Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Курсовой проект по курсу**

**«Операционные системы»**

**Тема**

**«Сравнение алгоритмов аллокации памяти»**

Студент: Эсмедляев Федор Романович

Группа: М8О–212Б–22

Вариант: 22

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

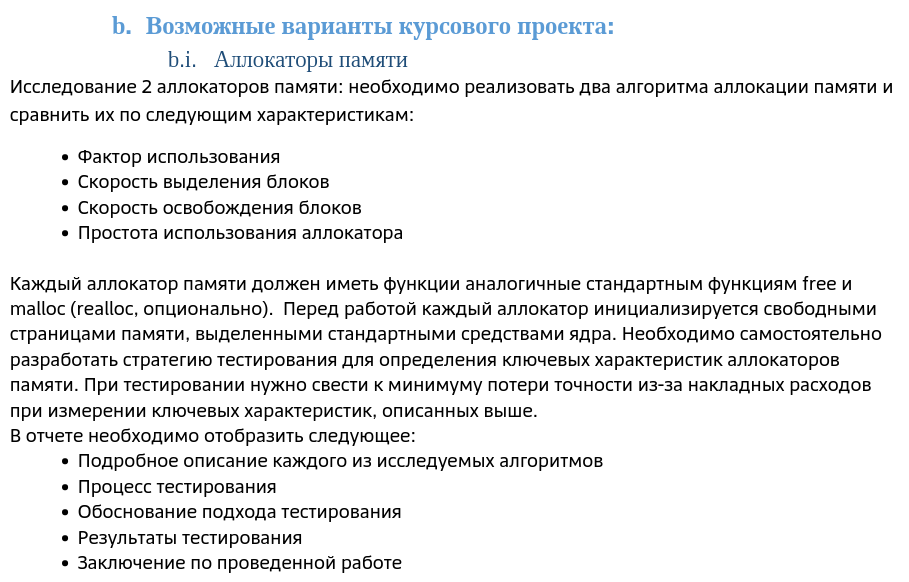
Москва, 2023.

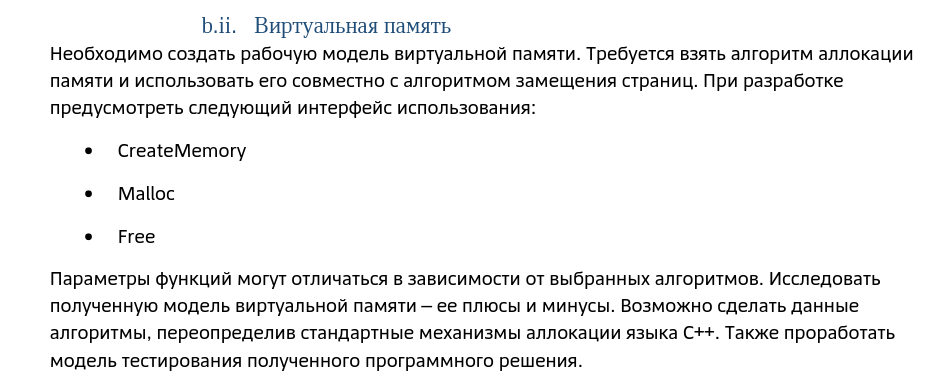
**Постановка задачи**

## Задание

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

2. Проведение исследования в выбранной предметной области



****

**Общие сведения о программе**

Основные файлы — файлы аллокации, представлены в виде .h и .cpp. В них реализованы 2 алгоритма аллокации списки свободных блоков (первое

подходящее) и алгоритм Мак-Кьюзи-Кэрелса. Все тесты происходят при совместном использовании run.cpp и benchmark.cpp.

**Алгоритм решения**

Списки свободных блоков(первое подходящее):

1)Выделение

а)Находим нужный блок памяти

б)Уменьшаем его размер на N

в)Если есть свободный блок длинной N+1 нельзя выделить из него блок длиной N

2)Освобождение

а)Находим место для вставки освобожденного блока

б)При необходимости производим слияние соседних блоков.

