Національний технічний університет України

«Київський політехнічний інститут»

Факультет інформатики і обчислювальної техніки

Кафедра обчислювальної техніки

**Лабораторна робота №2**

**з курсу: «Системне програмування»**

*Виконав:*

студент групи ІО-12

Васильєв О.В.

Залікова книжка №1206

*Перевірив:*

Дифучин А.Ю.

Київ, 2020 р.

**Тема**: аллокатор пам’яті загального призначення (частина 2).

**Мета**: розробити бiльш складний метод для аллокатору загального призначення.

**Опис розроблених алгоритмів**:

1. Ініціалізація загального пулу. Пул блоків пам’яті виділяється за допомогою стандартної функції виділення пам’яті malloc:

Кожен блок виділеної пам’яті містить заголовок:

unsigned is\_free: 1; // bit which shows is this block

// store any information

unsigned data\_size: 15; // block size in bytes [0 - 32767]

mem\_block \*prev\_block; // pointer on next block

mem\_block \*next\_block; // pointer on previous block

};

Відповідно до опису заголовок має займати 1 + 1 + 4 + 4 = 10 байт

Під час ініціалізації виділяється 3 блоки:

* Початковий зайнятий блок-бар’єр нульового розміру.
* Кінцевий зайнятий блок-бар’єр нульового розміру.
* Проміжний вільний блок.

1. Звільнення загального пулу. Виконується за допомогою стандартної функції allocSearchForFreeBlock
2. Виділення пам’яті заданого розміру mem\_alloc(size\_t size). Передбачені 2 алгоритми пошуку потрібного блоку для виділення пам’яті:

* За необхідний вибирається перший знайдений підходящий блок, тобто такий, розмір якого не менший за size.
* За необхідний вибирається підходящий блок з мінімальним розміром.

Якщо блок не знайдений, то повертається NULL, якщо знайдений – то виконується функція виділення пам’яті в заданому блоці.

1. Функція виділення пам’яті в заданому блоці allocFreeBlock (Header \*headerPtr, size\_t size).
2. Якщо розмір блоку дещо більший за size, то він ділиться на 2 блоки.
3. Якщо розмір блоку майже дорівнює size, то блок залишається неподільним. Повертається вказівник на заголовок блоку.
4. Перевиділення пам’яті void \*allocRealloc(allocator \*alloc, void \*data, size\_t size)
5. Якщо addr = NULL, то виконується виділення пам’яті заданого розміру mem\_alloc(size).
6. Якщо розмір блоку, що виділений під данні вказівника addr, більший за size, то перевиділяємо пам’ять блоку за допомогою freeBlockAlloc(headerPtr, size).
7. У протилежному випадку намагаємося об’єднати даний блок з вільним правим (без перенесення даних) і/або лівим (з перенесенням даних) й перевиділити пам’ять блоку.
8. Якщо цього виявилося недостатньо, то знаходимо вільне місце для даних з допомогою mem\_alloc(), переносимо дані й звільняємо даний блок.
9. Звільнення пам’яті mem\_free(void \*addr). Помічаємо блок як вільний і об’єднуємо даний блок с сусідніми вільними блоками.

**Оцінка часу пошуку вільного блоку пам’яті**:

* Алгоритм «перший підходящий блок»: O(N).
* Алгоритм «підходящий з мінімальним розміром»: O(2 \* N).

N – кількість блоків в пулі.

**Оцінка часу звільнення занятого блоку**: O(1).

**Оцінка витрати пам'яті для зберігання службовох інформації:** 12 \* N байт.

N – кількість блоків в пулі.

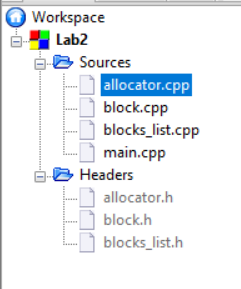
**Переваги аллокатора:**

1. Невеликий об’єм службової інформації.
2. Малий час виконання звільнення блоку.

**Недоліки аллокатора:**

1. Лінійниний час пошуку вільного блоку (повільно).
2. Можлива передчасна фрагментація пам’яті.

