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**

**«Аллокаторы памяти»**

Студент: Меркулов Фёдор Алексеевич

Группа: М8О-207Б-21

Вариант: 14

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2022

**Содержание**

1. Репозиторий
2. Постановка задачи
3. Подробное описание каждого из исследуемых алгоритмов
4. Процесс тестирования и обоснование процесса тестирования
5. Исходный код
6. Результаты тестирования
7. Заключение по проведённой работе

**Репозиторий**

<https://github.com/WhatTheMUCK/OSi/tree/main/OSKP>

**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

Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям free и malloc. Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра. Необходимо самостоятельно разработать стратегию тестирования для определения ключевых характеристик аллокаторов памяти. При тестировании нужно свести к минимуму потери точности из-за накладных расходов при измерении ключевых характеристик, описанных выше.

В отчете необходимо отобразить следующее:

* Подробное описание каждого из исследуемых алгоритмов
* Процесс тестирования
* Обоснование подхода тестирования
* Результаты тестирования
* Заключение по проведенной работе

**Вариант 14:**

Необходимо сравнить два алгоритма аллокации: списки свободных блоков (первое подходящее) и алгоритм Мак-Кьюзи-Кэрелса

**Подробное описание каждого из исследуемых алгоритмов**

**Первый подходящий участок:**

*Карта ресурсов* (resource map) - это набор пар <base, size>(<базовый адрес, размер>), используемый для отслеживания свободных областей памяти. Изначально область памяти описывается при помощи единственного вхождения карты, в котором указатель равен стартовому адресу области, а размер равен её общему объёму памяти. После этого клиенты начинают запрашивать и освобождать участки памяти, вследствие чего область становится фрагментированной. Ядро создаёт для каждого нового последовательного свободного участка памяти новое вхождение карты. Элементы карты сортиру.тся в порядке возрастания адресов, что упрощает задачу слияния свободных участков.

При помощи карты ресурсов ядро может выполнить запрос на выделение памяти:

**Первый подходящий участок.** Выделение памяти из первой по счёту свободной области, имеющей достаточный для удовлетворения запроса объём. Это самый быстрый алгоритм из всех трёх (Первый подходящий участок, Наиболее походящий участок, Наименее подходящий участок), но он не совсем оптимален применительно к соображениям уменьшения фрагментации.

Карта ресурсов применяется для простейшего случая распределения памяти. Она обладает несколькими преимуществами:

* алгоритм прост для практичесой реализации
* карта ресурсов не ограничена в применении только для задач выделения и освобождения памяти. Она может быть использована и для обработки наборов различных других объектов, расположенных в определённом порядке и доступных для выделения и освобождения непрерывними участками (к таким объектам относятся, к примеру, вхождения таблицы страниц и семафоры);
* карта позволяет выделять точное количество байтов, равное запрошенному, без потерь памяти. На практике распределитель памяти всегда округляет количество выделяемой памяти до числа, делящегося на четыре или восемь, с точки зрения простоты и удобства выравнивания;
* от клиента не требуется всегда возвращать участок памяти, равный запрошенному. Как показывает предыдущий пример, клиент может освободить любую часть выделенного ранее участка, при этом распределитель памяти корректно отработает возникшую ситуацию. Такая возможность стала доступна потому, что в качестве аргумента процедуры rmfree() указывается размер освобождаемого участка, а учётная информация (то есть карта ресурсов) поддерживается системой отдельно от самой выделяемой памяти;
* распределитель соединяет последовательные участки памяти в один, что даёт возможность выделять в дальнейшем области памяти различнйо длины.

Распределитель ресурсов имеет и ряд существенных недостатков:

* по истечении какого-то времени работы карта становится сильно фрагментированной. В ней оказывается большое количество участков малого размера. Это приводит к низкой востребованности ресурса. В частности, распределитель карты ресурсов плохо справляется с задачей обслуживания "больших" запросов;
* по мере увеличения фрагментации синхронно наращивается и сама карта ресурсов, так как для размещения данных о каждом новом свободном участке требуется новое вхождение. Если карта настроена на фиксированное количество вхождений, то в некоторый момент времени она может переполниться, а распределитель памяти потерять данные о какой-то доле свбодных участков;
* если карта будет расти динамически, то для её вхождений потребуется собственный распределитель. Эта проблема является "рекурсивной"
* для решения задачи объединения свободных смежных областей памяти распределитель должен поддерживать карту, упорядоченную в порядке увеличения смещения от базового адреса. Операция сортировки весбма затратна, более того, она должна производиться по месту в том случае, если карта реализована в виде массива фиксированного размера. Нагрузка на систему, возникающая при сортировке, является весьма ощутимой даже в том случае, если карта размещается в памяти динамически и организована в виде связанного списка;
* часто требуется выполнять операцию последовательного поиска в карте с целью обнаружения достаточно большого для удовлетворения запроса участка. Эта процедура занимает много времени и выполняется медленнее при сильной фрагментации памяти;
* несмотря на наличие возможности возврата свободных участков памяти в хвост пула страничной подсистемы, алгоритм выделения и освобождения памяти не приспособлен для такой операции. На практике распределитель никогда не стремится достичь большей непрерывности ввереных ему областей.

