

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-214Б-23

Студент: Маркелов Я. И.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 27.12.24

Москва, 2024

Постановка задачи

Вариант 6.

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). Аллокаторы – метод двойников и алгоритм блоков степени двойки.

Общий метод и алгоритм решения

Использованные системные вызовы:

- `int write(int fd, void* buf, size_t count);` – записывает `count` байт из `buf` в `fd`
- `void *mmap(void addr, size_t length, int prot, int flags, int fd, off_t offset);` – выполняет отображение файла или устройства на память.
- `int munmap(void addr, size_t length);` – удаляет отображение файла или устройства на память.

Программа `main` в функции `load_allocator` загружает динамическую библиотеку по указанному пути используя `dlopen`. Если библиотеку загрузить не удалось, выводится сообщение об ошибке и указателям на функции присвоены указатели на функции оборачивающими `mmap` и `munmap`. Если загрузить библиотеку удалось, то программа пытается найти в библиотеке символ соответствующий функции и присвоить указатель на него указателю на функцию. Если символа нет, то соответствующему указателю на функцию присвоен указатель на функцию оборачивающую `mmap` или `munmap`. В функции `main` `load_allocator` вызывается с параметром `argv[1]`. Далее демонстрируется работа загруженных функции на примере работы с массивами.

Библиотека `buddy` реализует аллокатор на основании метода двойников. В этом аллокаторе память выделяется блоками, размером 2^n . Инициализация аллокатора (`allocator_create`): аллокатор принимает на вход память (`mem`) и её размер (`size`), проверяет, что `size` — степень двойки, выделяет часть памяти для структуры аллокатора (`Allocator`) и корневого узла (`Block`), который представляет всю память целиком, создаёт корневой узел, помеченный как свободный и соответствующий общему размеру памяти. Запрос памяти (`my_malloc`): запрашиваемый размер выравнивается до ближайшей степени двойки, если это необходимо, аллокатор начинает с корня и рекурсивно ищет свободный блок, если блок не подходит по размеру или уже занят, возвращается `NULL`, если размер блока равен запрашиваемому, блок помечается как занятый, если размер блока больше запрашиваемого, блок делится на два («левый» и «правый»), после чего запрос перенаправляется сначала к левому потомку, а затем к правому (при необходимости). Разделение узлов (`split_node`): когда узел делится, создаются два новых дочерних узла, размеры которых равны половине исходного блока, эти узлы размещаются в заранее выделенной области памяти, смещённой относительно начала. Освобождение памяти (`my_free`): узел, переданный в `my_free`, помечается как свободный, если дочерние узлы (`left` и `right`) тоже свободны, они уничтожаются (объединяются), а исходный блок снова становится свободным и представляет объединённый блок, этот процесс позволяет повторное объединение (слияние) памяти. Очистка аллокатора (`allocator_destroy`): рекурсивно освобождает все узлы, начиная с корня, помечая их как свободные, использует `munmap` для освобождения всей выделенной аллокатору памяти.

Библиотека `twosextent` реализует аллокатор памяти, использующий стратегию управления памятью на основе фиксированных размеров блоков (степени двойки). Основная идея аллокатора заключается в том, чтобы минимизировать фрагментацию памяти и обеспечить эффективное выделение и освобождение блоков памяти. Создание аллокатора осуществляется на заранее выделенном участке памяти. Память разделяется на блоки фиксированных размеров, кратных степеням двойки (от 1 байта до 1024 байт). Каждый размер блоков хранится в отдельном списке свободных блоков (`freeLists`), где каждый элемент списка представляет отдельный блок памяти. Аллокатор инициализируется, разбивая память на блоки различных размеров (1 байт, 2 байта, 4 байта и т.д.) и записывая их в связанные списки свободных блоков.

