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: Старцев Иван Романовмч

Группа: М8О-201Б-21

Вариант: 20

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/IvanTvardovsky/OS-labs

**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 20:**

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

**Подробное описание каждого из исследуемых алгоритмов**

**Списки свободных блоков:**

*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 сортируются в порядке возрастания адресов, что упрощает задачу слияния свободных участков.

**Наиболее подходящий участок.** Выделение памяти из наиболее подходящей свободной области, имеющей достаточный для удовлетворения запроса объём. Это самый выгодный по памяти алгоритм из всех трёх (первый подходящий участок, наиболее походящий участок, наименее подходящий участок), но он не самый быстрый.

**Алгоритм Мак-Кьюзи-Кэрелса:**

Маршалл Кирк Мак-Кьюзик и Майкл Дж. Кэрлс разработали усовершенствованный метол выделения памяти, который был реализован во многих вариантах системы 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).

Каждая страница может находиться в одном из трёх перечисленных состояний.

* Быть свободной.
* Быть разбитой на буферы определённого размера.
* Являться частью буфера, объединяющего сразу несколько страниц.

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.

**Процесс тестирования и обоснование процесса тестирования:**

В запросы: requests, состоящие из адресов и размеров запросов псевдослучайным образом вмещаются значения от 1 до MAX\_BYTES, причём счетом запросов является NUMBER\_REQUESTS. еще создаётся массив permute из NUMBER\_REQUESTS индексов запросов requests, причём эти индексы псевдослучайным образом перемешиваются. Потом мы пробуем выделить место для каждого запроса и как раз освобождаем запросы псевдослучайным образом. 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 <stdio.h>

#include <iomanip>

#include <iostream>

#include <time.h>

#include "./include/allocator\_list.h"

#include "./include/allocator\_mkk.h"

using namespace std;

// запросы содержат адрес и размер запрашиваемого места

typedef struct request\_strucе {

void\* address;

size\_t bytes;

} request;

// char\* в число

size\_t ToSizeT(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 REQUEST\_QUANTITY = 1000;

const size\_t MAX\_BYTES = 5000;

clock\_t first\_time, second\_time;

size\_t first\_index, second\_index, third\_index;

size\_t argument;

size\_t query = 0;

size\_t total = 0;

size\_t\* permute = (size\_t\*)malloc(sizeof(size\_t) \* REQUEST\_QUANTITY);

request\* requests = (request\*)malloc(sizeof(request) \* REQUEST\_QUANTITY);

// генератор случайных чисел

srand((unsigned int)time(0));

argument = ToSizeT(argument\_vector[1]);

// аллокация списка свободных блоков

if (!InitializationList(argument)) {

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

return 0;

}

// аллокация Мак-Кьюзи-Кэрлса

if (!MKKInitialization(argument)) {

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

return 0;

}

// псевдослучайные запросы и массив для них

for (first\_index = 0; first\_index < REQUEST\_QUANTITY; ++first\_index) {

requests[first\_index].bytes = 1 + rand() % MAX\_BYTES;

permute[first\_index] = first\_index;

}

for (first\_index = 0; first\_index < REQUEST\_QUANTITY; ++first\_index) {

second\_index = rand() % REQUEST\_QUANTITY;

third\_index = rand() % REQUEST\_QUANTITY;

argument = permute[second\_index];

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

permute[third\_index] = argument;

}

cout << "Количество запросов: " << REQUEST\_QUANTITY << "\n";

cout << "Байт: " << MAX\_BYTES << "\n\n";

cout << "Аллокация списка свободных блоков:\n";

first\_time = clock();

// для каждого запроса ищем свободный блок

for (first\_index = 0; first\_index < REQUEST\_QUANTITY; ++first\_index) {

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

}

second\_time = clock();

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

// снова счетчики

query = ListOfRequests();

total = TotalList();

for (first\_index = 0; first\_index < REQUEST\_QUANTITY; ++first\_index) {

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

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

}

first\_time = clock();

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

cout << "КПД использованной и запрошенной памяти: " << (long double)query / total << "\n\n";

cout << "Аллокация Мак-Кьюзи-Кэрлса\n";

first\_time = clock();

for (first\_index = 0; first\_index < REQUEST\_QUANTITY; ++first\_index) {

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

}

second\_time = clock();

