НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування та спеціалізованих комп‘ютерних систем

**Лабораторна робота №1**

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

«Аллокатор пам’яті загального призначення (1 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.

Кожен блок пам'яті складається з заголовка фіксованого розміру і даних. У заголовку вказується розмір поточного блоку, розмір попереднього блоку і при-знак зайнятості блоку. Вибирається область пам'яті, яка містить хоча б один вільний блок. Далі в знайденої області пам'яті скануються всі блоки по порядку, і вибирається перший відповідний вільний блок. Знайдений блок поділяється на два, один позначається як вільний, а другий позначається як занятий і адреса його даних повертається з додатком. Якщо в знайденої області пам'яті не вдалося знайти вільний блок достатнього розміру, то проводиться пошук в іншій області пам'яті. При звільненні блоку проводиться склейка з лівим і правим вільними блоками.

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 "Allocator.h"

using namespace std;

int main() {

printf("Work Started!\n");

Allocator al = Allocator(3000);

al.Print();

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

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

void\* add1 = al.mem\_alloc(1000);

void\* add2 = al.mem\_alloc(500);

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

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

al.Print();

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

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

al.mem\_free(add2);

printf("After free.\n");

al.Print();

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

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

al.mem\_realloc(add1, 500);

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

al.Print();

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

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

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

al.mem\_realloc(add3, 1000);

al.Print();

std::getchar();

return 0;

}

#include "malloc.h"

#include "Block.h"

#include "string.h"

#include "cstring"

#include <vector>

using namespace std;

/\*Клас-аллокатор пам'яті.\*/

class Allocator {

public:

/\*Конструктор за замовченням.\*/

Allocator();

/\*Параметризований конструктор.\*/

Allocator(const unsigned int newMemorySize);

/\*Деструктор.\*/

~Allocator();

/\*Спроба виділеня області пам'яті заданого розміру.\*/

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

/\*Спроба звільнити задану область пам'яті.\*/

void mem\_free(void\* addr);

/\*Спроба змінити розмір блока з визначеною адресою.\*/

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

/\*Виводить у консоль інформвцію про блок.\*/

void Print() {

printf("\nMemory Size: %d\nCount of free blocks: %d\n", memorySize, (int)freeBlocks.size());

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

printf("\nCurrent block: %d", i + 1);

freeBlocks[i].Print();

}

printf("\nCount of allocated blocks: %d\n", (int)allocatedBlocks.size());

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

printf("\nCurrent block: %d", i + 1);

allocatedBlocks[i].Print();

}

}

private:

/\*Початкова ініціалізація змінних.\*/

void VarsInitialization(unsigned int newMemorySize);

/\*Початкове виділення пам'яті.\*/

void StartingMemoryAllocation();

/\*Спроба склеювання сусідніх вільних блоків пам'яті.\*/

void MergeFreeWith(Block block);

/\*Розмір пулу пам'яті у байтах.\*/

unsigned int memorySize;

/\*Власне пул.\*/

void\* memory;

/\*Заголовки порожніх блоків.\*/

vector<Block> freeBlocks;

/\*Заголовки зайнятих блоків.\*/

vector<Block> allocatedBlocks;

};

#include "Allocator.h"

Allocator::Allocator() {

VarsInitialization(0);

}

Allocator::Allocator(const unsigned int newMemorySize){

VarsInitialization(newMemorySize);

StartingMemoryAllocation();

freeBlocks.push\_back(Block(memorySize, memory));

}

Allocator::~Allocator(){

freeBlocks.clear();

allocatedBlocks.clear();

}

/\*Спроба видылення пам'яті заданого розміру.\*/

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

/\*Область пам'яті можна виділити, якщо її розмір не перевищує розміру доступної пам'яті.\*/

if(size > memorySize){

return NULL;

}

/\*Область для виділення слід шукати серед вільних блоків.\*/

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

/\*Потрібно знайти перший доступний за розміром блок.\*/

if(freeBlocks[i].Size() >= size){

/\*Якщо його розмір співпадає із потрібним, цей блок просто заноситься у список зайнятих.\*/

if(size == freeBlocks[i].Size()){

allocatedBlocks.push\_back(freeBlocks[i]);

freeBlocks.erase(freeBlocks.begin() + i);

return allocatedBlocks.end()->StartAddress();

}

/\*Інакше з нього виділяється блок потрібного розміру і заноситься у список зайнятих.\*/

Block allocatedBlock = \*freeBlocks[i].Allocate(size);

allocatedBlocks.push\_back(allocatedBlock);

return allocatedBlock.StartAddress();

}

}

/\*Якщо блоків потрібного роміру не знайшлось, про це сигналізується.\*/

return NULL;

