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: Коростин Н.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 16.11..24

Москва, 2024

Постановка задачи

Вариант 6.

Реализовать два алгоритма аллокации памяти и сравнить их по ряду характеристик. Виды аллокаторов:

1. Блоки по 2^n.
2. Алгоритм двойников (buddy allocator).

Общий метод и алгоритм решения

Использованные системные вызовы:

- mmap() – выделение анонимной памяти (файловый дескриптор -1)
- munmap() – удаление отображения памяти
- dlopen() – загрузка динамической библиотеки
- dlsym() – получение указателя на символ в библиотеке
- dlclose() – выгрузка динамической библиотеки

Алгоритм Блоки по 2^n:

Работа аллокаторов такого вида заключается в выделении блоков памяти, размер которых всегда равен степени двойки. Такой подход позволяет упростить управление памятью и повысить производительность. В начале аллокатору выделяется большой блок памяти (через mmap). Вся выделенная память рассматривается как единый блок самой большой степени двойки, который может быть использован. Для управления памятью создаются списки свободных блоков (free_lists), где каждый индекс соответствует блоку памяти определенного размера (например, 2^0, 2^1, 2^2 и т.д.). При освобождении блока памяти указатель на блок помещается в список свободных блоков соответствующего размера. Далее, если нужно снова выделить блок памяти такого же размера, то блок сразу берется из списка свободных блоков.

Алгоритм выделения памяти (alloc):

1. Пользователь запрашивает блок памяти определенного размера.
2. Размер округляется до ближайшей степени двойки, равной или больше запрашиваемого (например, запрос на 13 байт округляется до 16 байт).
3. Если подходящий блок есть в соответствующем списке свободных блоков (free_lists), он выделяется.
4. Если подходящего блока нет, аллокатор ищет блок большего размера в следующих списках. Если такой блок найден:
    a. Блок делится на два равных меньших блока.
    b. Один из этих меньших блоков возвращается пользователю, а второй добавляется в список свободных блоков для меньшего размера.
5. Если блок большего размера также отсутствует, возвращается ошибка (недостаточно памяти).

Алгоритм освобождения памяти (free):

1. Пользователь освобождает ранее выделенный блок памяти.
2. Освобожденный блок добавляется в соответствующий список свободных блоков (free_lists).

Данный алгоритм обладает несколькими преимуществами, такими как, например, простота реализации: структура свободных списков и логика деления блоков довольно тривиальны и эффективны. Другое преимущество: работа с фрагментацией. Внутренняя фрагментация (пустое место в выделенных блоках) уменьшается, так как блоки кратны ближайшей степени двойки. Внешняя фрагментация минимизируется, так как память разделяется строго по фиксированным размерам и переиспользуется при освобождении блока.

По этим причинам данный алгоритм лежит в основе других видов аллокаторов памяти, таких как, например, buddy allocator.

Алгоритм двойников (buddy allocator):

Алгоритм аллокации методом двойников (Buddy Allocator) – это усовершенствованная версия аллокатора на основе степеней двойки. Его основной особенностью является возможность объединения ("слияния") двух свободных блоков одинакового размера, если они являются "соседними" (buddy). Такой подход позволяет эффективно управлять памятью, уменьшая внешнюю фрагментацию.

Данный алгоритм обеспечивает не только разбиение больших блоков на более мелкие для выделения блоков, наиболее близких к размеру запрашиваемой памяти (для уменьшения внутренней фрагментации), но также реализует и объединение двух соседних свободных блоков одного размера в один большой блок.

Инициализация аллокатора (create):

1. Аллокатору выделяется большой блок памяти (через mmap), который делится на блоки степеней двойки.
2. Для управления памятью создаются списки свободных блоков (free_lists), где каждый индекс отвечает за размер блоков, равный 2^i байт.
3. Начальный блок (самый большой) добавляется в соответствующий список свободных блоков.

