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

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 вверены 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 раз-
мера 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 ско-
рость 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 ко-
личеству 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 - не-
которое 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 - не-
которое 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 ско-
рость 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 ско-
рость 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 ко-
личеству 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 запрашивае-
мого 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 управ-
ления 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 сте-
пень 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 но-
мер 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 дальней-
шей 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 завер-
шаем 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 раз-
делена 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 буф-
фера
                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
```

```c
#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