**Структура проекту:**



**Лістинг заготовочного файлу allocator.h:**

#ifndef ALLOCATOR\_H

#define ALLOCATOR\_H

#include <string>

#include <sstream>

#include <iomanip>

#include "blocks\_list.h"

using namespace std;

class AllocHeader

{

public:

typedef unsigned char byte;

const static int categories\_amount = 12;

const static int block\_sizes[categories\_amount];

void init(int size);

bool is\_suit\_size(int size);

Block \*get\_suit\_block(int size);

static bool can\_make\_twin(Block \*block, int needed\_size);

Block \*make\_suit\_block(int from\_block\_class, int size);

void join\_twins\_recursive(Block \*block);

Block \*get\_twin(Block \*block);

std::string dump();

BlocksList \*free\_blocks[categories\_amount];

private:

AllocHeader();

Block \*merge\_blocks(Block \*first\_block, Block \*second\_block);

};

class Allocator

{

private:

AllocHeader \*header;

public:

typedef unsigned char byte;

Allocator(int size=65536);

~Allocator();

std::string dump();

void \*alloc(int size);

void free(void \*data);

bool test();

};

#endif

**Лістинг заготовочного файлу block.h:**

#ifndef BLOCK\_H

#define BLOCK\_H

class Block

{

private:

Block();

public:

unsigned is\_free: 1;

unsigned size: 23;

void init(int size=0);

int free\_size();

void \*get\_data\_ptr();

static int size\_ommit\_header(int size);

static Block \*data\_ptr\_to\_block\_ptr(void \*data);

// int get\_size();

};

#endif

**Лістинг заготовочного файлу block\_list.h:**

#ifndef BLOCKS\_LIST\_H

#define BLOCKS\_LIST\_H

#include <iostream>

#include "block.h"

class BlocksListNode

{

public:

Block \*data;

BlocksListNode \*next;

BlocksListNode();

BlocksListNode(Block \*block);

};

class BlocksList

{

private:

BlocksListNode \*head;

BlocksListNode \*tail;

public:

BlocksList();

~BlocksList();

void push(Block \*elem);

Block \*pop();

void del(Block \*elem);

bool is\_empty();

int len();

// void dump()

// {

// BlocksListNode \*node = this->head;

// int i = 0;

// while (node != nullptr) {

// std::cout << i << " > " << (int\*)(node->data) << std::endl;

// i++;

// node = node->next;

// }

// }

};

#endif

**Лістинг виконавчого файлу allocator.cpp:**

#include "allocator.h"

#include <iostream>

const int AllocHeader::block\_sizes[AllocHeader::categories\_amount] = {

32, 64, 128,

256, 512, 1024,

2048, 4096, 8192,

16384, 32768, 65536,

};

AllocHeader::AllocHeader()

{

}

void AllocHeader::init(int size)

{

// finding pointer on first data block

Block \*first\_block = (Block \*)((AllocHeader::byte \*)this + sizeof(AllocHeader));

first\_block->init(size);

// init array with empty lists

for (int i = 0; i < AllocHeader::categories\_amount; i++) {

this->free\_blocks[i] = new BlocksList();

}

// add first block to array of free blocks

for (int i = AllocHeader::categories\_amount - 1; i >= 0; i--) {

if (size == AllocHeader::block\_sizes[i]) {

this->free\_blocks[i]->push(first\_block);

return;

}

}

throw "Invalid allocator size.";

}

Block \*AllocHeader::get\_suit\_block(int size)

{

for (int i = 0; i < AllocHeader::categories\_amount; i++) {

if (Block::size\_ommit\_header(AllocHeader::block\_sizes[i]) >= size &&

!this->free\_blocks[i]->is\_empty())

{

return this->make\_suit\_block(i, size);

}

}

}

bool AllocHeader::can\_make\_twin(Block \*block, int needed\_size)

{

if (block->size / 2 - sizeof(Block) >= needed\_size) {

return true;

}

return false;

}

Block \*AllocHeader::make\_suit\_block(int from\_block\_class, int size)

{

// std::cout << this->dump() << std::endl;