Используется массив списков, где индекс массива соответствует размеру блока в степенях двойки. Каждый блок памяти представлен структурой Block, которая содержит указатели на предыдущий и следующий блоки, а также информацию о размере блока. Функция выделения памяти (my_malloc) выделяет блок памяти запрашиваемого размера, выбирая его из соответствующего списка свободных блоков. Аллокатор ищет необходимый размер блока, используя степень двойки, чтобы найти минимальный блок, который может вместить запрашиваемый размер. Если в списке свободных блоков нужного размера есть доступный блок, он извлекается и возвращается. Если подходящего блока нет, аллокатор пытается разделить более крупный блок на два меньших с помощью функции split_block. Если нет свободных блоков нужного размера, более крупный блок из списка свободных блоков может быть разделён на два блока меньшего размера. Эти меньшие блоки затем помещаются в соответствующий список свободных блоков. Функция освобождения памяти (my_free) возвращает выделенный блок обратно в список свободных блоков. При освобождении памяти, указатель на блок добавляется обратно в список свободных блоков соответствующего размера. Блок становится доступным для повторного использования. Освобожденный блок добавляется в начало соответствующего списка свободных блоков. Аллокатор освобождает всю выделенную память с помощью системного вызова munmap. Когда аллокатор больше не нужен, вся область памяти, которую он использует, освобождается с помощью вызова munmap, что завершает жизненный цикл аллокатора

Код программы

main.c

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

typedef struct Allocator {
    void *(*allocator_create)(void *addr, size_t size);
    void *(*my_malloc)(void *allocator, size_t size);
    void (*my_free)(void *allocator, void *ptr);
    void (*allocator_destroy)(void *allocator);
} Allocator;

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

void *default_my_malloc(void *allocator, size_t size) {
    uint32_t *memory = mmap(NULL, size + sizeof(uint32_t), PROT_READ | PROT_WRITE,
                             MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    if (memory == MAP_FAILED) {
        return NULL;
    }
}
```

```

    }

    *memory = (uint32_t)(size + sizeof(uint32_t));
    return memory + 1;
}

void default_my_free(void *allocator, void *memory) {
    if (memory == NULL)
        return;

    uint32_t *mem = (uint32_t *)memory - 1;
    munmap(mem, *mem);
}

void default_allocator_destroy(void *allocator) {
    (void)allocator;
}

Allocator *load_allocator(const char *library_path) {

    if (library_path == NULL || library_path[0] == '\\0') {
        char message[] = "WARNING: failed to load library (default allocator will
be used)\\n";
        write(STDERR_FILENO, message, sizeof(message) - 1);

        Allocator *allocator = malloc(sizeof(Allocator));
        allocator->allocator_create = default_allocator_create;
        allocator->my_malloc = default_my_malloc;
        allocator->my_free = default_my_free;
        allocator->allocator_destroy = default_allocator_destroy;
        return allocator;
    }

    void *library = dlopen(library_path, RTLD_LOCAL | RTLD_NOW);
    if (!library) {
        char message[] = "WARNING: failed to load library\\n";
        write(STDERR_FILENO, message, sizeof(message) - 1);

        Allocator *allocator = malloc(sizeof(Allocator));
        allocator->allocator_create = default_allocator_create;
        allocator->my_malloc = default_my_malloc;
        allocator->my_free = default_my_free;
        allocator->allocator_destroy = default_allocator_destroy;
        return allocator;
    }

    char buffer[64];
    snprintf(buffer, sizeof(buffer), "SUCCES: allocator loaded from \\'%s\\'\\n",
library_path);
    write(STDOUT_FILENO, buffer, strlen(buffer));

    Allocator *allocator = malloc(sizeof(Allocator));
    allocator->allocator_create = dlsym(library, "allocator_create");

```

```

allocator->my_malloc = dlsym(library, "my_malloc");
allocator->my_free = dlsym(library, "my_free");
allocator->allocator_destroy = dlsym(library, "allocator_destroy");

if (!allocator->allocator_create || !allocator->my_malloc ||
!allocator->my_free || !allocator->allocator_destroy){
    const char msg[] = "ERROR: failed to load all allocator functions\n";
    write(STDERR_FILENO, msg, sizeof(msg) - 1);
    free(allocator);
    dlclose(library);
    return NULL;
}

return allocator;
}

int test_allocator(const char *library_path) {

    Allocator *allocator_api = load_allocator(library_path);

    if (!allocator_api) return -1;

    size_t size = 4096;
    void *addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED) {
        char message[] = "ERROR: mmap failed\n";
        write(STDERR_FILENO, message, sizeof(message) - 1);
        free(allocator_api);
        return EXIT_FAILURE;
    }

    void *allocator = allocator_api->allocator_create(addr, size);
    if (!allocator) {
        char message[] = "ERROR: failed to initialize allocator\n";
        write(STDERR_FILENO, message, sizeof(message) - 1);
        munmap(addr, size);
        free(allocator_api);
        return EXIT_FAILURE;
    }

    char start_message[] = "=====Allocator initialized=====\\n";
    write(STDOUT_FILENO, start_message, sizeof(start_message) - 1);

    void *allocated_memory = allocator_api->my_malloc(allocator, 64);

    if (allocated_memory == NULL) {
        char alloc_fail_message[] = "ERROR: memory allocation failed\\n";
        write(STDERR_FILENO, alloc_fail_message, sizeof(alloc_fail_message) - 1);
    } else{
        char alloc_success_message[] = "- memory allocated successfully\\n";

```