**Алгоритм Мак-Кьюзи-Кэрелса:**

Маршалл Кирк Мак-Кьюзик и Майкл Дж. Кэрелс разработали усовершенствованный метол выделения памяти, который был реализован во многих вариантах системы UNIX, в том числе 4.4BSD и Digital UNIX. Методика позволяет избавиться от потерь в тех случаях, когда размер запрашиваемого участка памяти равен некоторой степени двойки. В нём также была произведена оптимизация перебора в цикле. Такие действия теперь нужно производить только в том случае, если на момент компиляции неизвестен размер выделенного участка.

Алгоритм подразумевает, что память разбита на набор последовательных страниц, и все буферы, относящиеся к одной странице, должны иметь одинаковый размер (являющийся некоторой степенью числа 2). Для управления страницами распределитель использует дополнительный массив memory\_size\_mkk[].

Каждая страница может находиться в одном из трёх перечисленных состояний.

* Быть свободной. В этом случае соответствующий элемент массива memory\_size\_mkk[] содержит указатель на элемент, описывающий следующую свободную страницу.
* Быть разбитой на буферы определённого размера. Элемент массива содержит размер буфера.
* Являться частью буфера, объединяющего сразу несколько страниц. Элемент массива указывает на первую страницу буфера, в которой находятся данные о его длине.

Массив list\_mkk содержит заголовки всех буферов, имеющих размер меньше одной страницы (размер страницы также некоторая степень двйоки).

Так как длина всех буферов одной страницы одинакова, нет нужды хранить в заголовках выделенных буферов указатель на список свободных буферов. Процедура free() находит страницу путём маскирования младщих разрядов адреса буфера и обнаружения размера буфера в соответствующем элементе массива memory\_size\_mkk[]. Отсутствие заголовка в выделенных буферах позволяет экономить память при удовлетворении запросов с потребностью в памяти, кратной некоторой степени числа 2.

Вызов процедуры malloc() заменён макроопределением, которое производит округления значения длины запрашиваемого участка вверх до достижения числа, являющегося степенью двойки (при этом не нужно прибавлять какие-либо дополнительные байты на заголовок) и удаляет буфер из соответствующего списка свободных буферов. Макрос вызывает функцию malloc() для запроса одной или нескольких страниц тогда, когда список свободных буферов необходимого размера пуст. В этом случае malloc() вызывает процедуру, которая берёт свободную страницу и разделяет её на буферы необходимого размера. Здесь цикл заменён на схему вычислений по условию.

Приведённый алгоритм значительно улучшает методику распределения памяти на основе степени числа 2. Он работает намного быстрее, потери памяти при его применении сильно сокращаются. Алгоритм позволяет эффективно обрабатывать запросы на выделения как малых, так и больших участков памяти. Однако описанная методика обладает и некоторыми недостатками, связанными с необходимостью использования участков равных некоторой степени числа 2. Не существует какого-либо способа перемещения участков из одного списка в другой. Это делает распределитель не совсем подходящим средством при неравномерном использовании памяти, например, если системе необходимо много буферов малого размера на короткий промежуток времени. Технология также не даёт возможности возвращать участки памяти, запрошенные ранее у страничной системы.

**Процесс тестирования и обоснование процесса тестирования:**

В запросы: requests, состоящие из адресов и размеров запросов псевдослучайным образом помещаются значения от 1 до MAX\_BYTES (в размер запросов), причём количеством запросов является NUMBER\_REQUESTS. Также создаётся массив permute из NUMBER\_REQUESTS индексов запросов requests, причём эти индексы псевдослучайным образом перемешиваются. Затем мы пытаемся выделить место для каждого запроса и как как раз таки освобождаем запросы псевдослучайным образом (В цикле от 1 до NUMBER\_REQUESTS освобождается requests[permute[i]], а не requests[i]). Таким образом, при тестировании сведены к минимуму потери точности из-за накладных расходов при измерении ключевых характеристик: скорости выделения, освобождения памяти и фактора использования.

