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-210Б-23

Студент: Дворников М.Д.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 25.12.24

Москва, 2024

Постановка задачи

Вариант 8.

Требуется создать две динамические библиотеки, реализующие два аллокатора памяти: списки свободных блоков (наиболее подходящее) и алгоритм Мак-Кьюзика-Кэрелса;

Общий метод и алгоритм решения

Использованные системные вызовы:

- ssize_t write(int fd, const void *buf, size_t count); - Записывает данные в файл или файловый дескриптор.
- void *dlopen(const char *filename, int flag); - Открывает динамическую библиотеку.
- void *dlsym(void *handle, const char *symbol); - Извлекает адрес функции или переменной symbol из открытой библиотеки handle.
- int dlclose(void *handle); - Закрывает динамическую библиотеку handle.

- 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".

Описание программы

main.c

Главный файл, отвечающий за загрузку динамических библиотек, извлечение функций из них и выполнение тестов для двух аллокаторов. Если функции не обнаружены, используются заглушки, оборачивающие системные вызовы, это делается для избежания ошибок (например, mmap и munmap).

library.h

Подключает сторонние библиотеки и объявляет фукнции, которые реализовывает аллокатор.

mckusick.c

Здесь реализована логика работы аллокатора на основе алгоритма Мак-Кьюзика-Кэрелса.

- **Инициализация:**
  1. Память делится на страницы фиксированного размера (по умолчанию 4096 байт).
  2. Каждая страница имеет одно из трех состояний:
     - Свободная страница: доступна для выделения и хранит указатель на следующую свободную страницу.
     - Разделённая страница: содержит блоки фиксированного размера (размер является степенью двойки, например, 32, 64, 128 и т. д.).
     - Часть объединённой страницы: принадлежит группе страниц, используемой для выделения больших блоков памяти.
  3. Для каждой степени двойки создаётся массив страниц. Индекс массива соответствует логарифму размера блока (например, размер блока 32 байта соответствует индексу 0, 64 байта — индексу 1 и т. д.).

- **Разделение страниц на блоки:**
  1. При выделении памяти страница разбивается на блоки фиксированного размера.
  2. Для отслеживания состояния блоков внутри страницы используется битовая карта (bitmap), которая:
     - Отмечает, занятый блок или свободный.
     - Позволяет минимизировать затраты на хранение метаданных.
  3. Блоки внутри одной страницы имеют одинаковый размер.

- **Выделение памяти:**
  1. Округление запроса:
     - Запрашиваемый размер округляется до ближайшей степени двойки.
     - Например, запрос 50 байт округляется до 64, а 1 байт — до 32.
  2. Поиск подходящей страницы:
     - Аллокатор ищет страницу, содержащую свободные блоки нужного размера.
     - Если страница отсутствует, создаётся новая, которая делится на блоки требуемого размера.
  3. Выделение блока:
     - В битовой карте находится первый свободный блок.
     - Этот блок помечается как занятый.
     - Указатель на начало блока возвращается вызывающей стороне.

- **Освобождение памяти:**

1. Определение блока:
   - Освобождаемый блок определяется по переданному указателю.
   - Устанавливается соответствующий бит в битовой карте, чтобы отметить блок как свободный.
2. Объединение блоков:
   - Если все блоки в странице становятся свободными, страница помечается как полностью свободная и возвращается в список свободных страниц.
   - Аллокатор проверяет, можно ли объединить соседние страницы, чтобы уменьшить фрагментацию.

- **Граничные условия:**

1. Ограничения размеров:
   - Размеры блоков ограничены минимальной (32 байта) и максимальной (1024 байта) степенью двойки.

- Запросы, превышающие 1024 байта, требуют использования нескольких страниц.

2. Недостаток памяти:
   - Если память исчерпана, аллокатор возвращает ошибку (NULL).