Алгоритм выделения памяти (alloc):

1. Пользователь запрашивает блок памяти.
2. Размер запрашиваемого блока округляется до ближайшей степени двойки.
3. Аллокатор проверяет, есть ли свободный блок нужного размера в free_lists.
4. Если подходящий блок найден, он выделяется.
5. Если свободного блока нужного размера нет, аллокатор ищет больший блок:
    a. Если найден блок большего размера, он разделяется на две части:
        i. Первый блок выделяется пользователю.
        ii. Второй блок (его buddy) добавляется в список свободных блоков меньшего размера.
6. Процесс повторяется до тех пор, пока не будет найден или создан блок нужного размера.

Алгоритм освобождения памяти (free):

1. Пользователь освобождает блок памяти.
2. Аллокатор добавляет освобожденный блок в список свободных блоков соответствующего размера.
3. Затем аллокатор проверяет, можно ли объединить этот блок с его "buddy":

a. Если блок и его buddy свободны, они объединяются в один блок большего размера.
b. Новый блок помещается в список свободных блоков большего размера.Процесс объединения продолжается до тех пор, пока блок не станет максимально возможного размера или его buddy занят.

С одной стороны, объединение блоков меньшего размера в большой блок является преимуществом buddy allocator’a по сравнению с аллокатором степени двойки. Но уменьшение фрагментации происходит за счет усложнения алгоритма и влияет на эффективность.

Тестирование

Из таблицы видно, что алгоритм двойников выигрывает алгоритм степени двойки на больших значениях количества аллокации, и примерно сравним с ним при единичных аллокациях и освобождениях памяти. Аварийная обертка на mmap проигрывает обоим этим аллокаторам и во время аллокации памяти, и во время освобождения.

Сравнение использования аллокаторов по времени, в секундах:

Вид аллокатора	**Память 1030 байт (аллокация)**	**Память 10567 (удаление)**	**Серия из 1000 аллокаций и освобождений**
2^n allocator	0.000933	0.000063	0.005763
buddy allocator	0.000744	0.000086	0.002568
emergency (mmap)	0.008086	0.000091	0.046574

Код программы

Аллокатор 2^n:

power_two_allocator.cpp:

```
#include "buddy_allocator.h"
#include <iostream>


size_t round_to_power_of_two(size_t size) {
    size_t power = 1;
    while (power < size) {
        power <<= 1;
    }
    return power;
}


int get_power_of_two(size_t size) {
    return (int)log2(size);
}


void* allocator_create(void* const memory, const size_t size) {
    if (!memory || size < sizeof(Block)) {
        return NULL;
    }
    BuddyAllocator* allocator =
        (BuddyAllocator*)mmap(NULL, sizeof(BuddyAllocator), PROT_READ | PROT_WRITE,
                              MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (allocator == MAP_FAILED) {
        return NULL;
    }
    allocator->memory = memory;
```

```
    allocator->total_size = size;
    for (size_t i = 0; i < NUM_SIZES; i++) {
        allocator->free_lists[i] = NULL;
    }
    size_t block_size = round_to_power_of_two(size);
    int power_exponent = get_power_of_two(block_size);
    if (power_exponent >= NUM_SIZES) {
        munmap(allocator, sizeof(BuddyAllocator));
        return NULL;
    }
    Block* initial_block = (Block*)memory;
    initial_block->size = block_size;
    initial_block->is_free = 1;
    initial_block->next = NULL;
    allocator->free_lists[power_exponent] = initial_block;
    return allocator;
}

void allocator_destroy(void* const buddy_allocator) {
    if (!buddy_allocator) return;
    BuddyAllocator* allocator = (BuddyAllocator*) buddy_allocator;
    munmap(allocator, sizeof(BuddyAllocator));
}

void* allocator_alloc(void* const buddy_allocator, const size_t size) {
    if (!buddy_allocator || size == 0) return NULL;
    BuddyAllocator* allocator = (BuddyAllocator*)buddy_allocator;
    size_t block_size = round_to_power_of_two(size);
    int power_exponent = get_power_of_two(block_size);
    for (int i = power_exponent; i < NUM_SIZES; i++) {
```

```
        if (allocator->free_lists[i]) {
            Block* block = allocator->free_lists[i];
            allocator->free_lists[i] = block->next;
            while (i > power_exponent) {
                i--;
                size_t smaller_block_size = 1 << i;
                Block* buddy = (Block*)((char*)block + smaller_block_size);
                buddy->size = smaller_block_size;
                buddy->is_free = 1;
                buddy->next = allocator->free_lists[i];
                allocator->free_lists[i] = buddy;
                block->size = smaller_block_size;
            }
            block->is_free = 0;
            return (void*)((char*)block + sizeof(Block));
        }
    }
    return NULL;
}

void allocator_free(void* const buddy_allocator, void* const memory) {
    if (!buddy_allocator || !memory) return;
    BuddyAllocator* allocator = (BuddyAllocator*)buddy_allocator;
    Block* block = (Block*)((char*)memory - sizeof(Block));
    block->is_free = 1;
    size_t block_size = block->size;
    int power_exponent = get_power_of_two(block_size);
    while (power_exponent < NUM_SIZES - 1) {
        size_t block_offset = (char*)block - (char*)allocator->memory;
        size_t buddy_offset = block_offset ^ block_size;
```

```
        Block* buddy = (Block*)((char*)allocator->memory + buddy_offset);
        if ((char*)buddy < (char*)allocator->memory ||
            (char*)buddy >= (char*)allocator->memory + allocator->total_size ||
            !buddy->is_free || buddy->size != block_size) {
            break;
        }
        Block** list = &allocator->free_lists[power_exponent];
        while (*list && *list != buddy) {
            list = &(*list)->next;
        }
        if (*list) {
            *list = buddy->next;
        }
        if (buddy_offset < block_offset) {
            block = buddy;
        }
        block_size *= 2;
        block->size = block_size;
        power_exponent++;
    }
    block->next = allocator->free_lists[power_exponent];
    allocator->free_lists[power_exponent] = block;
}
```

power_two_allocator.h:

```
#ifndef _2N_ALLOCATOR_
#define _2N_ALLOCATOR_

#ifdef __cplusplus
```

```
extern "C" {
#endif
#include <unistd.h>
#include <math.h>
#include <cstddef>
#include <sys/mman.h>

#define MIN_BLOCK_SIZE 8
#define MAX_BLOCK_SIZE 1024
#define MAX_BLOCK_INDEX 22

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

typedef struct TwonAllocator {
    Block* free_lists[MAX_BLOCK_INDEX];
    void* memory;
    size_t total_size;
} TwonAllocator;

size_t round_to_power_of_two(size_t);
int get_power_of_two(size_t);

void* allocator_create(void* const, const size_t);
void allocator_destroy(void* const);
void* allocator_alloc(void* const, const size_t);
void allocator_free(void* const, void* const);
```

```
#ifdef __cplusplus
}

#endif
#endif // _2N_ALLOCATOR_
```

Buddy allocator:

buddy_allocator.cpp:

```
#include "power_two_allocator.h"
#include <sys/mman.h>
size_t round_to_power_of_two(size_t size) {
    size_t power = MIN_BLOCK_SIZE;
    while (power < size) {
        power <<= 1;
    }
    return power;
}
int get_power_of_two(size_t size) {
    return (int)log2(size);
}

void* allocator_create(void* const memory, const size_t size) {
    if (!memory || size < MIN_BLOCK_SIZE) {
        return NULL;
    }
    TwonAllocator* allocator = (TwonAllocator*)mmap(
        NULL, sizeof(TwonAllocator), PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
    if (allocator == MAP_FAILED) {
        return NULL;
```

```
    }
    size_t total_size = round_to_power_of_two(size);
    size_t max_size = 1 << (MAX_BLOCK_INDEX - 1);
    if (total_size > max_size) {
        munmap(allocator, sizeof(TwonAllocator));
        return NULL;
    }

    allocator->memory = memory;
    allocator->total_size = total_size;
    for (size_t i = 0; i < MAX_BLOCK_INDEX; i++) {
        allocator->free_lists[i] = NULL;
    }
    Block* initial_block = (Block*)allocator->memory;
    initial_block->size = total_size;
    initial_block->next = NULL;
    int index = get_power_of_two(total_size) - get_power_of_two(MIN_BLOCK_SIZE);
    allocator->free_lists[index] = initial_block;
    return allocator;
}

void allocator_destroy(void* const twon_allocator) {
    if (!twon_allocator) return;
    TwonAllocator* allocator = (TwonAllocator*)twon_allocator;
    munmap(allocator, sizeof(TwonAllocator));
}

void* allocator_alloc(void* const twon_allocator, const size_t size) {
    if (!twon_allocator || size == 0) return NULL;
    TwonAllocator* allocator = (TwonAllocator*)twon_allocator;
```

```
    size_t block_size = round_to_power_of_two(size);
    size_t max_size = 1 << (MAX_BLOCK_INDEX - 1);
    if (block_size > max_size) {
        return NULL;
    }
    int index = get_power_of_two(block_size) - get_power_of_two(MIN_BLOCK_SIZE);
    if (index < 0 || index >= MAX_BLOCK_INDEX) {
        return NULL;
    }
    while (index < MAX_BLOCK_INDEX && !allocator->free_lists[index]) {
        index++;
    }
    if (index >= MAX_BLOCK_INDEX) {
        return NULL;
    }
    Block* block = allocator->free_lists[index];
    allocator->free_lists[index] = block->next;
    while (block->size > block_size) {
        size_t new_size = block->size >> 1;
        Block* buddy = (Block*)((char*)block + new_size);
        buddy->size = new_size;
        buddy->next =
            allocator->free_lists[get_power_of_two(new_size) - get_power_of_two(MIN_BLOCK_SIZE)];
        allocator->free_lists[get_power_of_two(new_size) - get_power_of_two(MIN_BLOCK_SIZE)] =
            buddy;
        block->size = new_size;
    }
    return (void*)((char*)block + sizeof(Block));
}
```

```
void allocator_free(void* const twon_allocator, void* const memory) {
    if (!twon_allocator || !memory) return;
    TwonAllocator* allocator = (TwonAllocator*)twon_allocator;
    Block* block = (Block*)((char*)memory - sizeof(Block));
    size_t block_size = block->size;
    int index = get_power_of_two(block_size) - get_power_of_two(MIN_BLOCK_SIZE);
    if (index < 0 || index >= MAX_BLOCK_INDEX) return;
    block->next = allocator->free_lists[index];
    allocator->free_lists[index] = block;
}
```

buddy_allocator.h:

```
#ifndef _BUDDY_ALLOCATOR_
#define _BUDDY_ALLOCATOR_
#ifdef __cplusplus
extern "C" {
#endif
#include <sys/mman.h>
#include <math.h>
#include <unistd.h>
#include <cstddef>
#define NUM_SIZES 30

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

typedef struct BuddyAllocator {
```

```
    void* memory;
    size_t total_size;
    Block* free_lists[NUM_SIZES];
} BuddyAllocator;

size_t round_to_power_of_two(size_t);
int get_power_of_two(size_t);

void* allocator_create(void* const, const size_t);
void allocator_destroy(void* const);
void* allocator_alloc(void* const, const size_t);
void allocator_free(void* const, void* const);

#ifdef __cplusplus
}
#endif
#endif // _BUDDY_ALLOCATOR_
```

