Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Лабораторная работа №4 по курсу

«Операционные системы»

Группа: М8О-211Б-23

Студент: Фоменко А. С.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 28.12.24

Москва, 2024

Постановка задачи

Цель работы:

Приобретение практических навыков в:

1. Создании аллокаторов памяти и их анализу;
2. Создании динамических библиотек и программ, использующие динамические библиотеки.

Задание:

Исследовать два аллокатора памяти: необходимо реализовать два алгоритма аллокации памяти и сравнить их по следующим характеристикам:

- ➢ Фактор использования
- ➢ Скорость выделения блоков
- ➢ Скорость освобождения блоков
- ➢ Простота использования аллокатора

Требуется создать две динамические библиотеки, реализующие два аллокатора, соответственно. Библиотеки загружаются в память с помощью интерфейса ОС (dlopen / LoadLibrary) для работы с динамическими библиотеками. Выбор библиотеки, реализующей аллокатор, осуществляется чтением первого аргумента при запуске программы (argv[1]). Этот аргумент должен содержать путь до динамической библиотеки (относительный или абсолютный).

Если аргумент не передан или по переданному пути библиотеки не оказалось, то указатели на функции, реализующие API аллокатора ниже, должны быть присвоены функциям, которые оборачивают системный аллокатор ОС (mmap / VirtualAlloc) в этот API. Эти аварийные оберточные функции должны быть реализованы внутри программы, которая загружает динамические библиотеки.

Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям malloc и free (realloc, опционально). Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра (mmap / VirtualAlloc). 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.

Каждый аллокатор должен обладать следующим интерфейсом (могут быть отличия в зависимости от особенностей алгоритма):

- Allocator* allocator_create(void *const memory, const size_t size) (инициализация аллокатора на памяти memory размера size);
- void allocator_destroy(Allocator *const allocator) (деинициализация структуры аллокатора);
- void* allocator_alloc(Allocator *const allocator, const size_t size) (выделение памяти аллокатором памяти размера size);
- void allocator_free(Allocator *const allocator, void *const memory) (возвращает выделенную память аллокатору);

Вариант 7. Списки свободных блоков (наиболее подходящее) и блоки по 2^n.

Общий метод и алгоритм решения

Использованные системные вызовы:

- void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset); – отражает length байтов, начиная со смещения offset файла (или другого объекта), определенного файловым дескриптором fd, в память, начиная с адреса start.
- int munmap(void *start, size_t length); – удаляет все отражения из заданной области памяти, после чего все ссылки на данную область будут вызывать ошибку "неправильное обращение к памяти".
- void *dlopen(const char *filename, int flag); – загружает динамическую библиотеку, имя которой указано в строке filename, и возвращает прямой указатель на начало динамической библиотеки.
- void *dlsym(void *handle, char *symbol); – использует указатель на динамическую библиотеку, возвращаемую dlopen, и оканчивающееся нулем символьное имя, а затем возвращает адрес, указывающий, откуда загружается этот символ. Если символ не найден, то возвращаемым значением dlsym является NULL;
- int dlclose(void *handle); – уменьшает на единицу счетчик ссылок на указатель динамической библиотеки.

Аллокатор памяти на основе списка свободных блоков

Это один из классических методов управления динамической памятью. Он основан на поддержании списка, в котором хранятся все доступные (свободные) участки памяти. Когда программе требуется выделить память, аллокатор ищет в этом списке подходящий свободный блок. После использования выделенную память возвращают обратно, и она снова попадает в список свободных блоков.

Основные компоненты и принципы работы:

Список свободных блоков представляет собой структуру данных (обычно односвязный или двусвязный список), где каждый элемент описывает свободный участок памяти.

Каждый элемент (блок) обычно содержит:

- Размер блока: количество байтов, доступных в этом свободном блоке.
- Указатель на следующий свободный блок: используется для связывания элементов списка.
- Возможны дополнительные поля, такие как флаг “свободен” или метаинформация для целей отладки.