**Исходный код**

**main.cpp**

#include <iomanip>

#include <iostream>

#include <time.h>

#include "allocator\_list.h"

#include "allocator\_mkk.h"

using namespace std;

//Запросы содержат адрес и размер запрашиваемого места

typedef struct request\_structure {

void\* address;

size\_t bytes;

} request;

//Нужно для переработки char\* в число

size\_t parse\_size(const char\* string) {

size\_t size = 0;

while(\*string != '\0') {

if(\*string < '0' || \*string > '9') {

return 0;

}

size = size \* 10 + \*string - '0';

++string;

}

return size;

}

int main(int argument\_count, char\* argument\_vector[]) {

const size\_t NUMBER\_REQUESTS = 1000;

const size\_t MAX\_BYTES = 5000;

clock\_t first\_time;

clock\_t second\_time;

size\_t first\_index; //size\_t - беззнаковый тип, созданспециально для хранения размера оюъектов любых типов

size\_t second\_index;

size\_t third\_index;

size\_t argument;

size\_t query = 0;

size\_t total = 0;

size\_t\* permute = (size\_t\*)malloc(sizeof(size\_t) \* NUMBER\_REQUESTS);

request\* requests = (request\*)malloc(sizeof(request) \* NUMBER\_REQUESTS);

srand((unsigned int)time(0)); //Выполняем инициализацию генератора случайных чисел rand

if(argument\_count < 2) {

cout << "Usage: " << argument\_vector[0] << " <SIZE>\n";

return 0;

}

argument = parse\_size(argument\_vector[1]);

//Инициализация алгоритма аллокации списка свободных блоков(первого подходящего) argument битами

if(!initialization\_list(argument)) {

cout << "Error. No memory\n";

return 0;

}

//Инициализация алгоритма аллокации Мак-Кьюзи-ККэрэлса argument битами

if(!initialization\_mkk(argument)) {

cout << "Error. No memory\n";

return 0;

}

for(first\_index = 0; first\_index < NUMBER\_REQUESTS; ++first\_index) {

requests[first\_index].bytes = 1 + rand() % MAX\_BYTES; //Псевдослучайным образом определяем размеры запросов

permute[first\_index] = first\_index; //permute хранит индексы массива request и нужен будет для того чтобы доставать псевдослучайный запрос

}

for(first\_index = 0; first\_index < NUMBER\_REQUESTS; ++first\_index) {

second\_index = rand() % NUMBER\_REQUESTS;

third\_index = rand() % NUMBER\_REQUESTS;

argument = permute[second\_index];

permute[second\_index] = permute[third\_index];

permute[third\_index] = argument;

//Поменяли местами permute[second\_index] и permute[third\_index]

}

cout << "Alloc requests: " << NUMBER\_REQUESTS;

//Требуется произвести NUMBER\_REQUESTS аллокаций памяти

cout << "\nBytes: 1 to " << MAX\_BYTES;

//Необходимое место: от 1 до MAX\_BYTES

cout << "\n\nAllocator LIST:\n";

//Характеристики алгоритма аллокации: список свободных блоков(первое подходящее)

//Замеряем скорость выделения блоков

first\_time = clock();

for(first\_index = 0; first\_index < NUMBER\_REQUESTS; ++first\_index) {

requests[first\_index].address = malloc\_list(requests[first\_index].bytes);

//Для каждого запроса пытаемся найти свободный блок размером >= размера запроса (request[first\_index].bytes)

}

second\_time = clock();

//Заканчиваем замерять скорость выделения блоков

//И параллельно начинаем замерять скорость освобождения блоков

printf("Alloc time: %lf\n", (double)(second\_time - first\_time) / CLOCKS\_PER\_SEC);

//Разница между концом и началом замеров скорости выделения блоков и есть скорость выделения блоков (/CLOCKS\_PER\_SEC нужен так как необходимо миллисекунды перевести в секунжы)

query = get\_request\_list(); //Счётчик количества запрашиваемого места в общем

total = get\_total\_list(); //Счётчик количества используемого места в общем

for(first\_index = 0; first\_index < NUMBER\_REQUESTS; ++first\_index) {

if(requests[permute[first\_index]].address == NULL) {

continue;

}

//Вот и раскрывается сакральный смысл permute (нужен для того чтобы более точно и качественно определить скорость освобождения блоков)

list\_free(requests[permute[first\_index]].address);

//Для каждого запроса освобождаем блоки

}

first\_time = clock();

//Заканчиваем замерять скорость освобождения блоков

printf("Free time: %lf\n", (double)(first\_time - second\_time) / CLOCKS\_PER\_SEC);

//Разница между концом и началом замеров скорости освобождения блоков и есть скорость освобождения блоков (printf нужен для того чтобы число выводилось целиком, а не в естественной форме)

cout << "Usage factor: " << (long double)query / total << "\n\n";

//Фактор использования определяется отношением количества запрашиваемой памяти к количеству использованной

cout << "Allocator MKK\n";

//Характеристики алгоритма аллокации: Мак-Кьюзи-Кэрэлса

//Замеряем скорость выделения блоков

first\_time = clock();

for(first\_index = 0; first\_index < NUMBER\_REQUESTS; ++first\_index) {

requests[first\_index].address = malloc\_mkk(requests[first\_index].bytes);