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

// снова счетчики

query = MKKRequest();

total = MKKTotal();

for (first\_index = 0; first\_index < REQUEST\_QUANTITY; ++first\_index) {

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

continue;

}

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

}

first\_time = clock();

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

cout << "КПД использования и запрошенной памяти: " << (long double)query / total << "\n";

Destroy();

MKKDestroy();

free(requests);

free(permute);

return 0;

}

**allocator\_list.h**

#ifndef ALLOCATOR\_LIST\_H

#define ALLOCATOR\_LIST\_H

#include <stdio.h>

#include <stdlib.h>

typedef unsigned char\* PBYTE\_LIST;

// Структура двусвязного списка для свободных блоков

typedef struct block\_list {

size\_t size;

struct block\_list\* previous;

struct block\_list\* next;

} block\_list;

//Стартовый адрес области

static block\_list\* begin\_list;

//Адрес первого свободного блока

static block\_list\* free\_list;

//Общий размер выделенной памяти

static size\_t size\_list;

//Счётчик количества запрашиваемой информации

static size\_t request\_list = 0;

//Счётчик количества используемой информации

static size\_t total\_list = 0;

int InitializationList(size\_t size);

void Destroy();

void\* BlockAlloc(block\_list\* block, size\_t size);

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

void FreeList(void\* address);

size\_t ListOfRequests();

size\_t TotalList();

#endif

**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;

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

// 1. свободная ->

// -> соответствующий элемент массива содержит указатель на элемент, описывающий следующую свободную страницу

// 2. разбитая на буферы определённого размера (некоторая степень 2) -> элемент массива содержит размер буфера

// 3. частью буфера, объединяющего сразу несколько страниц ->

// -> элемент массива указывает на первую страницу буфера, в котором находятся данные о его длине

// односвязный список для блоков

typedef struct block\_mkk\_structure {

struct block\_mkk\_structure\* next;

} block\_mkk;

// размер одной страницы

static const size\_t PAGE\_SIZE\_MKK = 4096;

// стартовый адрес области

static void\* heap\_mkk = NULL;

// массив для управления страницами

static size\_t\* memory\_size\_mkk = NULL;

//Массив, содержащий заголовки всех буферов, имеющих размер меньше одной страницы

static block\_mkk\*\* list\_mkk = NULL;

//Общее количество страниц

static size\_t pages\_mkk = 0;

// Степень двойки

static size\_t pow\_mkk = 0;

//Минимальный размер необходимый для хранения указателя на элемент

static size\_t pow\_index\_minimum = 0;

//Счётчик количества запрашиваемой информации

static size\_t request\_mkk = 0;

//Счётчик количества использованной информации

static size\_t total\_mkk = 0;

int MKKInitialization(size\_t size);

void MKKDestroy();

void\* MKKMalloc(size\_t size);

void MKKFree(void\* address);

block\_mkk\* MKKAllocPage(size\_t size);

void MKKFreePage(block\_mkk\* block);

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

size\_t PowOfTwo(size\_t size);

size\_t MKKPageCounter(size\_t size);

size\_t MKKPageIndex(block\_mkk\* block);

size\_t MKKRequest();

size\_t MKKTotal();

#endif

**allocator\_list.cpp**

#include "../include/allocator\_list.h"

int InitializationList(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;

return 1;

}

void Destroy() {

free(begin\_list);

}

void\* BlockAlloc(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) {

// если блок может вместить необходимую информацию, значит он подходит

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

first\_size = current->size;

first\_block = current;

flag = 1;

}

current = current->next;

}

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

// добавляем в счетчики

request\_list += old\_size;

total\_list += size;

// фрагментация найденного блока для использования оставшего места

return BlockAlloc(first\_block, size);

}

// освобождение места занимаемого запросом по адресу address

void FreeList(void\* address) {

// находим адрес блока

block\_list\* block = (block\_list\*)((PBYTE\_LIST)address - sizeof(size\_t));

block\_list\* current = free\_list;

block\_list\* left\_block = NULL;

block\_list\* right\_block = NULL;

while (current != NULL) {

// блок, который располагается левее освобождённого и самый близкий

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

// аналогично правый

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

right\_block = current;

break;

}

current = current->next;

}

// добавление освобождённого блока в двусвязный список свободных блоков

if (left\_block != NULL) left\_block->next = block;

else free\_list = block;

if (right\_block != NULL) right\_block->previous = block;

block->previous = left\_block;

block->next = right\_block;

current = free\_list;