Сама структура аллокатора обычно включает:

- Указатель на память;
- Указатель на голову списка;
- Размер памяти.

Операции:

1) Выделение памяти:
    1. Аллокатор просматривает список свободных блоков, чтобы найти подходящий блок, размер которого больше или равен запрошенному.
    2. Если подходящий блок найден, то происходит его “разрез”: часть блока выделяется для пользователя, а оставшаяся часть (если есть) остается в списке свободных блоков. Если размер подходящего блока точно соответствует запрошенному, он полностью удаляется из списка.
    3. Возвращается указатель на выделенную область памяти.

2) Освобождение памяти:
    1. Аллокатор получает указатель на ранее выделенную область.
    2. Исходя из заголовка блока, он определяет размер освобождаемого блока.
    3. Освобождаемый блок возвращается в список свободных блоков.
    4. Также часто происходит слияние освобожденного блока со смежными свободными блоками для предотвращения фрагментации.

3) Инициализация:
    1. Выделяется начальный большой блок памяти.
    2. Этот блок добавляется в список свободных блоков.

4) Стратегии поиска свободного блока:
    1. First-Fit (первый подходящий): ищет первый блок, который достаточно большой для запроса. Простая реализация и высокая скорость поиска блока, но сильно увеличивается фрагментация.
    2. Best-Fit (наилучший подходящий): ищет наименьший блок, который достаточно большой для запроса. Ведет к меньшей фрагментации, но требует просмотра всего списка, что увеличивает время выделения памяти.

Преимущества:

- Простота реализации: основные алгоритмы достаточно просты для понимания и кодирования.
- Гибкость: может быть адаптирован к различным сценариям использования и требованиям.

Недостатки:

- Фрагментация: при частом выделении и освобождении блоков возникает фрагментация – множество мелких свободных блоков, непригодных для выделения под большие запросы.
- Зависимость от стратегии: эффективность сильно зависит от выбранной стратегии поиска свободного блока и сценария использования.
- Затраты на поиск: стратегия best-fit требует просмотра всего списка, что может быть затратно по времени при большом количестве свободных блоков блоков.

Аллокатор памяти на основе блоков 2^n

Этот тип аллокатора использует подход, основанный на выделении блоков памяти размером, являющимся степенью двойки. Он эффективно управляет выделением и освобождением памяти, минимизируя фрагментацию.

Основные компоненты и принципы работы:

Информация о свободных блоках хранится в массиве (m) списков свободных блоков, где индекс массива (i) удовлетворяет условию: m[i].size == min_block_size * 2^i
Каждый элемент (блок) обычно содержит:

- Размер блока: количество байтов, доступных в этом свободном блоке (степень 2).
- Указатель на следующий свободный блок такого же размера: используется для связывания элементов списка.
- Возможны дополнительные поля, такие как флаг “свободен” или метаинформация для целей отладки.

Сама структура аллокатора обычно включает:

- Указатель на память;
- Размер памяти;
- Массив списков (динамический или статический);
- Количество списков (если массив динамический);
- Минимальный размер блока.

Операции:

1) Выделение памяти:
    1. Аллокатор ищет блок памяти размером 2^n, который равен или больше запрошенного размера. Индекс массива определяется по степени двойки.
    2. Если по найденному индексу есть блок, то он удаляется из списка блоков, иначе начинается обход блоков больших размеров.
    3. Если найден блок размером в 2 и более раз больше нужного, он разбивается на более мелкие блоки, один из которых будет выделен под запрашиваемую память.
    4. Возвращается указатель на выделенную память.

2) Освобождение памяти:
    1. Аллокатор получает указатель на ранее выделенную область.
    2. Исходя из заголовка блока, он определяет размер освобождаемого блока и высчитывает по какому индексу нужно сделать вставку (есть альтернативный вариант, у занятых блоков хранить указатель на начало списка блоков такого же размера).
    3. Освобождаемый блок возвращается в список свободных блоков.