   - **Особенности реализации:**
     - Битовая карта:
       - Размещается в начале каждой страницы.
       - Эффективно отслеживает состояние блоков внутри страницы.
     - Поддержка массивов страниц:
       - Массив позволяет быстро находить и использовать страницы нужного размера.
       - Упрощает управление страницами и выделенными блоками.
     - Высокая скорость работы:
       - Благодаря битовым картам и массиву страниц поиск и освобождение памяти происходят быстро.
       - Объединение блоков минимизирует фрагментацию.

freeblocks.c

Файл в котором реализована логика работы аллокатора на списке свободных блоков.

Инициализация:

- При создании аллокатора вся доступная память разбивается на один большой свободный блок.

Список свободных блоков:

- Используется односвязный список для отслеживания всех свободных блоков.
- Размеры блоков могут быть произвольными, что позволяет гибко использовать память.

Выделение памяти:

- Происходит поиск наименьшего свободного блока, подходящего под запрос.
- Если найденный блок больше, чем необходимо, он разделяется на два: первый блок удовлетворяет запрос, второй остаётся в списке свободных блоков.

Освобождение памяти:

- Освобожденный блок добавляется обратно в список свободных.
- Если соседние блоки также свободны, они объединяются в один более крупный блок для уменьшения фрагментации.

Объединение блоков:

- После освобождения блок проверяет своих соседей. Если они также свободны, блоки объединяются, чтобы минимизировать количество фрагментов памяти.

Граничные условия:

- Если размер запроса меньше минимального блока, выделяется минимально допустимый блок.
- В случае исчерпания памяти аллокатор возвращает ошибку.