//Для каждого запроса пытаемся либо выбрать на странице буфер размером 2^n (n - некоторое целое число, такое, что 2^n ближайшая степень 2 (>=) к requests[first\_index].bytes)

//Либо выделить некоторое количество страниц (у которых размер тоже 2^m, где m - некоторое целое число (>=0)) и на последней странице также выбрать буфер

}

second\_time = clock();

//Заканчиваем замерять скорость выделения блоков

//И параллельно начинаем замерять скорость освобождения блоков

printf("Alloc time: %lf\n", (double)(second\_time - first\_time) / CLOCKS\_PER\_SEC);

//Разница между концом и началом замеров скорости освобождения блоков и есть скорость освобождения блоков (printf нужен для того чтобы число выводилось целиком, а не в естественной форме)

query = get\_request\_mkk(); //Счётчик количества запрашиваемого места в общем

total = get\_total\_mkk(); //Счётчик количества используемого места в общем

for(first\_index = 0; first\_index < NUMBER\_REQUESTS; ++first\_index) {

if(requests[permute[first\_index]].address == NULL) {

continue;

}

free\_mkk(requests[permute[first\_index]].address);

//Для каждого запроса освобождаем блоки

}

first\_time = clock();

printf("Free time: %lf\n", (double)(first\_time - second\_time) / CLOCKS\_PER\_SEC);

//Разница между концом и началом замеров скорости освобождения блоков и есть скорость освобождения блоков (printf нужен для того чтобы число выводилось целиком, а не в естественной форме)

cout << "Usage factor: " << (long double)query / total << "\n";

//Фактор использования определяется отношением количества запрашиваемой памяти к количеству использованной

destroy\_list();

destroy\_mkk();

free(requests);

free(permute);

return 0;

}

**allocator\_list.cpp**

#include "allocator\_list.h"

int initialization\_list(size\_t size) {

if(size < sizeof(block\_list)) {

size = sizeof(block\_list);

}

begin\_list = (block\_list\*)malloc(size);

if(begin\_list == NULL) {

return 0;

}

begin\_list->size = size;

begin\_list->previous = NULL;

begin\_list->next = NULL;

free\_list = begin\_list;

size\_list = size;

//Задаём изначальную карту ресурсов: состоящую из стартового адреса области (begin\_list) и общего размера памяти (size)

return 1;

}

void destroy\_list() {

free(begin\_list);

//Освобождаем стартовый адрес области тем самым удаляя аллокатор списка свободных элементов

}

void\* alloc\_block\_list(block\_list\* block, size\_t size) {

block\_list\* next\_block = NULL;

if(block->size >= size + sizeof(block\_list)) {

next\_block = (block\_list\*)((PBYTE\_LIST)block + size);

next\_block->size = block->size - size;

next\_block->previous = block->previous;

next\_block->next = block->next;

block->size = size;

if(block->previous != NULL) {

block->previous->next = next\_block;

}

if(block->next != NULL) {

block->next->previous = next\_block;

}

if(block == free\_list) {

free\_list = next\_block;

}

}

else {

if(block->previous != NULL) {

block->previous->next = block->next;

}

if(block->next != NULL) {

block->next->previous = block->previous;

}

if(block == free\_list) {

free\_list = block->next;

}

}

return (void\*)((PBYTE\_LIST)block + sizeof(size\_t));

}

void\* malloc\_list(size\_t size) {

size\_t first\_size = size\_list;

size\_t old\_size = size;

block\_list\* first\_block = free\_list;

block\_list\* current = free\_list;

size += sizeof(size\_t);

if(size < sizeof(block\_list)) {

size = sizeof(block\_list);

}

int flag = 0;

while(current != NULL && flag == 0) {

if (current->size < first\_size && current->size >= size) {

//Если размер рассматриваемого блока меньше всего свободного места и блок сможет вместить необходимое количество данных, то это значит что мы нашли первый подходящий свободный блок

first\_size = current->size;

first\_block = current;

flag = 1;

}

current = current->next;

}

if(free\_list == NULL || first\_block->size < size){

//Если свободное место отсутствует или размер свободного места меньше запрашиваемого размера, то мы не сможем выделить место для запроса

return NULL;

}

request\_list += old\_size; //Подсчёт запрашиваемого объёма места

total\_list += size; //Подсчёт в итоге используемого объёма места

return alloc\_block\_list(first\_block, size); //Фрагментируем найденный блок (чтобы потом можно было ещё использовать оставшееся место в нём)

}

void list\_free(void\* address) { //Освобождение места занимаемого запросом по адресу address

block\_list\* block = (block\_list\*)((PBYTE\_LIST)address - sizeof(size\_t)); //Находим адрес блока (так как информация находится после заголовка, который занимает sizeof(size\_t) бит)

block\_list\* current = free\_list; //Текущий блок от самого первого свободного блока пройдёт все для того чтобы добавить освобождённый блок в список свободных

block\_list\* left\_block = NULL;

block\_list\* right\_block = NULL;

while(current != NULL) {

if((block\_list\*)((PBYTE\_LIST)current + current->size) <= block) {

//Нахождение блока, который располагается левее освобождённого, но является самым близким к освобождённому

left\_block = current;