3) Инициализация (один из подходов):
    1. Инициализируется массив списков.
    2. Выделяется блок максимальной возможной степени 2, а остальные ячейки массива заполняются блоками оставшейся памяти.

Преимущества:

- Высокая скорость поиска: поиск блока определённого размера значительно упрощен, т.к. блоки разбиты по степеням двойки.
- Простая реализация: аллокатор имеет относительно простую структуру, что упрощает его разработку и отладку.
- Отсутствие внешней фрагментации: поскольку размеры блоков фиксированы (степени двойки), не возникает внешней фрагментации.
- Минимизация накладных расходов: отсутствие перераспределения и объединения блоков уменьшает накладные расходы на управление памятью.

Недостатки:

- Неэффективность для нестандартных запросов: если размер запроса не является степенью двойки, то аллокатор будет выделять блок памяти, большего размера, чем требуется.
- Внутренняя сегментация: оптимальный размер блока может не совпадать с реальными потребностями программы, что создаёт пустоты в памяти, которые нельзя использовать, до освобождения всего блока.
- Отсутствие слияния свободных блоков: может привести к отсутствию блоков большого размера после длительного использования одного аллокатора.

Сравнение алгоритмов аллокаторов: списки свободных блоков (наиболее подходящее) и блоки по 2^n

1. **Аллокатор блоками по 2^n:**

   Фактор использования:

   Выделяются блоки фиксированного размера.

   Блок, разница размера которого с ближайшей степенью двойки в большую сторону будет меньше, чем размер блока хедера, будет занимать блок следующей степени двойки.

   Скорость выделения блоков:

   Быстрый поиск подходящего блока: Индекс списка свободных блоков вычисляется на основе двоичного логарифма.

   Извлечение блока из списка — это также быстрая операция (удаление элемента из начала списка).

   Итого: время выделения константное и не зависит от состояния памяти аллокатора

   Скорость освобождения блоков:

   Освобожденный блок просто помещается в начало списка свободных блоков, соответствующего его размеру.

   Размер блока определяется по адресу, а затем log2, что также очень быстро. Константное время.

   Простота использования:

   Необходимо заранее знать какие блоки будут более востребованы, для лучшей работы аллокатора, что затрудняет использование аллокатора в общем случае.

2. **Аллокатор списки свободных блоков (наиболее подходящее):**

   Фактор использования:

   Выделяются блоки нефиксированного размера. Выбирается наилучший блок: при поиске выбирается блок, размер которого наиболее близок к запрошенному, что в теории минимизирует фрагментацию. При освобождении можно объединять освобождаемый блок с соседними, что уменьшает фрагментацию.

   Утилизация лучше, чем у аллокатора блоков 2^n.

   Скорость выделения блоков:

   Скорость выделения блоков линейная.

   Скорость освобождения блоков:

   Возвращение блоков аллокатору происходит за линейное время.

   Простота использования:

   Алгоритм более удобен в общем случае так как можно выделять блоки любого размера, при чем фрагментация будет не большой.

Код программ:

main.c

```c
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>

#include <unistd.h>
#include <dlfcn.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <math.h>

#include "allocator.h"

Allocator* fallback_allocator_create(void* memory, size_t size) {
    return NULL;
}

void fallback_allocator_destroy(Allocator* allocator) {
    ;
}

void* fallback_allocator_alloc(Allocator* allocator, size_t size) {
    void* memory = malloc(size);
    return memory;
}

void fallback_allocator_free(Allocator* allocator, void* memory) {
    free(memory);
}

char *get_string(char * s, int *len) {
    *len = 0;
    int capacity = 1;

    char c = getchar();

    while (c != '\n') {
        s[(*len)++] = c;

        c = getchar();
    }
```

```
    s[*len] = '\0'; // завершаем строку символом конца строки

    return s;
}

int main(int argc, char **argv) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s <allocator_library> <memory_size>\n",
argv[0]);
        return EXIT_FAILURE;
    }

    void *library = dlopen(argv[1], RTLD_LOCAL | RTLD_NOW);
    if (!library) {
        fprintf(stderr, "Error loading library: %s\n", dlerror());
        return EXIT_FAILURE;
    }

    Allocator* (*allocator_create)(void*, size_t) = dlsym(library,
"allocator_create");
    void (*allocator_destroy)(Allocator*) = dlsym(library,
"allocator_destroy");
    void* (*allocator_alloc)(Allocator*, size_t) = dlsym(library,
"allocator_alloc");
    void (*allocator_free)(Allocator*, void*) = dlsym(library,
"allocator_free");

    if (!allocator_create || !allocator_destroy || !allocator_alloc ||
!allocator_free) {
            fprintf(stderr, "Error loading functions from the library. Using
fallback.\n");
            allocator_create = fallback_allocator_create;
            allocator_destroy = fallback_allocator_destroy;
            allocator_alloc = fallback_allocator_alloc;
            allocator_free = fallback_allocator_free;
    }


    size_t memory_size = (size_t)atoi(argv[2]);
    void* memory = mmap(NULL, memory_size, PROT_READ | PROT_WRITE,
MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    Allocator* allocator = allocator_create(memory, memory_size);

    // Пример использования аллокатора
```

```
    int size;

    char* ptr1 = allocator_alloc(allocator, sizeof(char) * 100);
    get_string(ptr1, &size);
    printf("%s\n", ptr1);
    allocator_free(allocator, ptr1);

    char* ptr2 = allocator_alloc(allocator, 200);
    get_string(ptr2, &size);
    printf("%s\n", ptr2);
    allocator_free(allocator, ptr2);


    char* ptr3 = allocator_alloc(allocator, 50);
    get_string(ptr3, &size);
    printf("%s\n", ptr3);
    allocator_free(allocator, ptr3);


    // Освобождение аллокатора
    allocator_destroy(allocator);
    munmap(memory, memory_size);
    dlclose(library);
    return EXIT_SUCCESS;
}
```

allocator_list.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <limits.h>

// NOTE: MSVC compiler does not export symbols unless annotated
#ifdef _MSC_VER
#define EXPORT __declspec(dllexport)
#else
#define EXPORT
#endif

typedef struct Block {
    struct Block *next;
    size_t size;
```

```
}Block;

typedef struct Allocator {
    Block *head;
    void *memory;
    size_t size;

}Allocator;

EXPORT Allocator* allocator_create(void *const memory, const size_t size) {
    Allocator *allocator = (Allocator *) mmap(NULL, sizeof(Allocator),
PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    allocator->memory = memory;
    allocator->size = size;
    allocator->head = (Block *) allocator->memory;

    allocator->head->size = size - sizeof(Block);
    allocator->head->next = NULL;

    return allocator;
}

EXPORT void allocator_destroy(Allocator *const allocator) {
    munmap(allocator, sizeof(Allocator));
}

EXPORT void* allocator_alloc(Allocator *const allocator, const size_t size) {
//best fit
    Block *current = allocator->head;
    Block *pred_block = current;
    Block *found_pred_block = current;
    Block *found = NULL;
    Block *new_block;
    size_t found_block_size = ULONG_MAX;

    while(current) { // ищем свободный блок с best fit
        if(current->size >= (size + sizeof(Block)) && current->size <
found_block_size) {
            found = current;
            found_block_size = current->size;
            found_pred_block = pred_block;
        }
        pred_block = current;
```

```
        current = current->next;
    }

    new_block = (Block *) (((char *)found) + sizeof(Block) + size);
//оставшийся свободным кусок памяти
    new_block->size = found->size - (sizeof(Block) + size);
    new_block->next = found->next;

    if(found_pred_block == allocator->head) {
        allocator->head = new_block;
    }
    else {
        found_pred_block->next = new_block; // нужно только это на самом деле
    }

    found->next = allocator->head;
    found->size = size;

    found = (void *)(((char *)found) + sizeof(Block));

    return found;
}

EXPORT Block* merge_block_with_the_next_one(Allocator *allocator, Block*
header) { //if posible
    Block *next_one = (Block *)( ((char *)header) + sizeof(Block) + header-
>size);
    if(next_one == header->next) {
        header->next = next_one->next;
        header->size += next_one->size + sizeof(Block);
    }
    return header;
}