Код программы

main.c

```c
#include "library.h"

#define MEMORY_POOL_SIZE 65536

void HandleError(const char *message) {
    write( fd: STDERR_FILENO,  buf: message,  nbyte: strlen( s: message));
    exit(EXIT_FAILURE);
}

static Allocator *allocator_create_stub(void *const memory, const size_t size) {
    HandleError( message: "allocator_create_stub: Fallback to mmap\n");

    void *mapped_memory = mmap(memory, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED, -1, 0);
    if (mapped_memory == MAP_FAILED) {
        HandleError( message: "allocator_create_stub: mmap failed\n");
    }

    return (Allocator *)mapped_memory;
}

static void allocator_destroy_stub(Allocator *const allocator) {
    HandleError( message: "allocator_destroy_stub: Fallback to munmap\n");

    if (allocator) {
        if (munmap(allocator, MEMORY_POOL_SIZE) == -1) {
            HandleError( message: "allocator_destroy_stub: munmap failed\n");
        }
    }
}

static void *allocator_alloc_stub(Allocator *const allocator, const size_t size) {
    HandleError( message: "allocator_alloc_stub: Fallback to mmap\n");

    void *mapped_memory = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (mapped_memory == MAP_FAILED) {
        HandleError( message: "allocator_alloc_stub: mmap failed\n");
    }

    return mapped_memory;
}

static void allocator_free_stub(Allocator *const allocator, void *const memory) {
    HandleError( message: "allocator_free_stub: Fallback to munmap\n");

    if (memory && munmap(memory, sizeof(memory)) == -1) {
        HandleError( message: "allocator_free_stub: munmap failed\n");
    }
}

static struct {
    allocator_create_f *create;
    allocator_destroy_f *destroy;
    allocator_alloc_f *alloc;
    allocator_free_f *free;
} allocator_funcs;

int main(int argc, char **argv) {
    if (argc < 2) {
        HandleError( message: "Usage: ./main <library_path>\n");
    }

    void *library = dlopen( path: argv[1],  mode: RTLD_LOCAL | RTLD_NOW);
    if (!library) {
        HandleError( message: "Failed to load library\n");
    }

    allocator_funcs.create = dlsym( handle: library,  symbol: "allocator_create");
    allocator_funcs.destroy = dlsym( handle: library,  symbol: "allocator_destroy");
    allocator_funcs.alloc = dlsym( handle: library,  symbol: "allocator_alloc");
    allocator_funcs.free = dlsym( handle: library,  symbol: "allocator_free");

    if (!allocator_funcs.create) allocator_funcs.create = allocator_create_stub;
    if (!allocator_funcs.destroy) allocator_funcs.destroy = allocator_destroy_stub;
    if (!allocator_funcs.alloc) allocator_funcs.alloc = allocator_alloc_stub;
    if (!allocator_funcs.free) allocator_funcs.free = allocator_free_stub;

    void *memory_pool = mmap(NULL, MEMORY_POOL_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (memory_pool == MAP_FAILED) {
        HandleError( message: "Memory mapping failed\n");
    }

    Allocator *allocator = allocator_funcs.create(memory_pool, MEMORY_POOL_SIZE);
    if (!allocator) {
        HandleError( message: "Failed to initialize allocator\n");
        munmap(memory_pool, MEMORY_POOL_SIZE);
        dlclose( handle: library);
    }

    int *allocated_integer = (int *)allocator_funcs.alloc(allocator, sizeof(int));
    if (allocated_integer) {
        *allocated_integer = 123;
        write( fd: STDOUT_FILENO,  buf: "Memory block allocated: integer with value 123\n",  nbyte: 48);
        allocator_funcs.free(allocator, allocated_integer);
        write( fd: STDOUT_FILENO,  buf: "Memory block freed: integer\n",  nbyte: 29);
    }

    float *allocated_float = (float *)allocator_funcs.alloc(allocator, sizeof(float));
    if (allocated_float) {
        *allocated_float = 456.78f;
        write( fd: STDOUT_FILENO,  buf: "Memory block allocated: float with value 456.78\n",  nbyte: 49);
        allocator_funcs.free(allocator, allocated_float);
        write( fd: STDOUT_FILENO,  buf: "Memory block freed: float\n",  nbyte: 26);
    }

    double *allocated_double = (double *)allocator_funcs.alloc(allocator, sizeof(double));
    if (allocated_double) {
        *allocated_double = 789.123;
        write( fd: STDOUT_FILENO,  buf: "Memory block allocated: double with value 789.123\n",  nbyte: 51);
        allocator_funcs.free(allocator, allocated_double);
        write( fd: STDOUT_FILENO,  buf: "Memory block freed: double\n",  nbyte: 28);
    }

    int *array = (int *)allocator_funcs.alloc(allocator, 10 * sizeof(int));
    if (array) {
        for (int i = 0; i < 10; i++) {
            array[i] = i;
        }
        write( fd: STDOUT_FILENO,  buf: "Memory block allocated: integer array\n",  nbyte: 39);
        allocator_funcs.free(allocator, array);
        write( fd: STDOUT_FILENO,  buf: "Memory block freed: integer array\n",  nbyte: 35);
    }

    allocator_funcs.destroy(allocator);
    write( fd: STDOUT_FILENO,  buf: "Allocator destroyed\n",  nbyte: 21);

    dlclose( handle: library);
    munmap(memory_pool, MEMORY_POOL_SIZE);

    return EXIT_SUCCESS;
}
```

library.h

```h
#ifndef ALLOCATOR_H
#define ALLOCATOR_H

#include <dlfcn.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>

#ifdef _MSC_VER
#define EXPORT __declspec(dllexport)
#else
#define EXPORT
#endif

typedef struct Allocator Allocator;
typedef struct Block Block;
typedef struct Page Page;

typedef Allocator *allocator_create_f(void *const memory, const size_t size);
typedef void allocator_destroy_f(Allocator *const allocator);
typedef void *allocator_alloc_f(Allocator *const allocator, const size_t size);
typedef void allocator_free_f(Allocator *const allocator, void *const memory);

#endif
```

