах.\*/

unsigned int memorySize;

/\*Розмір однієї сторінки у байтах.\*/

unsigned int pageSize;

/\*Вектор описів сторінок пам'яті.\*/

vector<MemoryPage> memoryPages;

/\*Вектор номерів вільних сторінок.\*/

vector<unsigned int> freePageNumbers;

/\*Вектор номерів частково зайнятих сторінок, розбитих за класами.\*/

vector< vector<unsigned int> > inUsePageNumbers;

/\*Вектор номерів повністю зайнятих сторінок.\*/

vector<unsigned int> fullFilledPageNumbers;

/\*Вектор можливих розмірів блоку, який можна умістити на одну сторінку.\*/

vector<unsigned int> possibleBlockSizes;

/\*Хеш-таблиця блоків, що займають одну або більше сторінок пам'яті.\*/

unordered\_map<void\*, vector<unsigned int> > bigBlocks;

};

#include "pagememoryallocator.h"

/\*Конструктор за замовченням.\*/

PageMemoryAllocator::PageMemoryAllocator() {

pageSize = Constants::DEFAULT\_PAGE\_SIZE;

memorySize = Constants::DEFAULT\_MEMORY\_SIZE;

memory = (void\*)malloc(memorySize \* 4);

possibleBlockSizes = vector<unsigned int>();

CalculatePossibleBlockSizes();

PageNumberVectorsInitializtion();

ShareMemoryBetweenPages();

bigBlocks = unordered\_map<void\*, vector<unsigned int> >();

}

/\*Конструктор з параметрами.\*/

PageMemoryAllocator::PageMemoryAllocator(unsigned int newMemorySize, unsigned int newPageSize) {

pageSize = newPageSize;

memorySize = newMemorySize;

memory = (void\*)malloc(memorySize \* 4);

possibleBlockSizes = vector<unsigned int>();

CalculatePossibleBlockSizes();

PageNumberVectorsInitializtion();

ShareMemoryBetweenPages();

bigBlocks = unordered\_map<void\*, vector<unsigned int> >();

}

/\*

\* Спроба виділити пам'ять заданого розміру.

\* При успіху повертає адресу початку виділеної області пам'яті.

\* Інакше повертає NULL.

\* \*/

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

void\* ret = NULL;

/\*Якщо необхідний розмір блоку менший за половину сторінки, то:...\*/

if (size <= pageSize / 2) {

/\*1. З'ясовується клас цього блоку\*/

unsigned int requiredBlockClass = FindBlockClassWithRequiredSize(size);

/\*2. Виконується пошук сторінки, яка містить вільний блок необхідного розміру.\*/

/\*Якщо сторінок, поділених на блоки відповідного розміру немає, то:...\*/

if (inUsePageNumbers[requiredBlockClass].empty()) {

/\*1. Якщо вільних сторінок немає, повертається NULL.\*/

if (freePageNumbers.empty()) {

return ret;

}

/\*2. Інакше береться перша вільна сторінка, ділиться на блоки, перший з цих блоків повертається, а сама сторінка переходить в розряд тих, що використовуються.\*/

memoryPages[freePageNumbers[0]].SplitPage(possibleBlockSizes[requiredBlockClass], pageSize / possibleBlockSizes[requiredBlockClass]);

ret = memoryPages[freePageNumbers[0]].AllocateFirstFreeBlock();

inUsePageNumbers[requiredBlockClass].push\_back(freePageNumbers[0]);

freePageNumbers.erase(freePageNumbers.begin());

}else{

/\*Якщо необхідні сторінки є, то у першій з них виділяється перший доступний блок.\*/

ret = memoryPages[inUsePageNumbers[requiredBlockClass][0]].AllocateFirstFreeBlock();

/\*Якщо після цього сторінка виявиться заповненою, про це окремо вказується.\*/

if(memoryPages[inUsePageNumbers[requiredBlockClass][0]].IsFullFilled()){

fullFilledPageNumbers.push\_back(inUsePageNumbers[requiredBlockClass][0]);

inUsePageNumbers[requiredBlockClass].erase(inUsePageNumbers[requiredBlockClass].begin());

}

}

}else{

/\*Якщо необхідний блок, більший за половину сторінки, то:...\*/

unsigned int requiredPagesCount = size / pageSize + 1;

if(size % pageSize == 0){

requiredPagesCount--;

}

if(requiredPagesCount > freePageNumbers.size()){

return ret;

}

return AllocateBigMemoryBlock(requiredPagesCount);

}

return ret;

}

/\*Спроба звільнити блок пам'яті із заданою адресою початку.\*/

void PageMemoryAllocator::mem\_free(void \*addr) {

if(!IsInMemoryRange(addr)){

return;