EXPORT void allocator_free(Allocator *const allocator, void *const memory) {
    Block *header = (Block *) (((char *)memory) - sizeof(Block));
    Block *current = allocator->head;
    Block *prev = NULL;
    int flag = 1, header_before_head = 0;


    if(header < allocator->head) { // освобождаем блок до начала списка
        header->next = allocator->head;
```

```
        allocator->head = merge_block_with_the_next_one(allocator, header);
// проверка на возможность мержа исам мерж
        header_before_head = 1;
        return;
    }

    if(!(current->next)) { // свободный блок единственный
        if(header > current) { // очищаемый блок после единственного
свободного
            current->next = header;
            header->next = NULL;
            current = merge_block_with_the_next_one(allocator, current);
        }
        else { // очищаемый блок до единственного свободного
            header->next = current;
            allocator->head = merge_block_with_the_next_one(allocator,
header);
        }
        return;
    }

    while(current && header > current) { // поиск таких элементов, чтобы
prev->curr... header ...prev->current
        prev = current; // вставить после prev
        current = current->next;
    }

    if(!header_before_head) { // общий случай вставки между prev и current
        prev->next = header;
        header->next = current;
        header = merge_block_with_the_next_one(allocator, header);
        prev = merge_block_with_the_next_one(allocator, prev);
    }
}
```

allocator_fixed.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <math.h>
#include "allocator.h"

// NOTE: MSVC compiler does not export symbols unless annotated
#ifdef _MSC_VER
#define EXPORT __declspec(dllexport)
#else
#define EXPORT
#endif

typedef struct Block {
    struct Block *next;
    int index;
}Block;

typedef struct Allocator {
    Block **list;
    size_t size;
}Allocator;

EXPORT Allocator* allocator_create(void *const memory, const size_t size) {
    Allocator *allocator = (Allocator *) mmap(NULL, sizeof(Allocator),
PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    // проверка на size
    // лист будет содержать 6 указателей на списки блоков, размеры каждого
блока: 2^(i + 5)
    size_t multiplier = size / 3840;
    // количество блоков каждого размера: multiplier * (7 - i) ... 3840 = 6 *
32 + 5 * 64 + 4 * 128 + 3 * 256 + 2 * 512 + 1 * 1024
    allocator->size = size;
    allocator->list = (Block **) memory;
    Block* current_block = (Block *)(((char*)memory) + sizeof(Block *) * 6);
// первый блок вообще из всех
    Block* next_block;
    Block* prev_block;

    for(int i = 0; i < 6; i++) {
```

```
        allocator->list[i] = current_block;
        current_block->index = i;
        size_t block_size = pow(2, i + 5);
        for(int j = 0; j < (6 - i) * multiplier; j++) {
            prev_block = current_block;
            current_block->next = (Block *)(((char *)current_block) +
block_size/* + sizeof(Block)*/);
            current_block = current_block->next;
            current_block->index = i;
            if (j == ((6 - i) * multiplier - 1)) current_block->next = NULL;
        }
        prev_block->next = NULL; // последний блок в списке, ссылается на
NULL
    }


    return allocator;
}

EXPORT void allocator_destroy(Allocator *const allocator) {
    munmap(allocator->list, allocator->size);
    munmap(allocator, sizeof(Allocator));
    return;
}

EXPORT void* allocator_alloc(Allocator *const allocator, const size_t size) {
    if(size > 1008 || size <= 0 || !allocator) return NULL;
    size_t size_ = size + sizeof(Block);
    int index = (fmod(log2(size_), 1.0) < __DBL_EPSILON__) ?
((int)log2(size_)) : ((int)log2(size_)) + 1;
    index -= 5;
    Block *head = allocator->list[index]; // выделяем head

    if(!head) return NULL;
    Block *current = head->next;

    allocator->list[index] = current;
    head->next = current;
    printf("Allocated block adress %p\n", head);
    head = (Block *)((char *)head) + sizeof(Block);
    return head;
}

EXPORT void allocator_free(Allocator *const allocator, void *const memory) {
```

```
    Block* head = (Block *)((char *)memory) - sizeof(Block);
//((Block)memory) -1
    Block* current = allocator->list[head->index];

    printf("1st position: %p, freeing: %p\n", allocator->list[head->index],
head);
    allocator->list[head->index] = head;
    printf("New 1st position: %p, second one: %p\n", allocator->list[head-
>index], allocator->list[head->index]->next);

}
```