main.cpp:

```
#include <dlfcn.h>
#include <sys/mman.h>
#include <time.h>
#include <unistd.h>

#include <cstddef>
#include <ctime>
#include <map>
#include <string>
#include <iostream>

#include "../include/errors.hpp"
#include "../include/io.hpp"

typedef void* (*create_func)(void* const, const size_t);
typedef void (*destroy_func)(void* const);
typedef void* (*alloc_func)(void* const, const size_t);
```

```
typedef void (*free_func)(void* const, void* const);

static create_func allocator_create = nullptr;
static destroy_func allocator_destroy = nullptr;
static alloc_func allocator_alloc = nullptr;
static free_func allocator_free = nullptr;

std::map<void*, size_t> allocated_blocks;

void* stub_allocator_create(void* const memory, const size_t size) {
    return memory;
}

void stub_allocator_destroy(void* const allocator) {
    write_to_file(STDOUT_FILENO, "Emergency allocator destroyed\n");
    return;
}

void* stub_allocator_alloc(void* const allocator, const size_t size) {
    void* ptr = mmap(nullptr, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANON,
-1, 0);
    if (ptr == MAP_FAILED) {
        return nullptr;
    }
    allocated_blocks[ptr] = size;
    return ptr;
}

void stub_allocator_free(void* const allocator, void* const memory) {
    munmap(memory, allocated_blocks[memory]);
    allocated_blocks.erase(memory);
}

void load_emergency_functions() {
    allocator_create = stub_allocator_create;
    allocator_destroy = stub_allocator_destroy;
    allocator_alloc = stub_allocator_alloc;
    allocator_free = stub_allocator_free;
}

void load_library_functions(char* lib_path, void** cur_library) {
    void* library = dlopen(lib_path, RTLD_LAZY);
    if (!library) {
        std::cerr << "Error: " << dlerror() << std::endl;
        return;
    }
```

```
    allocator_create = (create_func)dlsym(library, "allocator_create");
    if (!allocator_create) {
        write_to_file(STDERR_FILENO, "Failed to load 'allocator_create'\n");
    }

    allocator_destroy = (destroy_func)dlsym(library, "allocator_destroy");
    if (!allocator_destroy) {
        write_to_file(STDERR_FILENO, "Failed to load 'allocator_destroy'\n");
    }

    allocator_alloc = (alloc_func)dlsym(library, "allocator_alloc");
    if (!allocator_alloc) {
        write_to_file(STDERR_FILENO, "Failed to load 'allocator_alloc'\n");
    }

    allocator_free = (free_func)dlsym(library, "allocator_free");
    if (!allocator_free) {
        write_to_file(STDERR_FILENO, "Failed to load 'allocator_free'\n");
    }

    *cur_library = library;
}

double calculate_time(struct timespec start, struct timespec end) {
    return (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
}

int main(int argc, char** argv) {
    void* library = NULL;
    if (argc < 2) {
        log_errors(WRONG_NUMBER_OF_PARAMS);
        write_to_file(STDOUT_FILENO, "Using Emergency Implementations\n");
        load_emergency_functions();
    } else {
        load_library_functions(argv[1], &library);
        if (!allocator_create || !allocator_destroy || !allocator_alloc ||
!allocator_free) {