в)С односвязным списком сложность O(n)

Мак-Кьюзи-Кэрелся:

1)Выделение

а)При запросе памяти округляем запрос до степени 2

б)При выделении блока в блоке не требуется хранить дополнительную информацию

в)Если нет блока нужного размера, то выделяем новую страницу, которую разбиваем на блоки данного размера

2)Освобождение

а)Для освобождения не требуется указывать размер блока

**Основные файлы программы**

**Iallocator.h**

#pragma once

#include <exception>

#include <iostream>

#include <math.h>

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <stdbool.h>

#include <sys/mman.h>

class Allocator {

public:

using void\_pointer = void\*;

using size\_type = std::size\_t;

using difference\_type = std::ptrdiff\_t;

using propagate\_on\_container\_move\_assignment = std::true\_type;

using is\_always\_equal = std::true\_type;

protected:

Allocator() = default;

public:

virtual ~Allocator() = default;

virtual void\_pointer alloc(const size\_type) = 0;

virtual void free(void\_pointer) = 0;

};

**ListAllocator.h**

#pragma once

#include "IAllocator.h"

struct BlockHeader {

size\_t \_size;

BlockHeader\* \_next;

};

class ListAllocator final : public Allocator {

private:

BlockHeader\* \_free\_blocks\_list;

public:

ListAllocator() = delete;

ListAllocator(void\_pointer, size\_type);

virtual ~ListAllocator();

virtual void\_pointer alloc(size\_type) override;

virtual void free(void\_pointer) override;

};

**MacKuseyCaretsAllocator.h**

#pragma once

#include "IAllocator.h"

struct Page {

Page\* \_next;

bool \_is\_large;

size\_t \_block\_size;

};

class MacKuseyCarelsAllocator final : public Allocator {

private:

void\* \_memory;

Page\* \_free\_pages\_list;

size\_t \_memory\_size;

size\_t \_page\_size;

public:

MacKuseyCarelsAllocator() = delete;

MacKuseyCarelsAllocator(void\_pointer, size\_type);

virtual ~MacKuseyCarelsAllocator();

virtual void\_pointer alloc(size\_type) override;

virtual void free(void\_pointer) override;

};

**ListAllocator.cpp**

#include "../../include/allocator/ListAllocator.h"

ListAllocator::ListAllocator(void\_pointer real\_memory, size\_type memory\_size)

{

\_free\_blocks\_list = reinterpret\_cast<BlockHeader\*>(real\_memory + sizeof(ListAllocator));

\_free\_blocks\_list->\_size = memory\_size - sizeof(ListAllocator) - sizeof(BlockHeader);

\_free\_blocks\_list->\_next = nullptr;

}

ListAllocator::~ListAllocator()

{

BlockHeader\* cur\_block = this->\_free\_blocks\_list;

while (cur\_block) {

BlockHeader\* to\_delete = cur\_block;

cur\_block = cur\_block->\_next;

to\_delete = nullptr;

}

this->\_free\_blocks\_list = nullptr;

}

typename Allocator::void\_pointer ListAllocator::alloc(size\_type new\_block\_size)

{

BlockHeader\* prev\_block = nullptr;

BlockHeader\* cur\_block = this->\_free\_blocks\_list;

size\_type adjusted\_size = new\_block\_size + sizeof(BlockHeader);

while (cur\_block) {

if (cur\_block->\_size >= adjusted\_size) {

if (cur\_block->\_size >= adjusted\_size + sizeof(BlockHeader)) {

BlockHeader\* new\_block = reinterpret\_cast<BlockHeader\*>(reinterpret\_cast<int8\_t\*>(cur\_block) + adjusted\_size);

new\_block->\_size = cur\_block->\_size - adjusted\_size - sizeof(BlockHeader);

new\_block->\_next = cur\_block->\_next;

cur\_block->\_next = new\_block;

cur\_block->\_size = adjusted\_size;

}

if (prev\_block) {

prev\_block->\_next = cur\_block->\_next;

} else {

this->\_free\_blocks\_list = cur\_block->\_next;

}

return reinterpret\_cast<int8\_t\*>(cur\_block) + sizeof(BlockHeader);

}

prev\_block = cur\_block;

cur\_block = cur\_block->\_next;

}

return nullptr;

}

void ListAllocator::free(void\_pointer block)

{

if (block == nullptr) return;

BlockHeader\* header = reinterpret\_cast<BlockHeader\*>(static\_cast<int8\_t\*>(block) - sizeof(BlockHeader));

header->\_next = this->\_free\_blocks\_list;

this->\_free\_blocks\_list = header;

}

**MacKuseyCarestAllocator.cpp**

#include "../../include/allocator/MacKuseyCarelsAllocator.h"

MacKuseyCarelsAllocator::MacKuseyCarelsAllocator(void\_pointer real\_memory, size\_type memory\_size)