Strace:

strace ./main ./liballocator2.so 8192

execve("./main", ["./main", "./liballocator2.so", "8192"], 0x7ffe81c47420 /* 35 vars */) = 0
brk(NULL) = 0x55ad66247000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=32236, ...}) = 0
mmap(NULL, 32236, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f2676503000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libdl.so.2", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\16\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=14560, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f2676501000
mmap(NULL, 2109712, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f26760de000
mprotect(0x7f26760e1000, 2093056, PROT_NONE) = 0
mmap(0x7f26762e0000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f26762e0000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\240\35\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2030928, ...}) = 0
mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f2675ced000
mprotect(0x7f2675ed4000, 2097152, PROT_NONE) = 0
mmap(0x7f26760d4000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7f26760d4000
mmap(0x7f26760da000, 15072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f26760da000
close(3) = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f26764fe000
arch_prctl(ARCH_SET_FS, 0x7f26764fe740) = 0
mprotect(0x7f26760d4000, 16384, PROT_READ) = 0
mprotect(0x7f26762e0000, 4096, PROT_READ) = 0
mprotect(0x55ad65201000, 4096, PROT_READ) = 0
mprotect(0x7f267650b000, 4096, PROT_READ) = 0

```
munmap(0x7f2676503000, 32236)           = 0
brk(NULL)                               = 0x55ad66247000
brk(0x55ad66268000)                     = 0x55ad66268000
openat(AT_FDCWD, "./liballocator2.so", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=15728, ...}) = 0
getcwd("/home/fantastik/projects/OS/OS/lab4", 128) = 36
mmap(NULL, 16424, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f2676506000
mmap(0x7f2676507000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x1000) = 0x7f2676507000
mmap(0x7f2676508000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) =
0x7f2676508000
mmap(0x7f2676509000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x2000) = 0x7f2676509000
close(3)                                = 0
mprotect(0x7f2676509000, 4096, PROT_READ) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f2676504000
mmap(NULL, 24, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f2676503000
fstat(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 5), ...}) = 0
read(0, asdfe
"asdfe\n", 1024)                = 6
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 5), ...}) = 0
write(1, "asdfe\n", 6asdfe
)               = 6
read(0, ffff
"ffff\n", 1024)                 = 5
write(1, "ffff\n", 5ffff
)                = 5
read(0, tttttt
"tttttt\n", 1024)               = 7
write(1, "tttttt\n", 7tttttt
)               = 7
munmap(0x7f2676503000, 24)              = 0
munmap(0x7f2676504000, 8192)            = 0
munmap(0x7f2676506000, 16424)           = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Вывод

В ходе написания данной лабораторной работы я узнал об устройстве аллокакторов, изучил работу виртуального адресного пространства. Научился создавать, подключать и использовать динамические библиотеки. Были реализованы два алгоритма аллокации памяти, работающие через один API и подключаемые через динамические библиотеки. Работа аллокаторов была проверена на собственных тестах, а также проведена сравнительная характеристика двух алгоритмов. Проблем во время написания лабораторной работы не возникло.