Block \*block = this->free\_blocks[from\_block\_class]->pop();

if (from\_block\_class > 0 && this->can\_make\_twin(block, size)) {

int twin\_block\_size = block->size / 2;

Block \*twin\_block = (Block \*)((AllocHeader::byte \*)block + twin\_block\_size);

twin\_block->init(twin\_block\_size);

block->size = twin\_block\_size;

this->free\_blocks[from\_block\_class - 1]->push(block);

this->free\_blocks[from\_block\_class - 1]->push(twin\_block);

return make\_suit\_block(from\_block\_class - 1, size);

}

block->is\_free = 0;

return block;

}

void AllocHeader::join\_twins\_recursive(Block \*block)

{

// exit funtion if current block is the biggest possible block

if (block->size == AllocHeader::block\_sizes[AllocHeader::categories\_amount - 1]) {

return;

}

// cout << "block is not biggest\n";

// calculate offset to block from allocator start

int offset\_to\_block = (AllocHeader::byte \*)block - ((AllocHeader::byte \*)this + sizeof(AllocHeader));

// offset\_to\_block = (AllocHeader::byte \*)block - 32;

// cout << "TTT size: " << (twin\_block->size << endl;

// finding the power of two from block size

int block\_size = block->size;

int power = -1;

while (block\_size > 0) {

power++;

block\_size >>= 1;

}

// cout << "offset\_to\_block: " << offset\_to\_block << endl;

// invert bit which located on power position

offset\_to\_block ^= 1 << power;

// cout << "offset\_to\_block: " << offset\_to\_block << endl;

// fingind twin block position

Block \*twin\_block = (Block \*)((AllocHeader::byte \*)this + sizeof(AllocHeader) + offset\_to\_block);

// cout << "twin block size: " << twin\_block->size << endl;

if (block->size == twin\_block->size &&

twin\_block->is\_free)

{

Block \*merged\_block = this->merge\_blocks(block, twin\_block);

if (merged\_block != nullptr) {

// this->free\_blocks[power - 4]->push(merged\_block);

this->join\_twins\_recursive(merged\_block);

}

}

}

Block \*AllocHeader::merge\_blocks(Block \*first\_block, Block \*second\_block)

{

if (first\_block > second\_block) {

return AllocHeader::merge\_blocks(second\_block, first\_block);

}

int block\_category = 0;

for (int i = 0; i < AllocHeader::categories\_amount; i++) {

if (AllocHeader::block\_sizes[i] == first\_block->size) {

block\_category = i;

break;

}

}

this->free\_blocks[block\_category]->del(first\_block);

this->free\_blocks[block\_category]->del(second\_block);

first\_block->size = first\_block->size \* 2;

this->free\_blocks[block\_category + 1]->push(first\_block);

return first\_block;

}

Block \*AllocHeader::get\_twin(Block \*block)

{

// int

}

std::string AllocHeader::dump()

{

std::string separator = "=========[ Memory dump ]=========";

std::stringstream output;

output << separator << std::endl;

for (int i = 0; i < AllocHeader::categories\_amount; i++) {

int size = AllocHeader::block\_sizes[i];

int amount = this->free\_blocks[i]->len();

output << "Block\_size: " << std::setw(6) << size << " => Amount: " << amount << std::endl;

}

output << "=================================" << std::endl;

return output.str();

}

Allocator::Allocator(int size)

{

void \*mem = (void \*) malloc(sizeof(AllocHeader) + size);

this->header = (AllocHeader \*) mem;

this->header->init(size);

}

Allocator::~Allocator()

{ }

std::string Allocator::dump()

{

return this->header->dump();

}

void \*Allocator::alloc(int size)

{

return header->get\_suit\_block(size)->get\_data\_ptr();

}

void Allocator::free(void \*data)

{

Block \*block = Block::data\_ptr\_to\_block\_ptr(data);

// std::cout << block->size << std::endl;

block->is\_free = 1;

// std::cout << block->is\_free << std::endl;

int block\_size = block->size;

int power = -1;

while (block\_size > 0) {

power++;

block\_size >>= 1;

}

this->header->free\_blocks[power - 5]->push(block);

header->join\_twins\_recursive(block);