}

if((block\_list\*)((PBYTE\_LIST)block + block->size) <= current) {

//Нахождение блока, который находится правее и сразу выход из цикла, чтобы найти самый близкий правый блок к освобождённому

right\_block = current;

break;

}

current = current->next;

}

//Добавление освобождённого блока в двусвязный список свободных блоков

if(left\_block != NULL) {

left\_block->next = block;

}

else {

//Если блок самый левый, то он является самым первым свободным блоком

free\_list = block;

}

if(right\_block != NULL) {

right\_block->previous = block;

}

block->previous = left\_block;

block->next = right\_block;

current = free\_list;

//Объединение свободных блоков стоящих вплотную в единый блок большего размера

while(current != NULL) {

if ((block\_list\*)((PBYTE\_LIST)current + current->size) == current->next) {

current->size += current->next->size;

current->next = current->next->next;

if (current->next != NULL) {

current->next->previous = current;

}

continue;

}

current = current->next;

}

}

size\_t get\_request\_list() {

return request\_list; //Определение количества запрашиваемого объёма данных за всё время

}

size\_t get\_total\_list() {

return total\_list; //Определение количества используемых данных за всё время

}

**allocator\_list.h**

#ifndef ALLOCATOR\_LIST\_H

#define ALLOCATOR\_LIST\_H

#include <stdio.h>

#include <stdlib.h>

typedef unsigned char\* PBYTE\_LIST;

typedef struct block\_list {

size\_t size;

struct block\_list\* previous;

struct block\_list\* next;

} block\_list; //Двусвязный список свободных блоков

static block\_list\* begin\_list; //Стартовый адрес области

static block\_list\* free\_list; //Адрес первого свободного блока

static size\_t size\_list; //Общий размер выделенной памяти

static size\_t request\_list = 0; //Счётчик количества запрашиваемой информации

static size\_t total\_list = 0; //Счётчик количества используемой информации

int initialization\_list(size\_t size);

void destroy\_list();

void\* alloc\_block\_list(block\_list\* block, size\_t size);

void\* malloc\_list(size\_t size);

void list\_free(void\* address);

size\_t get\_request\_list();

size\_t get\_total\_list();

#endif

**allocator\_mkk.cpp**

#include "allocator\_mkk.h"

int initialization\_mkk(size\_t size) {

size\_t index;

block\_mkk\* block = NULL;

pages\_mkk = get\_pages\_count\_mkk(size); //Определяем общее количество страниц (плгоритм подразумевает, что память разбита на набор последовательных страниц)

pow\_mkk = pow\_of\_size\_mkk(PAGE\_SIZE\_MKK); //Определяем степень двйоки, описывающей размер 1 страницы

pow\_index\_minimum = pow\_of\_size\_mkk(sizeof(block\_mkk)); //Определяем степень двойки, описывающей размер структуры block\_mkk

heap\_mkk = malloc(pages\_mkk \* PAGE\_SIZE\_MKK); //Стартовый адрес области страниц

memory\_size\_mkk = (size\_t\*)malloc(sizeof(size\_t) \* pages\_mkk); //Массив для управления страницами

list\_mkk = (block\_mkk\*\*)malloc(sizeof(block\_mkk\*) \* pow\_mkk); //Массив, содержащий заголовки всех буферов, имеющих размер меньше одной страницы

if(heap\_mkk == NULL || memory\_size\_mkk == NULL || list\_mkk == NULL) {

return 0;

}

memory\_size\_mkk[free\_state] = free\_state; //0 страница пока свободна

list\_mkk[free\_state] = (block\_mkk\*)heap\_mkk;

block = list\_mkk[free\_state];

for(index = 1; index < pages\_mkk; ++index) {

memory\_size\_mkk[index] = free\_state; //все страницы от 1 до количества страниц считаются свободными

block->next = (block\_mkk\*)((PBYTE\_MKK)block + PAGE\_SIZE\_MKK);

block = block->next;

}

block->next = NULL;

for(index = 1; index < pow\_mkk; ++index) {

list\_mkk[index] = NULL; //Пока нет буферов => нет буферов, имеющих размер меньше одной страницы

}

return 1;

}

void destroy\_mkk() {

free(heap\_mkk);

//Освобождаем стартовый адрес области

free(memory\_size\_mkk);

//Освобождаем массив для управления страницами

free(list\_mkk);