}

/\*Спроба звільнити задану область пам'яті.\*/

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

/\*Блок для звільнення шукається серед зайнятих.\*/

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

if(allocatedBlocks[i].StartAddress() == addr){

/\*Якщо потрібний блок знайдено, він звільняється і склеюється із сусідніми вільними.\*/

MergeFreeWith(allocatedBlocks[i]);

allocatedBlocks.erase(allocatedBlocks.begin() + i);

return;

}

}

}

/\*Спроба злиття поточного вільного блоку із вільними лівішим і правішим (якщо такі є).\*/

void Allocator::MergeFreeWith(Block block){

Block temp = block;

bool onceMerged = false;

/\*Злиття із лівішим блоком.\*/

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

if((unsigned int \*)freeBlocks[i].StartAddress() + freeBlocks[i].Size() == (unsigned int \*)temp.StartAddress()){

freeBlocks[i].SetSize(freeBlocks[i].Size() + temp.Size());

temp.~Block();

temp = freeBlocks[i];

onceMerged = true;

break;

}

}

/\*Злиття із правішим блоком.\*/

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

if((unsigned int \*)temp.StartAddress() + temp.Size() == (unsigned int \*)freeBlocks[i].StartAddress()){

temp.SetSize(temp.Size() + freeBlocks[i].Size());

freeBlocks.erase(freeBlocks.begin() + i);

freeBlocks.push\_back(temp);

return;

}

}

if(!onceMerged){

freeBlocks.push\_back(block);

}

}

/\*Спроба змінити розмір блоку з заданою початковою адресою.\*/

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

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

if(allocatedBlocks[i].StartAddress() == addr){

/\*Якщо новий розмір співпадає із попереднім, то робити більше нічого.\*/

if(size == allocatedBlocks[i].Size()){

return addr;

}

void\* possibleDestination = mem\_alloc(size);

if(NULL != possibleDestination){

try {

if (size <= allocatedBlocks[i].Size()) {

memcpy(possibleDestination, addr, size);

}

else {

memcpy(possibleDestination, addr, allocatedBlocks[i].Size());

}

}

catch (const char\* message) {

cout << message << endl;

}

mem\_free(addr);

return possibleDestination;

}

}

}

return addr;

}

void Allocator::VarsInitialization(unsigned int newMemorySize){

memory = NULL;

memorySize = newMemorySize;

freeBlocks = vector<Block>();

allocatedBlocks = vector<Block>();

}

void Allocator::StartingMemoryAllocation(){

if(NULL == memory) {

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

}

}

#include <iostream>

#include "malloc.h"

#include "stdio.h"

/\*Містить інформацію про блок пам'яті.\*/

class Block {

public:

/\*Конструктор за замовченням.\*/

Block();

/\*Параметризований конструктор.\*/

Block(unsigned int newSize, void\* newStartAddress);

/\*Деструктор.\*/

~Block();

/\*Спроба виділити блок заданого розміру із даного блоку.\*/

Block\* Allocate(unsigned int sizeToAllocate);

/\*Встановлення нового розміру блоку.\*/

void SetSize(unsigned int newSize){

size = newSize;

}

/\*Повертає поточний розмір блоку.\*/

unsigned int Size(){

return size;

}

/\*Повертає початкову адресу блоку.\*/

void\* StartAddress(){

return startAddress;

}

/\*Виводить у консоль інформацію про блок.\*/

void Print(){

printf("\nSize: %d\tAddr %d\n", size, startAddress);

}

private:

/\*Повертає TRUE у випадку, якщо з цього блоку можна виділити визначену кількість пам'яті.\*/

bool CanAllocate(unsigned int sizeToAllocate){

return sizeToAllocate <= size;

}

/\*Розмір блоку.\*/

unsigned int size;

/\*Адреса початку блоку.\*/

void\* startAddress;

};

#include "Block.h"

Block::Block(){

size = 0;

startAddress = NULL;

}

Block::Block(unsigned int newSize, void \*newStartAddress){

size = newSize;

startAddress = newStartAddress;

}

Block::~Block(){

size = 0;

startAddress = NULL;

}

Block\* Block::Allocate(unsigned int sizeToAllocate){

/\*Розділення блоку можливе тільки, якщо цей блок вільний і не менший, ніж необхідно.\*/

if(/\*!(IsFree() &&\*/ !CanAllocate(sizeToAllocate)){

return NULL;

}

/\*

\* Якщо необхідний розмір співпадає із розміром блоку, то цей блок помічається як зайнятий.

\* Розділення блоку в такому випадку не відбувається.

\* \*/

if(sizeToAllocate == size){

return this;

}

/\*Якщо ж необхідний розмір менший за розмір блоку, то цей блок розділяється на дві частини.\*/

size -= sizeToAllocate;

void\* startAddressForNewBlock = (unsigned int \*)startAddress + size;

return new Block(sizeToAllocate, startAddressForNewBlock);

}

**Тести**

