Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Курсовой проект по курсу**

**«Операционные системы»**

**Тема работы**

**Приобретение практических навыков в использовании знаний, полученных в течении курса.**

Студент: Полонский Кирилл Андреевич

Группа: М8О-208Б-20

Вариант: 16

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2021

**Содержание**

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Демонстрация работы программы
7. Выводы

**Постановка задачи**

Исследование 2 аллокаторов памяти: необходимо реализовать два алгоритма аллокации памяти и сравнить их по следующим характеристикам:

* Фактор использования
* Скорость выделения блоков
* Скорость освобождения блоков
* Простота использования аллокатора

Вариант 18: Необходимо сравнить два алгоритма аллокации: алгоритм двойников и блоки по 2 в степени n.

**Общие сведения о программе**

Файлы allocator\_pow\_of\_2.h и Bubby.h содержат реализацию аллокаторов. Файл main.cpp и main1.cpp содержат пример использования использования первого и второго аллокатора соответственно и демонстрацию их работы.

**Общий метод и алгоритм решения**

Алгоритм аллокации блоков степени 2 заключается в хранении списков указателей на свободные блоки памяти одного размера, каждый блок хранит указатель на следующий блок, если он свободен, или на указатель на голову если не свободен. Аллокатор на алгоритме двойников подразумевает, что 2 блоки памяти являются блоками размера степени двойки. Когда мы просим память, аллокатор ищет блок подходящего размера. Если мы просим память размера меньше половины блока, блок нужно разбить пополам. Так и получается блок и его двойник. Блок можно разбивать и дальше, пока мы не получим наименьший подходящий кусок памяти. Свободные блоки можно сливать с двойниками. Для этого его двойник должен тоже быть свободен. Каждый блок содержит размер, статус свободен-занят, адрес следующего блока же можно узнать на основании этих данных. При поиске блока мы разбиваем наш единственный блок, если вся доступная аллокатору память пока что - единый участок или проходимся по нашему неявному списку в поисках подходящего блока. Сливание происходит в отдельной функции, которая работает над списком, пока не останется подходящих для объединения блоков.

**Исходный код**

allocator\_pow\_of\_2.h

#pragma once

#include <iostream>

#include <unordered\_map>

#include <vector>

#include <math.h>

class Allocator\_pow\_of\_2{

public:

struct Header {

Header \* next;

};

std::unordered\_map<size\_t, Header \*> free\_blocks\_lists;

std::vector<size\_t> powsOf2 = {16, 32, 64, 128, 256, 512, 1024};

void \* start\_point;

void \* data;

static size\_t align(size\_t size) {

int i = 0;

while (pow(2, i) < size){

i++;

}

return (size\_t) pow(2, i);

}

Allocator\_pow\_of\_2(std::vector<size\_t>& blocks\_amount);

~Allocator\_pow\_of\_2();

void\* allocate(size\_t bytes\_amount);

void deallocate(void \*ptr);

};

Allocator\_pow\_of\_2::Allocator\_pow\_of\_2(std::vector<size\_t>& blocks\_amount){

size\_t bytes\_sum = 0;

for(size\_t i = 0; i < blocks\_amount.size(); i++){

bytes\_sum += blocks\_amount[i] \* (sizeof(Header \*) + powsOf2[i]);

}

start\_point = malloc(bytes\_sum + (sizeof(Header \*) \* blocks\_amount.size()));

data = start\_point;

void \* point\_now = start\_point;

for(size\_t i = 0; i < blocks\_amount.size(); i++){

Header \* head = (Header \*) point\_now;

free\_blocks\_lists[powsOf2[i]] = head;

point\_now = (void \*)((char \*)point\_now + sizeof(Header \*));

for(size\_t j = 0; j < blocks\_amount[i]; j++){

head->next = (Header \*) point\_now;

head = head->next;

point\_now = (void \*)((char \*)point\_now + powsOf2[i] + sizeof(Header \*));

}

head->next = nullptr;

}

}

void \* Allocator\_pow\_of\_2::allocate(size\_t bytes\_amount){

if(bytes\_amount == 0){

std::cerr << "Size must be bigger than 0\n";

return nullptr;

}

size\_t size = align(bytes\_amount);

Header \* head = free\_blocks\_lists[size];

if(head->next == nullptr){

std::cerr << "Zero free element\n";

return nullptr;

}

Header \* alloc\_elem = head->next;

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

alloc\_elem->next = head;

return (void \*) ((char \*)alloc\_elem + sizeof(Header \*));