freeblocks.c

```c
#include "library.h"

#define MIN_BLOCK_SIZE 32

typedef struct Block {
    size_t size;
    struct Block *next;
    bool is_free;
} Block;

typedef struct Allocator {
    Block *free_list;
    void *memory_start;
    size_t total_size;
} Allocator;

EXPORT Allocator *allocator_create(void *memory, size_t size) {
    if (!memory || size < sizeof(Allocator)) {
        return NULL;
    }

    Allocator *allocator = (Allocator *)memory;
    allocator->memory_start = (char *)memory + sizeof(Allocator);
    allocator->total_size = size - sizeof(Allocator);
    allocator->free_list = (Block *)allocator->memory_start;

    allocator->free_list->size = allocator->total_size - sizeof(Block);
    allocator->free_list->next = NULL;
    allocator->free_list->is_free = true;

    return allocator;
}

EXPORT void allocator_destroy(Allocator *const allocator) {
    if (allocator) {
        memset(allocator, 0, allocator->total_size);
    }
}

EXPORT void *allocator_alloc(Allocator *allocator, size_t size) {
    if (!allocator || size == 0) {
        return NULL;
    }

    size = (size + MIN_BLOCK_SIZE - 1) / MIN_BLOCK_SIZE * MIN_BLOCK_SIZE;

    Block *best = NULL;
    Block *prev_best = NULL;
    Block *current = allocator->free_list;
    Block *prev = NULL;

    while (current) {
        if (current->is_free && current->size >= size) {
            if (best == NULL || current->size < best->size) {
                best = current;
                prev_best = prev;
            }
        }
        prev = current;
        current = current->next;
    }
```


```c
#include "library.h"

#define MIN_BLOCK_SIZE 32

typedef struct Block {
    size_t size;
    struct Block *next;
    bool is_free;
} Block;

typedef struct Allocator {
    Block *free_list;
    void *memory_start;
    size_t total_size;
} Allocator;

EXPORT Allocator *allocator_create(void *memory, size_t size) {
    if (!memory || size < sizeof(Allocator)) {
        return NULL;
    }

    Allocator *allocator = (Allocator *)memory;
    allocator->memory_start = (char *)memory + sizeof(Allocator);
    allocator->total_size = size - sizeof(Allocator);
    allocator->free_list = (Block *)allocator->memory_start;

    allocator->free_list->size = allocator->total_size - sizeof(Block);
    allocator->free_list->next = NULL;
    allocator->free_list->is_free = true;

    return allocator;
}

EXPORT void allocator_destroy(Allocator *const allocator) {
    if (allocator) {
        memset(allocator, 0, allocator->total_size);
    }
}

EXPORT void *allocator_alloc(Allocator *allocator, size_t size) {
    if (!allocator || size == 0) {
        return NULL;
    }

    size = (size + MIN_BLOCK_SIZE - 1) / MIN_BLOCK_SIZE * MIN_BLOCK_SIZE;

    Block *best = NULL;
    Block *prev_best = NULL;
    Block *current = allocator->free_list;
    Block *prev = NULL;

    while (current) {
        if (current->is_free && current->size >= size) {
            if (best == NULL || current->size < best->size) {
                best = current;
                prev_best = prev;
            }
        }
        prev = current;
        current = current->next;
    }
```