{

\_memory = reinterpret\_cast<void\*>(reinterpret\_cast<int8\_t\*>(real\_memory) + sizeof(MacKuseyCarelsAllocator));

\_free\_pages\_list = nullptr;

\_memory\_size = memory\_size - sizeof(MacKuseyCarelsAllocator);

\_page\_size = getpagesize();

}

MacKuseyCarelsAllocator::~MacKuseyCarelsAllocator()

{

Page\* cur\_page = this->\_free\_pages\_list;

while (cur\_page) {

Page\* to\_delete = cur\_page;

cur\_page = cur\_page->\_next;

munmap(to\_delete, \_page\_size);

to\_delete = nullptr;

}

\_free\_pages\_list = nullptr;

}

typename Allocator::void\_pointer MacKuseyCarelsAllocator::alloc(size\_type new\_block\_size)

{

if (\_memory\_size < new\_block\_size) return nullptr;

size\_t rounded\_block\_size = 1;

while (rounded\_block\_size < new\_block\_size) {

rounded\_block\_size \*= 2;

}

Page\* prev\_page = nullptr;

Page\* cur\_page = \_free\_pages\_list;

while (cur\_page) {

if (!cur\_page->\_is\_large && cur\_page->\_block\_size == rounded\_block\_size) {

void\_pointer block = reinterpret\_cast<void\_pointer>(cur\_page);

\_free\_pages\_list = cur\_page->\_next;

\_memory\_size -= new\_block\_size;

return block;

}

prev\_page = cur\_page;

cur\_page = cur\_page->\_next;

}

if (\_memory\_size < \_page\_size) return nullptr;

Page\* new\_page = reinterpret\_cast<Page\*>(mmap(NULL, \_page\_size,

PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS,

-1, 0));

if (new\_page == MAP\_FAILED) {

throw std::bad\_alloc();

}

new\_page->\_is\_large = false;

new\_page->\_block\_size = rounded\_block\_size;

new\_page->\_next = nullptr;

size\_t num\_blocks = \_page\_size / rounded\_block\_size;

for (size\_t i = 0; i != num\_blocks; ++i) {

Page\* block\_page = reinterpret\_cast<Page\*>(reinterpret\_cast<int8\_t\*>(new\_page) + i \* rounded\_block\_size);

block\_page->\_is\_large = false;

block\_page->\_block\_size = rounded\_block\_size;

block\_page->\_next = this->\_free\_pages\_list;

this->\_free\_pages\_list = block\_page;

}

void\_pointer block = reinterpret\_cast<void\_pointer>(new\_page);

this->\_free\_pages\_list = new\_page->\_next;

return block;

}

void MacKuseyCarelsAllocator::free(void\_pointer block)

{

if (block == nullptr) return;

Page\* page = reinterpret\_cast<Page\*>(block);

page->\_next = \_free\_pages\_list;

\_free\_pages\_list = page;

}

**benchmark.cpp**

#include <chrono>

#include <cstdlib>

#include <vector>

#include "../include/allocator/ListAllocator.h"

#include "../include/allocator/MacKuseyCarelsAllocator.h"

size\_t page\_size = sysconf(\_SC\_PAGESIZE);

void benchmark() {

void\* list\_memory = sbrk(10000 \* page\_size);

void\* MKC\_memory = mmap(NULL, 1000 \* page\_size, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0);

ListAllocator list\_alloc(list\_memory, 10000 \* page\_size);

MacKuseyCarelsAllocator MKC\_alloc(MKC\_memory, 1000 \* page\_size);

std::vector<void\*> list\_blocks;

std::vector<void\*> MKC\_blocks;

std::cout << "Comparing ListAllocator and MacKuseyCarelsAllocator" << std::endl;

std::cout << "Block allocation rate" << std::endl;

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

for (size\_t i = 0; i != 100000; ++i) {

void\* block = list\_alloc.alloc(i % 50 + 10);

list\_blocks.push\_back(block);

}

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

std::cout << "Time of alloc ListAllocator: " <<

std::chrono::duration\_cast<std::chrono::milliseconds>(end\_time - start\_time).count() <<

" milliseconds" << std::endl;

start\_time = std::chrono::steady\_clock::now();

for (size\_t i = 0; i != 100000; ++i) {

void\* block = MKC\_alloc.alloc(i % 50 + 10);

MKC\_blocks.push\_back(block);