}

void Allocator\_pow\_of\_2::deallocate(void \* ptr){

Header \* ptr\_header = (Header \*)((char \*)ptr - sizeof(Header \*));

for(size\_t i = 0; i < free\_blocks\_lists.size(); i++){

if (free\_blocks\_lists[powsOf2[i]] == ptr\_header->next){

ptr\_header->next = free\_blocks\_lists[powsOf2[i]]->next;

free\_blocks\_lists[powsOf2[i]]->next = ptr\_header;

return;

}

}

std::cerr << "Uncorrect ptr\n";

}

Allocator\_pow\_of\_2::~Allocator\_pow\_of\_2(){

free(data);

}

Bubby.h

#pragma once

#include <iostream>

#include <cstring>

#include <cmath>

struct BuddyBlock {

size\_t blockSize;

bool isFree;

};

class BuddyAllocator {

private:

BuddyBlock \*head = nullptr;

BuddyBlock \*tail = nullptr;

void \*data = nullptr;

bool expanded = false;

BuddyBlock\* next(BuddyBlock \*block) {

return (BuddyBlock \*)((char \*)(block) + block->blockSize);

}

BuddyBlock\* split(BuddyBlock \*block, size\_t size) {

if (block != nullptr && size != 0) {

while (size < block->blockSize) {

size\_t sz = block->blockSize >> 1;

block->blockSize = sz;

block = this->next(block);

block->blockSize = sz;

block->isFree = true;

}

if (size <= block->blockSize){

return block;

}

}

return nullptr;

}

BuddyBlock\* find\_best(size\_t size) {

if (size == 0) return nullptr;

BuddyBlock \*bestBlock = nullptr;

BuddyBlock \*block = this->head;

BuddyBlock \*buddy = this->next(block);

if (buddy == this->tail && block->isFree) {

return this->split(block, size);

}

while (block < this->tail && buddy < this->tail) {

if (block->isFree && buddy->isFree && block->blockSize == buddy->blockSize) {

block->blockSize <<= 1;

if (size <= block->blockSize && (bestBlock == nullptr || block->blockSize <= bestBlock->blockSize)) {

bestBlock = block;

}

block = this->next(buddy);

if (block < this->tail) {

buddy = this->next(block);

}

continue;

}

if (block->isFree && size <= block->blockSize && (bestBlock == nullptr || block->blockSize <= bestBlock->blockSize)) {

bestBlock = block;

}

if (buddy->isFree && size <= buddy->blockSize && (bestBlock == nullptr || buddy->blockSize < bestBlock->blockSize)) {

bestBlock = buddy;

}

if (block->blockSize <= buddy->blockSize) {

block = this->next(buddy);

if (block < this->tail) {

buddy = this->next(block);

}

}

else {

block = buddy;

buddy = this->next(buddy);

}

}

if (bestBlock != nullptr) {

return this->split(bestBlock, size);

}

return nullptr;

}

void coalescence() {

while (true) {

BuddyBlock \*block = this->head;

BuddyBlock \*buddy = this->next(block);

bool no\_coalescence = true;

while (block < this->tail && buddy < this->tail) {

if (block->isFree && buddy->isFree && block->blockSize == buddy->blockSize) {

block->blockSize <<= 1;

block = this->next(block);

if (block < this->tail) {

buddy = this->next(block);

no\_coalescence = false;

}

}

else if (block->blockSize < buddy->blockSize) {

block = buddy;

buddy = this->next(buddy);

}

else {

block = this->next(buddy);

if (block < this->tail) {

buddy = this->next(block);

}

}

}

if (no\_coalescence) {

return;

}

}

}

static size\_t align(size\_t size) {

int i = 0;

while (pow(2, i) < size){

i++;

}

return (size\_t) pow(2, i);

}

public:

BuddyAllocator(size\_t size) {

this->expand(size);

}

~BuddyAllocator() {

this->head = nullptr;

this->tail = nullptr;

std::free(this->data);

}

void expand(size\_t size) {

if (this->head) {

size += this->head->blockSize;

}

size = align(size);

this->data = std::realloc(this->data, size);

this->head = (BuddyBlock \*)(data);

this->head->blockSize = size;

this->head->isFree = true;

this->tail = next(head);

}

void \*allocate(size\_t size) {

if (size == 0) {

std::cerr << "Size must be bigger than 0\n";

return nullptr;

}

size\_t new\_size = align(size + sizeof(BuddyBlock));

BuddyBlock \*found = this->find\_best(new\_size);

if (found == nullptr) {

this->coalescence();

found = this->find\_best(new\_size);