mckusick.c

```c
#include "library.h"

#define MIN_BLOCK_SIZE 32
#define MAX_BLOCK_SIZE 1024
#define MAX_PAGE_SIZE 4096

typedef struct Page {
    size_t block_size;
    size_t free_blocks;
    struct Page *next;
    uint8_t *bitmap;
    void *data;
} Page;

typedef struct Allocator {
    Page *pages[MAX_BLOCK_SIZE / MIN_BLOCK_SIZE + 1];
    void *memory_start;
    size_t total_size;
} Allocator;

EXPORT Allocator *allocator_create(void *memory, size_t size) {
    if (!memory || size < sizeof(Allocator)) {
        return NULL;
    }

    Allocator *allocator = (Allocator *)memory;
    allocator->memory_start = (char *)memory + sizeof(Allocator);
    allocator->total_size = size - sizeof(Allocator);

    for (size_t i = 0; i <= MAX_BLOCK_SIZE / MIN_BLOCK_SIZE; ++i) {
        allocator->pages[i] = NULL;
    }

    return allocator;
}

EXPORT void allocator_destroy(Allocator *const allocator) {
    if (!allocator) return;

    memset(allocator, 0, allocator->total_size);
}

static Page *create_page(Allocator *allocator, size_t block_size) {
    if (!allocator || block_size > MAX_BLOCK_SIZE || block_size < MIN_BLOCK_SIZE) {
        return NULL;
    }

    size_t page_size = MAX_PAGE_SIZE;
    size_t bitmap_size = page_size / block_size / 8;
    void *page_memory = (void *)((char *)allocator->memory_start + allocator->total_size - page_size);

    Page *page = (Page *)page_memory;
    page->block_size = block_size;
    page->free_blocks = page_size / block_size;
    page->bitmap = (uint8_t *)((char *)page_memory + sizeof(Page));
    page->data = (void *)((char *)page_memory + sizeof(Page) + bitmap_size);

    memset(page->bitmap, 0, bitmap_size);

    return page;
}

EXPORT void *allocator_alloc(Allocator *allocator, size_t size) {
    if (!allocator || size == 0 || size > MAX_BLOCK_SIZE) {
        return NULL;
    }

    size = (size + MIN_BLOCK_SIZE - 1) / MIN_BLOCK_SIZE * MIN_BLOCK_SIZE;
    size_t page_index = size / MIN_BLOCK_SIZE - 1;

    Page *page = allocator->pages[page_index];

    if (!page) {
        page = create_page(allocator,  block_size: size);
        if (!page) return NULL;

        allocator->pages[page_index] = page;
    }

    for (size_t i = 0; i < page->free_blocks; ++i) {
        if (!(page->bitmap[i / 8] & (1 << (i % 8)))) {
            page->bitmap[i / 8] |= (1 << (i % 8));
            --page->free_blocks;

            return (void *)((char *)page->data + i * size);
        }
    }

    return NULL;
}

EXPORT void allocator_free(Allocator *allocator, void *memory) {
    if (!allocator || !memory) return;

    for (size_t i = 0; i <= MAX_BLOCK_SIZE / MIN_BLOCK_SIZE; ++i) {
        Page *page = allocator->pages[i];

        while (page) {
            if ((char *)memory >= (char *)page->data && (char *)memory < (char *)page->data + MAX_PAGE_SIZE) {
                size_t offset = (char *)memory - (char *)page->data;
                size_t block_index = offset / page->block_size;

                page->bitmap[block_index / 8] &= ~(1 << (block_index % 8));
                ++page->free_blocks;
                return;
            }

            page = page->next;
        }
    }
}
```

Протокол работы программы

Процесс тестирования:

Для тестирования использовались следующие сценарии:

- **Массовое выделение и освобождение памяти**: Проверка поведения аллокатора при выделении памяти разного размера и её последующем освобождении. Это позволяет выявить корректность работы в условиях высокой нагрузки.
- **Проверка объединения блоков**: Выделение нескольких блоков, их освобождение в произвольном порядке и последующая проверка на правильность объединения свободных блоков.
- **Измерение производительности**: Сравнение времени выполнения операций выделения и освобождения памяти для каждого алгоритма.
- **Измерение фрагментации**: Анализ использования памяти и её распределения, оценка степени фрагментации при выполнении разных операций.

Обоснование подхода тестирования

Тесты разработаны для проверки следующих характеристик:

- **Эффективность выделения памяти**: Аллокатор должен обеспечивать высокую скорость выделения памяти, что важно для приложений, интенсивно использующих динамическую память.
- **Корректность работы**: Функциональность объединения и освобождения блоков должна быть реализована без ошибок.
- **Производительность**: Аллокатор должен минимизировать накладные расходы при частом выделении и освобождении памяти.
- **Фрагментация**: Алгоритм должен минимизировать как внутреннюю, так и внешнюю фрагментацию, особенно при длительном использовании.