//Освобождаем массив, содержащий заголовки буферов,

//Тем самым удаляя алгоритм аллокации Мак-Кьюзи-Кэрэлса

}

void\* malloc\_mkk(size\_t size) {

size\_t pow\_index = pow\_of\_size\_mkk(size); //Округляем вверх size до степени двойки

size\_t old\_size = size;

block\_mkk\* block = NULL;

if(pow\_index < pow\_index\_minimum) {

//Если размер меньше минимального, то просто делаем его минимальным

pow\_index = pow\_index\_minimum;

}

size = 1 << pow\_index;

//Определяем размер равный степени двойки, который вмести в себя old\_size

if(size < PAGE\_SIZE\_MKK) {

//Если размер меньше страницы, то мы сможем работать только с одной страницей

if(list\_mkk[pow\_index] == NULL) {

//Если нет ни единого буфера размером в 2^pow\_index, то выделим страницу для этого

block = alloc\_page\_mkk(size);

if(block == NULL) {

return NULL;

}

split\_page\_mkk(block, pow\_index); //Разделим страницу которую мы выделили под буфер размером 2^pow\_index, на максимальное количество буферов размером 2^pow\_index

}

block = list\_mkk[pow\_index]; //block берёт первый свободный буффер размером 2^pow\_index

list\_mkk[pow\_index] = block->next; //list\_mkk[pow\_index] начинает хранить следующий свободный буффер, так как предыдущий стал занят

request\_mkk += old\_size; //Подсчёт запрашиваемого количества данных

total\_mkk += size; //Подсчёт используемого количества данных

return (void\*)block;

}

else {

request\_mkk += old\_size; //Подсчёт запрашиваемого количества данных

total\_mkk += size; //Подсчёт используемого количества данных

return alloc\_page\_mkk(size);

}

}

void free\_mkk(void\* address) {

size\_t page\_index = get\_page\_index\_mkk((block\_mkk\*)address); //Определяем номер страницы, на которой располагается буффер, который мы хотим освободить

size\_t pow\_index = pow\_of\_size\_mkk(memory\_size\_mkk[page\_index]); //Определяем степень 2 этого буффера

block\_mkk\* block = (block\_mkk\*)address;

if(memory\_size\_mkk[page\_index] < PAGE\_SIZE\_MKK) { //Если буффер размером меньше страницы

block->next = list\_mkk[pow\_index]; //Первым свободным буффером размером 2^pow\_index становится block

list\_mkk[pow\_index] = block;

}

else { //Если буффер размером больше страницы

free\_page\_mkk(block);

}

}

block\_mkk\* alloc\_page\_mkk(size\_t size) {

size\_t count = 0;

size\_t page\_index = 0;

size\_t previous\_index = get\_page\_index\_mkk(list\_mkk[free\_state]); //Определяем номер страницы с которой есть свободное место

size\_t pages = get\_pages\_count\_mkk(size); //Определяем количество страниц которое понадобится для того чтобы вместить size бит

block\_mkk\* current = list\_mkk[free\_state]; //Определяем текущий блок для дальнейшей работы с ним

block\_mkk\* previous = NULL;

block\_mkk\* page = NULL;

while(current != NULL) {

page\_index = get\_page\_index\_mkk(current); //Определяем номер страницы, на котором находится буффер current

if(page\_index - previous\_index <= 1) {

//Если current занимает не больше одной страницы, то

if(page == NULL) {

//Мы нашли что будет располагаться на странице page

page = current;

}

++count;

}

else {

//Если current занимает больше 1 страницы, то мы сбрасываем счётчик до 1, а в страницу добавляем current, чтобы при повторном прохождении цикла не зайти в (if(page == NULL))

page = current;

count = 1;

}

if(count == pages) {

//Если счётчик сравнялся с необходимым количеством страниц, то мы завершаем цикл

break;

}

//Берём следующий буфер, при этом не забывая о предыдущем

previous = current;

current = current->next;

previous\_index = page\_index;

}

if(count < pages) {

//Если счётчик оказался меньше необходимого количества страниц, то значит что на одну страницу буффер не вместится

page = NULL;

}

if(page != NULL) {

//Если мы смогли что-то занести в страницу

page\_index = get\_page\_index\_mkk(page); //Определяем номер страницы

memory\_size\_mkk[page\_index] = size; //Страница под номером page\_index разделена на буферы размером size (это уже некоторая степень 2)

current = (block\_mkk\*)((PBYTE\_MKK)page + (pages - 1) \* PAGE\_SIZE\_MKK); //Адрес текущего блока

if (previous != NULL) {

previous->next = current->next;

}

else {

list\_mkk[free\_state] = current->next; //Вносим адрес свободного блока в list\_mkk[free\_state]

}

}

return page;