}

end\_time = std::chrono::steady\_clock::now();

std::cout << "Time of alloc MacKuseyCarelsAllocator: " <<

std::chrono::duration\_cast<std::chrono::milliseconds>(end\_time - start\_time).count() <<

" milliseconds" << std::endl;

std::cout << "Block free rate" << std::endl;

start\_time = std::chrono::steady\_clock::now();

for (size\_t i = 0; i != list\_blocks.size(); ++i) {

list\_alloc.free(list\_blocks[i]);

if (i < 20) {

std::cout << list\_blocks[i] << std::endl;

}

}

end\_time = std::chrono::steady\_clock::now();

std::cout << "Time of free ListAllocator: " <<

std::chrono::duration\_cast<std::chrono::milliseconds>(end\_time - start\_time).count() <<

" milliseconds" << std::endl;

start\_time = std::chrono::steady\_clock::now();

for (size\_t i = 0; i != MKC\_blocks.size(); ++i) {

MKC\_alloc.free(MKC\_blocks[i]);

if (i < 20) {

std::cout << MKC\_blocks[i] << std::endl;

}

}

end\_time = std::chrono::steady\_clock::now();

std::cout << "Time of free MacKuseyCarelsAllocator: " <<

std::chrono::duration\_cast<std::chrono::milliseconds>(end\_time - start\_time).count() <<

" milliseconds" << std::endl;

}  
   
 **Тесты**

void\* block1 = list\_alloc.alloc(10000);

void\* block2 = list\_alloc.alloc(10);

void\* block3 = list\_alloc.alloc(1);

void\* block4 = MKC\_alloc.alloc(10000);

void\* block5 = MKC\_alloc.alloc(10);

void\* block6 = MKC\_alloc.alloc(1);

**вывод**

Block 1: 0x5647a2c5e020

Block 2: 0x5647a2c60740

Block 3: 0x5647a2c6075a

Block 4: 0x7f459fc04000

Block 5: 0x7f459fbc0000

Block 6: 0x7f459fbbf000

Comparing ListAllocator and MacKuseyCarelsAllocator

Block allocation rate

Time of alloc ListAllocator: 10 milliseconds

Time of alloc MacKuseyCarelsAllocator: 523 milliseconds

Block free rate

0x5647a2c68020

0x5647a2c6803a

0x5647a2c68055

0x5647a2c68071

0x5647a2c6808e

0x5647a2c680ac

0x5647a2c680cb

0x5647a2c680eb

0x5647a2c6810c

0x5647a2c6812e

0x5647a2c68151

0x5647a2c68175

0x5647a2c6819a

0x5647a2c681c0

0x5647a2c681e7

0x5647a2c6820f

0x5647a2c68238

0x5647a2c68262

0x5647a2c6828d

0x5647a2c682b9

Time of free ListAllocator: 1 milliseconds

0x7f459fbbe000

0x7f459fbbd000

0x7f459fbbc000

0x7f459fbbb000

0x7f459fbba000

0x7f459faad000

0x7f459faac000

0x7f459faab000

0x7f459faaa000

0x7f459faa9000

0x7f459faa8000

0x7f459faa7000

0x7f459faa6000

0x7f459faa5000

0x7f459faa4000

0x7f459faa3000

0x7f459faa2000

0x7f459faa1000

0x7f459faa0000

0x7f459fa9f000

Time of free MacKuseyCarelsAllocator: 2 milliseconds

-------------------

void\* block1 = list\_alloc.alloc(10000);

void\* block2 = list\_alloc.alloc(10);

void\* block3 = list\_alloc.alloc(44651047871);

void\* block4 = MKC\_alloc.alloc(1024);

void\* block5 = MKC\_alloc.alloc(2049);

void\* block6 = MKC\_alloc.alloc(144420166);

**Вывод**

Block 1: 0x55d7b0979020

Block 2: 0x55d7b097b740

Block 3: (nil)

Block 4: 0x7f567b74e000

Block 5: 0x7f567b70a000

Block 6: (nil)

Comparing ListAllocator and MacKuseyCarelsAllocator

Block allocation rate

Time of alloc ListAllocator: 12 milliseconds

Time of alloc MacKuseyCarelsAllocator: 515 milliseconds

**Вывод**

Проделав работу, я стало понятно, что алгоритм на списке быстрее работает, если количество блоков небольшое, поскольку дерево поддерживать и проходить по элементам гараздо дольше, однако на большом количество аллокаций, время алгоритмов почти одинаковое.