}

/\*Чи не належить даний блок до класу великих блоків?\*/

unordered\_map<void\*, vector<unsigned int> >::const\_iterator gotInfo = bigBlocks.find(addr);

if(gotInfo == bigBlocks.end()){

/\*Визначення сторінки, до якої відноситься даний блок.\*/

unsigned int pageNumber = ((unsigned int)addr - (unsigned int)memory) / pageSize;

memoryPages[pageNumber].FreeBlock(addr);

if(memoryPages[pageNumber].IsEmpty()){

for(unsigned int i = 0; i < inUsePageNumbers.size(); i++){

vector<unsigned int>::const\_iterator it = find(inUsePageNumbers[i].begin(), inUsePageNumbers[i].end(), pageNumber);

if(it != inUsePageNumbers[i].end()){

freePageNumbers.push\_back(\*it);

sort(freePageNumbers.begin(), freePageNumbers.end());

inUsePageNumbers[i].erase(it);

break;

}

}

}

}else{

for(unsigned int i = 0; i < gotInfo->second.size(); i++){

freePageNumbers.push\_back(gotInfo->second[i]);

fullFilledPageNumbers.erase(find(fullFilledPageNumbers.begin(), fullFilledPageNumbers.end(), gotInfo->second[i]));

}

bigBlocks.erase(gotInfo);

sort(freePageNumbers.begin(), freePageNumbers.end());

}

}

/\*Спроба змінити розмір блоку пам'яті із заданою адресою початку.\*/

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

if(!IsInMemoryRange(addr)){

return NULL;

}

if(newSize <= 0){

mem\_free(addr);

return NULL;

}

/\*Чи не належить даний блок до класу великих блоків?\*/

unordered\_map<void\*, vector<unsigned int> >::const\_iterator gotInfo = bigBlocks.find(addr);

if(gotInfo == bigBlocks.end()){

/\*Визначення сторінки, до якої відноситься даний блок.\*/

unsigned int pageNumber = ((unsigned int)addr - (unsigned int)memory) / pageSize;

/\*Визначення класу блоку.\*/

for (unsigned int i = 0; i < inUsePageNumbers.size(); i++) {

vector<unsigned int>::const\_iterator it = find(inUsePageNumbers[i].begin(), inUsePageNumbers[i].end(), pageNumber);

if (it != inUsePageNumbers[i].end()) {

/\*Визначення номеру блоку, до якого відноситься адреса.\*/

unsigned int blockNumber = ((unsigned int)addr - (unsigned int)(memoryPages[pageNumber]).GetMemory()) / possibleBlockSizes[i];

if ((unsigned int)memoryPages[pageNumber].GetMemory() + blockNumber \* possibleBlockSizes[i] != (unsigned int)addr) {

return NULL;

}

void\* buf = (void\*)malloc(possibleBlockSizes[i]);

memcpy(buf, addr, possibleBlockSizes[i]);

mem\_free(addr);

void\* ret = mem\_alloc(newSize);

/\*Якщо місця для нового блоку немає, то потрібно повернути все назад.\*/

if(ret == NULL){

ret = mem\_alloc(possibleBlockSizes[i]);

memcpy(ret, addr, possibleBlockSizes[i]);

}else{

memcpy(ret, addr, newSize);

}

free(buf);

return ret;

}

}

return NULL;

}else{

/\*Чи ми зменшуємо або не змінюємо розмір блоку?\*/

unsigned int oldSize = gotInfo->second.size() \* pageSize;

if(oldSize >= newSize){

/\*Якщо ми зменшуємо великий блок, то потрібно звільнити визначену кількість сторінок в його кінці.\*/

unsigned int countOfPagesToFree;

if(newSize <= pageSize / 2){

countOfPagesToFree = gotInfo->second.size();

}else{

countOfPagesToFree = (oldSize - newSize) / pageSize;

}

vector<unsigned int> v = vector<unsigned int>(gotInfo->second);

for(unsigned int i = 0; i < countOfPagesToFree; i++){

freePageNumbers.push\_back(v.back());

fullFilledPageNumbers.erase(find(fullFilledPageNumbers.begin(), fullFilledPageNumbers.end(), v.back()));

v.pop\_back();

}

bigBlocks.erase(gotInfo);

bigBlocks.emplace(addr, v);

sort(freePageNumbers.begin(), freePageNumbers.end());

/\*Якщо блок після цього переходить в розряд маленьких, то запис про нього видаляється.\*/

if(gotInfo->second.empty()){

bigBlocks.erase(gotInfo);

void\* buf = (void\*)malloc(newSize);

memcpy(buf, addr, newSize);

void\* ret = mem\_alloc(newSize);

memcpy(ret, buf, newSize);

free(buf);

return ret;