Результаты тестирования

Метод свободных блоков

- **Производительность**: Быстрое выделение памяти благодаря алгоритму "наиболее подходящий". Однако при освобождении нескольких блоков процесс объединения может занимать больше времени из-за необходимости проверки соседних блоков.
- **Фрагментация**: Минимальная внешняя фрагментация благодаря возможности объединения блоков.
- **Память**: Эффективное использование для запросов любого размера за счёт гибкости подхода.

Алгоритм Мак-Кьюзика-Кэрелса

- **Производительность**: Высокая скорость выделения памяти благодаря использованию битовых карт и фиксированных размеров блоков. Освобождение памяти также происходит быстро, поскольку алгоритму не требуется хранить дополнительные метаданные внутри блоков.
- **Фрагментация**: Небольшая внутренняя фрагментация из-за округления запросов до ближайшей степени двойки, однако внешняя фрагментация отсутствует.
- **Память**: Эффективен для частых запросов небольших и средних размеров, кратных степени двойки. При больших запросах страница может быть выделена полностью.

```
❱ ./main libfreeblocks.so
Memory block allocated: integer with value 123
Memory block freed: integer
Memory block allocated: float with value 456.78
Memory block freed: float
Memory block allocated: double with value 789.123
Memory block freed: double
Memory block allocated: integer array
Memory block freed: integer array
Allocator destroyed
❱ ./main libmckusick.so
Memory block allocated: integer with value 123
Memory block freed: integer
Memory block allocated: float with value 456.78
Memory block freed: float
Memory block allocated: double with value 789.123
Memory block freed: double
Memory block allocated: integer array
Memory block freed: integer array
Allocator destroyed
```

dtrace:

SYSCALL(args) = return

munmap(0x101068000, 0x84000) = 0 0

munmap(0x1010EC000, 0x8000) = 0 0

munmap(0x1010F4000, 0x4000) = 0 0

munmap(0x1010F8000, 0x4000) = 0 0

munmap(0x1010FC000, 0x48000) = 0 0

munmap(0x101144000, 0x4C000) = 0 0

open(".\0", 0x100000, 0x0) = 3 0

fcntl(0x3, 0x32, 0x16F00B0D8) = 0 0

close(0x3) = 0 0

fsgetpath(0x16F00B0E8, 0x400, 0x16F00B0C8) = 58 0

fsgetpath(0x16F00B0F8, 0x400, 0x16F00B0D8) = 14 0

__mac_syscall(0x18FEF3D62, 0x2, 0x16F00B440) = 0 0

__mac_syscall(0x18FEF0B95, 0x5A, 0x16F00B480) = 0 0

sysctl([unknown, 3, 0, 0, 0, 0] (2), 0x16F00A9E8, 0x16F00A9E0, 0x18FEF2888, 0xD) = 0 0

sysctl([CTL_KERN, 157, 0, 0, 0, 0] (2), 0x16F00AA98, 0x16F00AA90, 0x0, 0x0) = 0 0

open("/\0", 0x20100000, 0x0) = 3 0

openat(0x3, "System/Cryptexes/OS\0", 0x100000, 0x0) = 4 0

dup(0x4, 0x0, 0x0) = 5 0

fstatat64(0x4, 0x16F00A571, 0x16F00A4E0) = 0 0

openat(0x4, "System/Library/dyld/\0", 0x100000, 0x0) = 6 0

fcntl(0x6, 0x32, 0x16F00A570) = 0 0

dup(0x6, 0x0, 0x0) = 7 0

dup(0x5, 0x0, 0x0) = 8 0

close(0x3) = 0 0

close(0x5) = 0 0

close(0x4) = 0 0

close(0x6) = 0 0

__mac_syscall(0x18FEF3D62, 0x2, 0x16F00AF60) = 0 0

shared_region_check_np(0x16F00AB80, 0x0, 0x0) = 0 0

fsgetpath(0x16F00B100, 0x400, 0x16F00B028) = 82 0

fcntl(0x8, 0x32, 0x16F00B100) = 0 0

close(0x8) = 0 0

close(0x7) = 0 0

getfsstat64(0x0, 0x0, 0x2) = 10 0

getfsstat64(0x100DF4050, 0x54B0, 0x2) = 10 0

getattrlist("/\0", 0x16F00B040, 0x16F00AFB0) = 0 0

stat64("/System/Volumes/Preboot/Cryptexes/OS/System/Library/dyld/dyld_shared_cache_arm64e\0", 0x16F00B3A0, 0x0) = 0 0