}

void free\_page\_mkk(block\_mkk\* block) {

size\_t index;

size\_t page\_index = get\_page\_index\_mkk(block); //Определяем номер страницы

size\_t block\_count = memory\_size\_mkk[page\_index] / PAGE\_SIZE\_MKK; //Определяем сколько страниц целиком заняты буффером block

block\_mkk\* left = NULL; //Левый буффер

block\_mkk\* right = NULL; //Правый буффер

block\_mkk\* current = block;

while(current != NULL) {

if (current < block) { //Определение самого близкого к block левого буффера

left = current;

}

else {

if(current > block) { //Определение самого близкого к block правого буффера

right = current;

break;

}

}

current = current->next;

}

for(index = 1; index < block\_count; ++index) {

block->next = (block\_mkk\*)((PBYTE\_MKK)block + PAGE\_SIZE\_MKK); //Отделяем от буффера цельные страницы

block = block->next;

}

block->next = right;

if(left != NULL) {

left->next = block; //Переопределяем связь левого с block, тем самым освободив все цельные страницы

}

else {

list\_mkk[free\_state] = block; //Утверждаем, что block свободен

}

}

void split\_page\_mkk(block\_mkk\* block, size\_t pow\_index) {

size\_t index;

size\_t page\_index = get\_page\_index\_mkk(block); //Определяемномер страницы

size\_t block\_size = 1 << pow\_index; //Определяем размер блоков как 2^pow\_index

size\_t block\_count = PAGE\_SIZE\_MKK / block\_size; //Количество блоков есть отношение общего размера страницы к размеру 1 блока

list\_mkk[pow\_index] = block; //Сохраняем буфер block, как буффер размером 2^pow\_index

memory\_size\_mkk[page\_index] = block\_size; //На странице page\_index хранится размер буфферов, на которые она поделена

for(index = 1; index < block\_count; ++index) {

block->next = (block\_mkk\*)((PBYTE\_MKK)block + block\_size); //Определяем связь буфферов на странице

block = block->next;

}

block->next = NULL;

}

size\_t pow\_of\_size\_mkk(size\_t size) { //Округление вверх до степени 2

size\_t pow = 0;

while(size > ((size\_t)1 << pow)) {

++pow;

}

return pow;

}

size\_t get\_pages\_count\_mkk(size\_t size) { //Определение необходимого количества страниц

return size / PAGE\_SIZE\_MKK + (size\_t)(size % PAGE\_SIZE\_MKK != 0);

}

size\_t get\_page\_index\_mkk(block\_mkk\* block) { //Определение номера страницы, на которой находится block

return (size\_t)((PBYTE\_MKK)block - (PBYTE\_MKK)heap\_mkk) / PAGE\_SIZE\_MKK;

}

size\_t get\_request\_mkk() { //Опредение запрашиваемого за всё время объёма памяти

return request\_mkk;

}

size\_t get\_total\_mkk() { //Определение использованного за всё время объёма памяти

return total\_mkk;

}

**allocator\_mkk.h**

#ifndef ALLOCATOR\_MKK\_H

#define ALLOCATOR\_MKK\_H

#include <stdio.h>

#include <stdlib.h>

typedef unsigned char\* PBYTE\_MKK;

typedef enum memory\_structure {

free\_state = 0

} memory\_state;

typedef struct block\_mkk\_structure {

struct block\_mkk\_structure\* next;

} block\_mkk; //Односвязный список блоков

static const size\_t PAGE\_SIZE\_MKK = 4096; //Размер одной страницы (обязательно некоторая степень 2)

static void\* heap\_mkk = NULL; //Стартовый адрес области

static size\_t\* memory\_size\_mkk = NULL; //Массив для управления страницами

//Каждая из страниц может находиться в одном из 3 состояний:

//Быть свободной => соответствующий элемент массива содержит указатель на элемент, описывающий следующую свободную страницу

//Быть разбитой на буферы определённого размера (некоторая степень 2). Элемент массива содержит размер буфера

//Являться частью буфера, объединяющего сразу несколько страниц. Элемент массива указывает на первую страницу буфера, в котором находяться данные о его длине

static block\_mkk\*\* list\_mkk = NULL; //Массив, содержащий заголовки всех буферов, имеющих размер меньше одной страницы

static size\_t pages\_mkk = 0; //Общее количество страниц

static size\_t pow\_mkk = 0; //Степень n двойки: 2^n = PAGE\_SIZE\_MKK

static size\_t pow\_index\_minimum = 0; //Минимальный размер необходимый для хранения указателя на элемент

static size\_t request\_mkk = 0; //Счётчик количества запрашиваемой информации

static size\_t total\_mkk = 0; //Счётчик количества использованной информации

int initialization\_mkk(size\_t size);

void destroy\_mkk();

void\* malloc\_mkk(size\_t size);

void free\_mkk(void\* address);

block\_mkk\* alloc\_page\_mkk(size\_t size);

void free\_page\_mkk(block\_mkk\* block);

void split\_page\_mkk(block\_mkk\* block, size\_t powIndex);

size\_t pow\_of\_size\_mkk(size\_t size);

size\_t get\_pages\_count\_mkk(size\_t size);

size\_t get\_page\_index\_mkk(block\_mkk\* block);

size\_t get\_request\_mkk();

size\_t get\_total\_mkk();