}

bool Allocator::test()

{

for (int i = 0; i < AllocHeader::categories\_amount - 1; i++) {

if (!this->header->free\_blocks[i]->is\_empty()) {

return false;

}

}

if (this->header->free\_blocks[AllocHeader::categories\_amount - 1]->is\_empty()) {

return false;

}

return true;

}

**Лістинг виконавчого файлу block.cpp:**#include "block.h"

Block::Block()

{ }

void Block::init(int size)

{

this->is\_free = 1;

this->size = size;

}

int Block::free\_size()

{

return size - sizeof(Block);

}

int Block::size\_ommit\_header(int size)

{

return size - sizeof(Block);

}

void \*Block::get\_data\_ptr()

{

return (void \*)((unsigned char \*)this + sizeof(Block));

}

Block \*Block::data\_ptr\_to\_block\_ptr(void \*data)

{

return (Block \*)((unsigned char \*)data - sizeof(Block));

}

**Лістинг виконавчого файлу block\_list.cpp:\**

#include "blocks\_list.h"

BlocksListNode::BlocksListNode(): data(nullptr), next(nullptr)

{ }

BlocksListNode::BlocksListNode(Block \*block) : data(block), next(nullptr)

{ }

BlocksList::BlocksList()

{

this->head = nullptr;

this->tail = nullptr;

}

void BlocksList::push(Block \*block)

{

BlocksListNode \*node = new BlocksListNode(block);

if (this->tail == nullptr) {

this->head = node;

this->tail = node;

}

else {

this->tail->next = node;

this->tail = node;

}

}

Block \*BlocksList::pop()

{

Block \*block = this->head->data;

if (this->head == this->tail) {

this->head = this->tail = nullptr;

return block;

}

BlocksListNode \*poped\_node = this->head;

this->head = this->head->next;

delete poped\_node;

return block;

}

void BlocksList::del(Block \*elem)

{

if (this->head->data == elem) {

if (this->head == this->tail) {

this->tail = nullptr;

}

this->head = this->head->next;

return;

}

BlocksListNode \*node = this->head;

while (node != nullptr) {

if (node->next->data == elem) {

if (node->next == this->tail) {

this->tail = node;

}

BlocksListNode \*deleting\_node = node->next;

node->next = deleting\_node->next;

delete deleting\_node;

return;

}

node = node->next;

}

}

bool BlocksList::is\_empty()

{

if (this->head == nullptr) {

return true;

}

return false;

}

int BlocksList::len()

{

int len = 0;

BlocksListNode \*node = this->head;

while (node != nullptr) {

len++;

node = node->next;

}

return len;

}

**Лістинг виконавчого файлу main.cpp:**

#include <iostream>

#include <ctime>

#include "allocator.h"

using namespace std;

int main()

{

Allocator alloc;

// srand(time(NULL));

// int test\_amount = 100;

// int mem\_alloc\_amount = 10;

// int max\_mem = 1024;

// int bad\_tests = test\_amount;

// for (int test = 0; test < test\_amount; test++) {

// void \*mem[mem\_alloc\_amount];

// for (int i = 0; i < mem\_alloc\_amount; i++) {

// int size = rand() % max\_mem;

// mem[i] = alloc.alloc(size);

// }

// for (int i = 0; i < mem\_alloc\_amount; i++) {

// alloc.free(mem[i]);

// }

// if (alloc.test()) {

// bad\_tests--;

// }

// }

// cout << "Runned " << test\_amount << endl;

// if (bad\_tests == 0) {

// cout << "All tests are correct." << endl;

// }

// else {

// cout << bad\_tests << " tests was bad!" << endl;

// }

cout << alloc.dump() << endl;

cout << "Allocating memory for x variable..." << endl;

int \*x = (int \*) alloc.alloc(4);

cout << alloc.dump() << endl;

cout << "Assign x to 5..." << endl;

\*x = 5;

cout << "X variable consist " << \*x << endl;

cout << "Free memory that was allocated for x..." << endl;

alloc.free(x);

cout << alloc.dump() << endl;

return 0;

}

**Приклад роботи аллокатора:**