// объединение двух блоков рядом в один 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;

}

}

// счетчики

size\_t ListOfRequests() {

return request\_list;

}

size\_t TotalList() {

return total\_list;

}

**allocator\_mkk.cpp**

#include "../include/allocator\_mkk.h"

int MKKInitialization(size\_t size) {

size\_t index;

block\_mkk\* block = NULL;

// память разбита на набор последовательных страниц

pages\_mkk = MKKPageCounter(size);

// степень двойки для размера одной страницы

pow\_mkk = PowOfTwo(PAGE\_SIZE\_MKK);

// степень двойки для структуры блока

pow\_index\_minimum = PowOfTwo(sizeof(block\_mkk));

// стартовый адрес области

heap\_mkk = malloc(pages\_mkk \* PAGE\_SIZE\_MKK);

// массив управления страницами

memory\_size\_mkk = (size\_t\*)malloc(sizeof(size\_t) \* pages\_mkk);

// заголовки буферов размер которых меньше одной страницы для объединения

list\_mkk = (block\_mkk\*\*)malloc(sizeof(block\_mkk\*) \* pow\_mkk);

if (heap\_mkk == NULL || memory\_size\_mkk == NULL || list\_mkk == NULL) return 0;

// определение первой страницы и остальных

memory\_size\_mkk[free\_state] = free\_state;

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;

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

block = block->next;

}

block->next = NULL;

// заголовков пока нет

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

list\_mkk[index] = NULL;

}

return 1;

}

void MKKDestroy() {

free(heap\_mkk);

free(memory\_size\_mkk);

free(list\_mkk);

}

void\* MKKMalloc(size\_t size) {

size\_t pow\_index = PowOfTwo(size);

size\_t old\_size = size;

block\_mkk\* block = NULL;

if (pow\_index < pow\_index\_minimum) pow\_index = pow\_index\_minimum;

size = 1 << pow\_index;

// если размер меньше размера страницы

if (size < PAGE\_SIZE\_MKK) {

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

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

block = MKKAllocPage(size);

if (block == NULL) return NULL;

//Разделим страницу которую мы выделили под буфер размером 2^pow\_index

// на максимальное количество буферов размером 2^pow\_index

MKKSplitPage(block, pow\_index);

}

// первый свободный буфер

block = list\_mkk[pow\_index];

list\_mkk[pow\_index] = block->next;

// снова счётчики

request\_mkk += old\_size;

total\_mkk += size;

return (void\*)block;

} else {

// снова счётчики

request\_mkk += old\_size;

total\_mkk += size;

return MKKAllocPage(size);

}

}

void MKKFree(void\* address) {

size\_t page\_index = MKKPageIndex((block\_mkk\*)address);

size\_t pow\_index = PowOfTwo(memory\_size\_mkk[page\_index]);

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

if (memory\_size\_mkk[page\_index] < PAGE\_SIZE\_MKK) {

block->next = list\_mkk[pow\_index];

list\_mkk[pow\_index] = block;

} else MKKFreePage(block);

}

block\_mkk\* MKKAllocPage(size\_t size) {

size\_t count = 0;

size\_t page\_index = 0;

// находим где есть свободное место

size\_t previous\_index = MKKPageIndex(list\_mkk[free\_state]);

size\_t pages = MKKPageCounter(size);

block\_mkk\* current = list\_mkk[free\_state];

block\_mkk\* previous = NULL;

block\_mkk\* page = NULL;

while (current != NULL) {

// определяем номер страницы, на котором находится буффер current

page\_index = MKKPageIndex(current);

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

if (page == NULL) 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) break;

// берем следующий буфер и обрабатываем текущий

previous = current;

current = current->next;

previous\_index = page\_index;

}

// не вместится на одну страницу

if (count < pages) page = NULL;

if (page != NULL) {

page\_index = MKKPageIndex(page);

// страница page\_index уже разделена на буферы размером с size

memory\_size\_mkk[page\_index] = size;

// адрес текущего блока

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

if (previous != NULL) previous->next = current->next;

// вносим адрес свободного блока

else list\_mkk[free\_state] = current->next;

}

return page;

}

void MKKFreePage(block\_mkk\* block) {

size\_t index;

size\_t page\_index = MKKPageIndex(block);

size\_t block\_count = memory\_size\_mkk[page\_index] / PAGE\_SIZE\_MKK;

block\_mkk\* left = NULL;

block\_mkk\* right = NULL;

block\_mkk\* current = block;

while (current != NULL) {

// самый близкий левый буффер

if (current < block) left = current;