#endif

**Результаты тестирования**

papik@papik-VirtualBox:~/OSKP/build$ ./main

Usage: ./main <SIZE>

papik@papik-VirtualBox:~/OSKP/build$ ./main 10

Alloc requests: 1000

Bytes: 1 to 5000

Allocator LIST:

Alloc time: 0.000011

Free time: 0.000063

Usage factor: 0.458333

Allocator MKK

Alloc time: 0.000048

Free time: 0.000041

Usage factor: 0.659859

papik@papik-VirtualBox:~/OSKP/build$ ./main 10

Alloc requests: 1000

Bytes: 1 to 5000

Allocator LIST:

Alloc time: 0.000022

Free time: 0.000069

Usage factor: 0.666667

Allocator MKK

Alloc time: 0.000174

Free time: 0.000054

Usage factor: 0.656526

papik@papik-VirtualBox:~/OSKP/build$ ./main 10

Alloc requests: 1000

Bytes: 1 to 5000

Allocator LIST:

Alloc time: 0.000022

Free time: 0.000052

Usage factor: 0.0416667

Allocator MKK

Alloc time: 0.000137

Free time: 0.000047

Usage factor: 0.648232

papik@papik-VirtualBox:~/OSKP/build$ ./main 100

Alloc requests: 1000

Bytes: 1 to 5000

Allocator LIST:

Alloc time: 0.000022

Free time: 0.000051

Usage factor: 0.9

Allocator MKK

Alloc time: 0.000133

Free time: 0.000038

Usage factor: 0.660055

papik@papik-VirtualBox:~/OSKP/build$ ./main 100

Alloc requests: 1000

Bytes: 1 to 5000

Allocator LIST:

Alloc time: 0.000022

Free time: 0.000044

Usage factor: 0.802469

Allocator MKK

Alloc time: 0.000131

Free time: 0.000031

Usage factor: 0.662521

papik@papik-VirtualBox:~/OSKP/build$ ./main 100

Alloc requests: 1000

Bytes: 1 to 5000

Allocator LIST:

Alloc time: 0.000028

Free time: 0.000052

Usage factor: 0.888889

Allocator MKK

Alloc time: 0.000180

Free time: 0.000040

Usage factor: 0.658159

papik@papik-VirtualBox:~/OSKP/build$ ./main 1000

Alloc requests: 1000

Bytes: 1 to 5000

Allocator LIST:

Alloc time: 0.000028

Free time: 0.000038

Usage factor: 0.959514

Allocator MKK

Alloc time: 0.000132

Free time: 0.000037

Usage factor: 0.664425

papik@papik-VirtualBox:~/OSKP/build$ ./main 1000

Alloc requests: 1000

Bytes: 1 to 5000

Allocator LIST:

Alloc time: 0.000027

Free time: 0.000040

Usage factor: 0.90081

Allocator MKK

Alloc time: 0.000133

Free time: 0.000036

Usage factor: 0.662178

papik@papik-VirtualBox:~/OSKP/build$ ./main 1000

Alloc requests: 1000

Bytes: 1 to 5000

Allocator LIST:

Alloc time: 0.000025

Free time: 0.000046

Usage factor: 0.935872

Allocator MKK

Alloc time: 0.000134

Free time: 0.000027

Usage factor: 0.649837

papik@papik-VirtualBox:~/OSKP/build$ ./main 10000

Alloc requests: 1000

Bytes: 1 to 5000

Allocator LIST:

Alloc time: 0.000037

Free time: 0.000057

Usage factor: 0.994393

Allocator MKK

Alloc time: 0.000134

Free time: 0.000039

Usage factor: 0.654851

papik@papik-VirtualBox:~/OSKP/build$ ./main 10000

Alloc requests: 1000

Bytes: 1 to 5000

Allocator LIST:

Alloc time: 0.000037

Free time: 0.000054

Usage factor: 0.992796

Allocator MKK

Alloc time: 0.000169

Free time: 0.000045

Usage factor: 0.659171

papik@papik-VirtualBox:~/OSKP/build$ ./main 10000

Alloc requests: 1000

Bytes: 1 to 5000

Allocator LIST:

Alloc time: 0.000040

Free time: 0.000053

Usage factor: 0.993596

Allocator MKK

Alloc time: 0.000136

Free time: 0.000041

Usage factor: 0.662198

**Заключение по проведённой работе**

В ходе данного курсового проекта я приобрёл практические навыки в использовании знаний, полученных в течении курса, а также провёл исследование 2 аллокаторов памяти: список свободных блоков (первое подходящее) и алгоритм Мак-Кьюзи-Кэрелса