}

/\*Якщо блок залишається великим, то нічого більше не змінюється.\*/

return addr;

}

/\*Якщо ми збільшуємо блок, то потрібно спочатку очистити поточний, запам'ятавши його вміст.\*/

void\* buf = (void\*)malloc(oldSize);

memcpy(buf, addr, oldSize);

mem\_free(addr);

void\* ret = mem\_alloc(newSize);

/\*Якщо місця для нового блоку немає, то потрібно повернути все назад.\*/

if(ret == NULL){

ret = mem\_alloc(oldSize);

memcpy(ret, addr, oldSize);

}else{

memcpy(ret, addr, newSize);

}

free(buf);

return ret;

}

return NULL;

}

string PageMemoryAllocator::mem\_dump() {

string ret = "";

ret.append("Memory total (bytes): ");

ret.append(to\_string(memorySize));

ret.append("\nPage size (bytes): ");

ret.append(to\_string(pageSize));

ret.append("\nCount of pages: ");

ret.append(to\_string(memoryPages.size()));

ret.append("\nOf them:");

ret.append("\n\tis free: ");

ret.append(to\_string(freePageNumbers.size()));

ret.append("\n\tnot completely full: ");

unsigned int inUseCount = 0;

for(unsigned int i = 0; i < inUsePageNumbers.size(); i++){

for(unsigned int j = 0; j < inUsePageNumbers[i].size(); j++){

inUseCount++;

}

}

ret.append(to\_string(inUseCount));

ret.append("\n\tFully filled: ");

ret.append(to\_string(fullFilledPageNumbers.size()));

return ret;

}

/\*Спроба розділити наявну пам'ять між сторінками.\*/

void PageMemoryAllocator::ShareMemoryBetweenPages() {

unsigned int pageCount = memorySize / pageSize;

for(unsigned int i = 0; i < pageCount; i++){

memoryPages.push\_back(MemoryPage((void \*)((unsigned int)memory + i \* pageSize)));

freePageNumbers.push\_back(i);

}

}

/\*Обчислення можливих розмірів блоків.\*/

void PageMemoryAllocator::CalculatePossibleBlockSizes() {

possibleBlockSizes.push\_back(1);

unsigned int base = 2;

do{

possibleBlockSizes.push\_back(base);

base = base << 1;

}while(base <= pageSize / 2);

}

/\*Ініціалізація векторів номерів сторінок.\*/

void PageMemoryAllocator::PageNumberVectorsInitializtion() {

freePageNumbers = vector<unsigned int>();

inUsePageNumbers = vector< vector<unsigned int> >();

for(unsigned int i = 0; i < possibleBlockSizes.size(); i++){

inUsePageNumbers.push\_back(vector<unsigned int>());

}

fullFilledPageNumbers = vector<unsigned int>();

}

/\*Спроба визначити клас блоку, який потрібний для розміщення пам'яті заданого розміру.\*/

unsigned int PageMemoryAllocator::FindBlockClassWithRequiredSize(size\_t size) {

for(unsigned int i = 0; i < possibleBlockSizes.size(); i++){

if(size <= possibleBlockSizes[i]){

return i;

}

}

return -1;

}

/\*

\* Спроба виділити блок пам'яті, що складається з визначеної кількості сторінок.

\* В разі успіху повертає true.

\* Інакше повертає false.

\* \*/

void\* PageMemoryAllocator::AllocateBigMemoryBlock(unsigned int countOfPages) {

void\* ret = NULL;

for(unsigned int i = 0; i <= freePageNumbers.size() - countOfPages;){

bool blockIsFound = true;

for(unsigned int j = 1; j < countOfPages; j++){

if(freePageNumbers[i + j] != freePageNumbers[i] + j){

i += j;

blockIsFound = false;

break;

}

}

if(blockIsFound){

vector<unsigned int> pageNumbers = vector<unsigned int>();

for(unsigned int j = i; j < i + countOfPages; j++){

fullFilledPageNumbers.push\_back(freePageNumbers[j]);

pageNumbers.push\_back(freePageNumbers[j]);

}

ret = memoryPages[freePageNumbers[i]].GetMemory();

bigBlocks.emplace(ret, pageNumbers);

freePageNumbers.erase(freePageNumbers.begin() + i, freePageNumbers.begin() + countOfPages);

break;

}

}

return ret;

}

bool PageMemoryAllocator::IsInMemoryRange(void \*addr) {

return NULL != addr && addr >= memory && addr <= (void \*)((unsigned int)memory + memorySize);

}

**Тести**