// аналогично правый

else {

if (current > block) {

right = current;

break;

}

}

current = current->next;

}

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

// отделяем целые страницы от буфера

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

block = block->next;

}

block->next = right;

// освобождаем целые страницы

if (left != NULL) left->next = block;

else list\_mkk[free\_state] = block;

}

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

size\_t index;

size\_t page\_index = MKKPageIndex(block);

size\_t block\_size = 1 << pow\_index;

size\_t block\_count = PAGE\_SIZE\_MKK / block\_size;

list\_mkk[pow\_index] = block;

memory\_size\_mkk[page\_index] = block\_size;

// определение связи буферов на странице

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

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

block = block->next;

}

block->next = NULL;

}

// степень двойки

size\_t PowOfTwo(size\_t size) {

size\_t pow = 0;

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

++pow;

}

return pow;

}

// необходимое количество страниц

size\_t MKKPageCounter(size\_t size) {

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

}

// определение номера страницы, где block

size\_t MKKPageIndex(block\_mkk\* block) {

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

}

// снова счетчики

size\_t MKKRequest() {

return request\_mkk;

}

size\_t MKKTotal() {

return total\_mkk;

}

**Результаты тестирования**

tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 10

Количество запросов: 1000

Байт: 5000

Аллокация списка свободных блоков:

Заняло времени: 0.000012

Очистка заняла: 0.000027

КПД использованной и запрошенной памяти: 0.541667

Аллокация Мак-Кьюзи-Кэрлса

Заняло времени: 0.000053

Очистка заняла: 0.000009

КПД использования и запрошенной памяти: 0.650623

tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 10

Количество запросов: 1000

Байт: 5000

Аллокация списка свободных блоков:

Заняло времени: 0.000013

Очистка заняла: 0.000022

КПД использованной и запрошенной памяти: 0.541667

Аллокация Мак-Кьюзи-Кэрлса

Заняло времени: 0.000053

Очистка заняла: 0.000008

КПД использования и запрошенной памяти: 0.656274

tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 100

Количество запросов: 1000

Байт: 5000

Аллокация списка свободных блоков:

Заняло времени: 0.000012

Очистка заняла: 0.000024

КПД использованной и запрошенной памяти: 0.7

Аллокация Мак-Кьюзи-Кэрлса

Заняло времени: 0.000059

Очистка заняла: 0.000010

КПД использования и запрошенной памяти: 0.656411

tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 100

Количество запросов: 1000

Байт: 5000

Аллокация списка свободных блоков:

Заняло времени: 0.000012

Очистка заняла: 0.000024

КПД использованной и запрошенной памяти: 0.797468

Аллокация Мак-Кьюзи-Кэрлса

Заняло времени: 0.000060

Очистка заняла: 0.000009

КПД использования и запрошенной памяти: 0.663266

tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 1000

Количество запросов: 1000

Байт: 5000

Аллокация списка свободных блоков:

Заняло времени: 0.000012

Очистка заняла: 0.000026

КПД использованной и запрошенной памяти: 0.956434

Аллокация Мак-Кьюзи-Кэрлса

Заняло времени: 0.000054

Очистка заняла: 0.000009

КПД использования и запрошенной памяти: 0.656072

tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 1000

Количество запросов: 1000

Байт: 5000

Аллокация списка свободных блоков:

Заняло времени: 0.000013

Очистка заняла: 0.000019

КПД использованной и запрошенной памяти: 0.95988

Аллокация Мак-Кьюзи-Кэрлса

Заняло времени: 0.000054

Очистка заняла: 0.000009

КПД использования и запрошенной памяти: 0.654697

tvard@tvard-HVY-WXX9:~/os/OS-labs/kp$ ./main 1000

Количество запросов: 1000

Байт: 5000

Аллокация списка свободных блоков:

Заняло времени: 0.000004

Очистка заняла: 0.000009

КПД использованной и запрошенной памяти: 0.983903

Аллокация Мак-Кьюзи-Кэрлса

Заняло времени: 0.000021

Очистка заняла: 0.000003

КПД использования и запрошенной памяти: 0.653337

**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 (наиболее подходящее) и алгоритм Мак-Кьюзи-Кэрлса.