}

if (found != nullptr) {

found->isFree = false;

this->expanded = false;

return (void \*)((char \*)(found) + sizeof(BuddyBlock));

}

if (this->expanded) {

this->expanded = false;

return nullptr;

}

this->expanded = true;

this->expand(size);

return this->allocate(size);

}

void deallocate(void \*ptr) {

if (ptr == nullptr) {

std::cerr << "Bad ptr";

return;

}

BuddyBlock \*block = (BuddyBlock \*)((char \*)(ptr) - sizeof(BuddyBlock));

block->isFree = true;

this->coalescence();

}

};

Main.cpp

#include "Bubby.h"

#include <vector>

#include <chrono>

int main(){

auto t1 = std::chrono::steady\_clock::now();

BuddyAllocator allocator = BuddyAllocator(4096);

auto t2 = std::chrono::steady\_clock::now();

auto duration = std::chrono::duration\_cast<std::chrono::nanoseconds>(t2 - t1).count();

std::cout << "Time to create BuddyAllocator : " << duration << '\n';

std::vector<void\*> v(1000);

auto t3 = std::chrono::high\_resolution\_clock::now();

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

v[i] = allocator.allocate(i);

}

auto t4 = std::chrono::high\_resolution\_clock::now();

auto duration1 = std::chrono::duration\_cast<std::chrono::nanoseconds>(t4 - t3).count();

std::cout << "Time to allocate BuddyAllocator : " << duration1 << '\n';

auto t5 = std::chrono::high\_resolution\_clock::now();

for (auto & i : v) {

allocator.deallocate(i);

}

auto t6 = std::chrono::high\_resolution\_clock::now();

auto duration2 = std::chrono::duration\_cast<std::chrono::nanoseconds>(t6 - t5).count();

std::cout << "Time to allocate BuddyAllocator : " << duration2 << '\n';

}

Main1.cpp

#include "allocator\_pow\_of\_2.h"

#include <vector>

#include <chrono>

int main(){

std::vector<size\_t> k = {16, 16, 32, 64, 128, 256, 512};

auto t1 = std::chrono::high\_resolution\_clock::now();

Allocator\_pow\_of\_2 alloc(k);

auto t2 = std::chrono::high\_resolution\_clock::now();

auto duration = std::chrono::duration\_cast<std::chrono::nanoseconds>(t2 - t1).count();

std::cout << "Time to create Allocator\_pow\_of\_2 : " << duration << '\n';

std::vector<void\*> v1(1000);

auto t3 = std::chrono::high\_resolution\_clock::now();

for (size\_t i = 9; i < v1.size(); i++) {

v1[i] = alloc.allocate(i);

}

auto t4 = std::chrono::high\_resolution\_clock::now();

auto duration1 = std::chrono::duration\_cast<std::chrono::nanoseconds>(t4 - t3).count();

std::cout << "Time to allocate Allocator\_pow\_of\_2 : " << duration1 << '\n';

auto t5 = std::chrono::high\_resolution\_clock::now();

for(size\_t i = 9; i < v1.size(); i++){

alloc.deallocate(v1[i]);

}

auto t6 = std::chrono::high\_resolution\_clock::now();

auto duration2 = std::chrono::duration\_cast<std::chrono::nanoseconds>(t6 - t5).count();

std::cout << "Time to allocate Allocator\_pow\_of\_2 : " << duration2 << '\n';

}

**Демонстрация работы программы**

|  |
| --- |
| PS C:\Users\Bonik\Documents\kp> ./a.exe  Time to create BuddyAllocator : 1100  Time to allocate BuddyAllocator : 382600  Time to allocate BuddyAllocator : 57100  PS C:\Users\Bonik\Documents\kp> ./a.exe  Time to create Allocator\_pow\_of\_2 : 218000  Time to allocate Allocator\_pow\_of\_2 : 205400  Time to allocate Allocator\_pow\_of\_2 : 408100 |

**Выводы**

В ходе выполнения курсового проекта я закрепил навыки работы с аллокаторами. Из результатов работы программы видно, что алгоритм двойников быстрее на этапе создания и деаллокации, в то время как алгоритм блоков 2 в степени n значительно быстрее на этапе аллокации. Отсюда можно сделать вывод, что по факту использования алгоритм блоков 2 в степени n стоит использовать в случае частой аллкоации и редкой деаллокации данных при условии того, что мы знаем примерный диапазон размеров входных данных. А в случае необходимости гибкости в диапазоне размера входных данных, а также при малом количестве данных использовании аллокатора.