stat64("/Users/matveyd/CLionProjects/OS-labs-active/lab4/src/main\0", 0x16F00A850, 0x0) = 0 0

open("/Users/matveyd/CLionProjects/OS-labs-active/lab4/src/main\0", 0x0, 0x0) = 3 0

mmap(0x0, 0x84A8, 0x1, 0x40002, 0x3, 0x0) = 0x100DF4000 0

fcntl(0x3, 0x32, 0x16F00A968) = 0 0

close(0x3) = 0 0

munmap(0x100DF4000, 0x84A8) = 0 0

__mac_syscall(0x18FEF3D62, 0x2, 0x16F0081E0) = 0 0

map_with_linking_np(0x16F0080C0, 0x1, 0x16F0080F0) = 0 0

close(0x3) = 0 0

mprotect(0x100DE8000, 0x4000, 0x1) = 0 0

open("/dev/dtracehelper\0", 0x2, 0x0) = 3 0

ioctl(0x3, 0x80086804, 0x16F007568) = 0 0

close(0x3) = 0 0

shared_region_check_np(0xFFFFFFFFFFFFFFFF, 0x0, 0x0) = 0 0

bsdthread_register(0x1901F60F4, 0x1901F60E8, 0x4000) = 1073746399 0

getpid(0x0, 0x0, 0x0) = 1769 0

shm_open(0x19008DF41, 0x0, 0xFFFFFFFF90234000) = 3 0

fstat64(0x3, 0x16F007BE0, 0x0) = 0 0

mmap(0x0, 0x8000, 0x1, 0x40001, 0x3, 0x0) = 0x100DFC000 0

close(0x3) = 0 0

csops(0x6E9, 0x0, 0x16F007D1C) = 0 0

ioctl(0x2, 0x4004667A, 0x16F007C8C) = 0 0

mprotect(0x100E0C000, 0x4000, 0x0) = 0 0

mprotect(0x100E18000, 0x4000, 0x0) = 0 0

mprotect(0x100E1C000, 0x4000, 0x0) = 0 0

mprotect(0x100E28000, 0x4000, 0x0) = 0 0

mprotect(0x100E2C000, 0x4000, 0x0) = 0 0

mprotect(0x100E38000, 0x4000, 0x0) = 0 0

mprotect(0x100E04000, 0xC8, 0x1) = 0 0

mprotect(0x100E04000, 0xC8, 0x3) = 0 0

mprotect(0x100E04000, 0xC8, 0x1) = 0 0

mprotect(0x100E3C000, 0x4000, 0x1) = 0 0

mprotect(0x100E40000, 0xC8, 0x1) = 0 0

mprotect(0x100E40000, 0xC8, 0x3) = 0 0

mprotect(0x100E40000, 0xC8, 0x1) = 0 0

mprotect(0x100E04000, 0xC8, 0x3) = 0 0

mprotect(0x100E04000, 0xC8, 0x1) = 0 0

mprotect(0x100E3C000, 0x4000, 0x3) = 0 0

mprotect(0x100E3C000, 0x4000, 0x1) = 0 0

issetugid(0x0, 0x0, 0x0) = 0 0

getentropy(0x16F0072F8, 0x20, 0x0) = 0 0

getattrlist("/Users/matveyd/CLionProjects/OS-labs-active/lab4/src/main\0", 0x16F007B80, 0x16F007B9C) = 0 0