            log_errors(LIBRARY_OPEN_ERROR);
            write_to_file(STDOUT_FILENO, "Using Emergency Implementations\n");
            load_emergency_functions();
        }
        write_to_file(STDOUT_FILENO, "Library Loaded Successfully\n");
    }

    size_t memory_size = 1024 * 1024;
    void* memory = mmap(NULL, memory_size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANON, -1, 0);
```

```
if (memory == MAP_FAILED) {
    log_errors(MEMORY_MAPPING_FAILED);
    return 1;
}

void* allocator = allocator_create(memory, memory_size);
if (!allocator) {
    log_errors(ALLOCATOR_CREATION_ERROR);
    munmap(memory, memory_size);
    return 1;
}
print_to_stdout("Allocator created successfully\n");


// Тестинг
struct timespec start, end;
double time_used;
size_t data_size = 1030;

clock_gettime(CLOCK_MONOTONIC, &start);
void* ptr1 = allocator_alloc(allocator, data_size);
clock_gettime(CLOCK_MONOTONIC, &end);

time_used = calculate_time(start, end);
if (ptr1) {
    print_to_stdout("Time of allocation of " + std::to_string(data_size) +
                    " bytes: " + std::to_string(time_used) + " second\n");
} else {
    print_to_stdout("Allocation failed\n");
}

clock_gettime(CLOCK_MONOTONIC, &start);
allocator_free(allocator, ptr1);
clock_gettime(CLOCK_MONOTONIC, &end);

time_used = calculate_time(start, end);

print_to_stdout("Time of freeing of " + std::to_string(data_size) +
                    " bytes: " + std::to_string(time_used) + " second\n");

//////////////////////////////////////////////////////////////////////////

data_size = 10567;

clock_gettime(CLOCK_MONOTONIC, &start);
void* ptr2 = allocator_alloc(allocator, data_size);
```

```
    clock_gettime(CLOCK_MONOTONIC, &end);

    time_used = calculate_time(start, end);
    if (ptr1) {
        print_to_stdout("Time of allocation of " + std::to_string(data_size) +
                        " bytes: " + std::to_string(time_used) + " second\n");
    } else {
        print_to_stdout("Allocation failed\n");
    }

    clock_gettime(CLOCK_MONOTONIC, &start);
    allocator_free(allocator, ptr2);
    clock_gettime(CLOCK_MONOTONIC, &end);

    time_used = calculate_time(start, end);

    print_to_stdout("Time of freeing of " + std::to_string(data_size) +
                        " bytes: " + std::to_string(time_used) + " second\n");

    ////////////////////////////////////////////////////////////////

    void * test[1000];
    size_t size = 1234;
    clock_gettime(CLOCK_MONOTONIC, &start);
    for (int i = 0; i < 1000; i++) {
        test[i] = allocator_alloc(allocator, size);
    }
    for (int i = 0; i < 1000; i++) {
        allocator_free(allocator, test[i]);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);

    time_used = calculate_time(start, end);

    print_to_stdout("Time of 1000 allocations and freeings of " +
std::to_string(data_size) +
                        " bytes: " + std::to_string(time_used) + " second\n");

    allocator_destroy(allocator);
    munmap(memory, memory_size);

    if (library) {
        dlclose(library);
    }

    return 0;
}
```

Протокол работы программы

```
strace ./main_exec
/home/nikita/operation_systems/lab4/lib_buddy_allocator/libbuddy_allocator.so
execve("./main_exec", ["./main_exec", "/home/nikita/operation_systems/l"...], 0x7ffdd3540a88
/* 56 vars */) = 0
brk(NULL)                               = 0x5dc2cc64a000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x750458a0f000
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=70647, ...}) = 0
mmap(NULL, 70647, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7504589fd000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libstdc++.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\0\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=2592224, ...}) = 0
mmap(NULL, 2609472, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x750458600000
mmap(0x75045869d000, 1343488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x9d000) = 0x75045869d000
mmap(0x7504587e5000, 552960, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e5000) =
0x7504587e5000
mmap(0x75045886c000, 57344, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x26b000) = 0x75045886c000
mmap(0x75045887a000, 12608, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1,
0) = 0x75045887a000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libgcc_s.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=183024, ...}) = 0
mmap(NULL, 185256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7504589cf000
mmap(0x7504589d3000, 147456, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x4000) = 0x7504589d3000
mmap(0x7504589f7000, 16384, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) =
0x7504589f7000
mmap(0x7504589fb000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x2b000) = 0x7504589fb000
close(3)                                = 0
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\220\243\2\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) =
784
fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) =
784
mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x750458200000
mmap(0x750458228000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
```

```
0x28000) = 0x750458228000
mmap(0x7504583b0000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) =
0x7504583b0000
mmap(0x7504583ff000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x1fe000) = 0x7504583ff000
mmap(0x750458405000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1,
0) = 0x750458405000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libm.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\0\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=952616, ...}) = 0
mmap(NULL, 950296, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7504588e6000
mmap(0x7504588f6000, 520192, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x10000) = 0x7504588f6000
mmap(0x750458975000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x8f000) =
0x750458975000
mmap(0x7504589cd000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0xe7000) = 0x7504589cd000
close(3)                                = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7504588e4000
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7504588e1000
arch_prctl(ARCH_SET_FS, 0x7504588e1740) = 0
set_tid_address(0x7504588e1a10)         = 11163
set_robust_list(0x7504588e1a20, 24)     = 0
rseq(0x7504588e2060, 0x20, 0, 0x53053053) = 0
mprotect(0x7504583ff000, 16384, PROT_READ) = 0
mprotect(0x7504589cd000, 4096, PROT_READ) = 0
mprotect(0x7504589fb000, 4096, PROT_READ) = 0
mprotect(0x75045886c000, 45056, PROT_READ) = 0
mprotect(0x5dc2a0b73000, 4096, PROT_READ) = 0
mprotect(0x750458a47000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7504589fd000, 70647)           = 0
futex(0x75045887a7bc, FUTEX_WAKE_PRIVATE, 2147483647) = 0
getrandom("\xed\xb0\x6c\xc4\xe1\x2f\x75\x9f", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x5dc2cc64a000
brk(0x5dc2cc66b000)                     = 0x5dc2cc66b000
openat(AT_FDCWD,
"/home/nikita/operation_systems/lab4/lib_buddy_allocator/libbuddy_allocator.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"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0775, st_size=40552, ...}) = 0
mmap(NULL, 16448, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x750458a0a000
mmap(0x750458a0b000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x1000) = 0x750458a0b000
mmap(0x750458a0c000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) =
0x750458a0c000
mmap(0x750458a0d000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x2000) = 0x750458a0d000
```

```
close(3)                                = 0
mprotect(0x750458a0d000, 4096, PROT_READ) = 0
write(1, "Library Loaded Successfully\n", 28Library Loaded Successfully
) = 28
mmap(NULL, 1048576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x750458500000
mmap(NULL, 256, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x750458a09000
write(1, "Allocator created successfully\n", 31Allocator created successfully
) = 31
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=1164157}) = 0
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=1175232}) = 0
write(1, "Time of allocation of 1030 bytes"..., 50Time of allocation of 1030 bytes: 0.000011
second
) = 50
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=1198094}) = 0
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=1201703}) = 0
write(1, "Time of freeing of 1030 bytes: 0"..., 47Time of freeing of 1030 bytes: 0.000003
second
) = 47
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=1210822}) = 0
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=1214372}) = 0
write(1, "Time of allocation of 2031 bytes"..., 50Time of allocation of 2031 bytes: 0.000004
second
) = 50
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=1223016}) = 0
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=1226474}) = 0
write(1, "Time of freeing of 2031 bytes: 0"..., 47Time of freeing of 2031 bytes: 0.000003
second
) = 47
munmap(0x750458a09000, 256)             = 0
munmap(0x750458500000, 1048576)         = 0
munmap(0x750458a0a000, 16448)           = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Вывод

В ходе выполнения лабораторной работы я получил навыки в создании динамических библиотек и реализовал программу, использующую динамическую библиотеку. Также были изучены принципы работы с памятью, создания аллокаторов различных типов и их анализа. Для выполнения лабораторной я разобрался в алгоритме buddy allocator, а также в более простых алгоритмах аллокации памяти, что помогло сравнить два реализованных мной аллокатора и оценить их эффективность.