access("/Users/matveyd/CLionProjects/OS-labs-active/lab4/src\0", 0x4, 0x0) = 0 0

open("/Users/matveyd/CLionProjects/OS-labs-active/lab4/src\0", 0x0, 0x0) = 3 0

fstat64(0x3, 0x120604470, 0x0) = 0 0

csrctl(0x0, 0x16F007D6C, 0x4) = 0 0

fcntl(0x3, 0x32, 0x16F007A68) = 0 0

close(0x3) = 0 0

proc_info(0x2, 0x6E9, 0xD) = 64 0

csops_audittoken(0x6E9, 0x10, 0x16F007DF0) = 0 0

sysctl([unknown, 3, 0, 0, 0, 0] (2), 0x16F008148, 0x16F008140, 0x193907D3A, 0x15) = 0 0

sysctl([CTL_KERN, 155, 0, 0, 0, 0] (2), 0x16F0081D8, 0x16F0081D0, 0x0, 0x0) = 0 0

open("libmckusick.so\0", 0x0, 0x0) = 3 0

fcntl(0x3, 0x32, 0x16F01A1C8) = 0 0

close(0x3) = 0 0

stat64("libmckusick.so\0", 0x16F019D30, 0x0) = 0 0

stat64("libmckusick.so\0", 0x16F019760, 0x0) = 0 0

mmap(0x0, 0x8330, 0x1, 0x40002, 0x3, 0x0) = 0x100E48000 0

fcntl(0x3, 0x32, 0x16F019878) = 0 0

close(0x3) = 0 0

open("/Users/matveyd/CLionProjects/OS-labs-active/lab4/src/libmckusick.so\0", 0x0, 0x0) = 3 0

fstat64(0x3, 0x16F018F20, 0x0) = 0 0

fcntl(0x3, 0x61, 0x16F019518) = 0 0

fcntl(0x3, 0x62, 0x16F019518) = 0 0

mmap(0x100E54000, 0x4000, 0x5, 0x40012, 0x3, 0x0) = 0x100E54000 0

mmap(0x100E58000, 0x4000, 0x3, 0x40012, 0x3, 0x4000) = 0x100E58000 0

mmap(0x100E5C000, 0x4000, 0x1, 0x40012, 0x3, 0x8000) = 0x100E5C000 0

close(0x3) = 0 0

munmap(0x100E48000, 0x8330) = 0 0

open("/Users/matveyd/CLionProjects/OS-labs-active/lab4/src/libmckusick.so\0", 0x0, 0x0) = 3 0

close(0x3) = 0 0

mprotect(0x100E58000, 0x4000, 0x1) = 0 0

mmap(0x0, 0x10000, 0x3, 0x41002, 0xFFFFFFFFFFFFFFFF, 0x0) = 0x100E60000 0

write(0x1, "Memory block allocated: integer with value 123\n\0", 0x30) = 48 0

write(0x1, "Memory block freed: integer\n\0", 0x1D) = 29 0

write(0x1, "Memory block allocated: float with value 456.78\n\0", 0x31) = 49 0

write(0x1, "Memory block freed: float\n\0", 0x1A) = 26 0

write(0x1, "Memory block allocated: double with value 789.123\n\0", 0x33) = 51 0

write(0x1, "Memory block freed: double\n\0", 0x1C) = 28 0

write(0x1, "Memory block allocated: integer array\n\0", 0x27) = 39 0

write(0x1, "Memory block freed: integer array\n\0", 0x23) = 35 0

write(0x1, "Allocator destroyed\n\0", 0x15) = 21 0

munmap(0x100E54000, 0xC000) = 0 0

munmap(0x100E60000, 0x10000) = 0 0

Вывод

В ходе выполнения этой лабораторной работы я научился работать с динамическими библиотеками, использовать системные вызовы, связанные с их подключением, и разрабатывать собственный аллокатор памяти на языке C. Я освоил создание и подключение динамических библиотек, научился обрабатывать ошибки, возникающие при их использовании, и эффективно применять их в программе. Самой сложной частью работы оказалось создание собственного аллокатора памяти, так как это была новая для меня тема. Пришлось разбираться с алгоритмами аллокаторов, искать информацию в книгах и интернете. Однако благодаря этому я лучше понял, как работает управление памятью и что лежит в основе популярных методов её распределения.