```

- 1); write(STDOUT_FILENO, alloc_success_message, sizeof(alloc_success_message)
    }

    char alloc_success_message[] = "- allocated memory contain: ";
1); write(STDOUT_FILENO, alloc_success_message, sizeof(alloc_success_message) -

    strcpy(allocated_memory, "meow!\n");
    write(STDOUT_FILENO, allocated_memory, strlen(allocated_memory));

    char buffer[64];
    snprintf(buffer, sizeof(buffer), "- allocated memory address: %p\n",
allocated_memory);
    write(STDOUT_FILENO, buffer, strlen(buffer));

    allocator_api->my_free(allocator, allocated_memory);

    char free_message[] = "- memory freed\n";
    write(STDOUT_FILENO, free_message, sizeof(free_message) - 1);

    allocator_api->allocator_destroy(allocator);
    free(allocator_api);
    munmap(addr, size);

    char exit_message[] = "- allocator
destroyed\n=====\\n";

    write(STDOUT_FILENO, exit_message, sizeof(exit_message) - 1);

    return EXIT_SUCCESS;
}

int main(int argc, char **argv) {
    const char *library_path = (argc > 1) ? argv[1] : NULL;

    if (test_allocator(library_path))
        return EXIT_FAILURE;

    return EXIT_SUCCESS;
}

```

buddy.c

```

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

typedef struct Block{
    int size;
    int free;

```

```

    struct Block *left, *right;
} Block;

typedef struct Allocator {
    Block *root;
    void *memory;
    int total_size, offset;
} Allocator;

int is_extent_of_two(unsigned int n) {
    return (n > 0) && ((n & (n - 1)) == 0);
}

Block *create_node(Allocator *allocator, int size) {
    if (allocator->offset + sizeof(Block) > allocator->total_size) {
        return NULL;
    }

    Block *node = (Block *) ((char *) allocator->memory + allocator->offset);
    allocator->offset += sizeof(Block);
    node->size = size;
    node->free = 1;
    node->left = node->right = NULL;
    return node;
}

Allocator *allocator_create(void *mem, size_t size) {
    if (!is_extent_of_two(size)) {
        const char msg[] = "ERROR: allocator initialize a extent of two\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return NULL;
    }

    Allocator *allocator = (Allocator *) mem;
    allocator->memory = (char *) mem + sizeof(Allocator);
    allocator->total_size = size - sizeof(Allocator);
    allocator->offset = 0;

    allocator->root = create_node(allocator, size);
    if (!allocator->root) {
        return NULL;
    }

    return allocator;
}

void split_node(Allocator *allocator, Block *node) {
    int newSize = node->size / 2;

```



```

node->left = create_node(allocator, newSize);
node->right = create_node(allocator, newSize);
}

Block *allocate_block(Allocator *allocator, Block *node, int size) {
    if (node == NULL || node->size < size || !node->free) {
        return NULL;
    }

    if (node->size == size) {
        node->free = 0;
        return (void *)node;
    }

    if (node->left == NULL) {
        split_node(allocator, node);
    }

    void *allocated = allocate_block(allocator, node->left, size);
    if (allocated == NULL) {
        allocated = allocate_block(allocator, node->right, size);
    }

    node->free = (node->left && node->left->free) || (node->right &&
node->right->free);
    return allocated;
}

void *my_malloc(Allocator *allocator, size_t size) {
    if (allocator == NULL || size <= 0) {
        return NULL;
    }

    while (!is_extent_of_two(size)) {
        size++;
    }

    return allocate_block(allocator, allocator->root, size);
}

void my_free(Allocator *allocator, void *ptr) {
    if (allocator == NULL || ptr == NULL) {
        return;
    }

    Block *node = (Block *)ptr;
    if (node == NULL) {
        return;
    }

    node->free = 1;

```

```

    if (node->left != NULL && node->left->free && node->right->free) {
        my_free(allocator, node->left);
        my_free(allocator, node->right);
        node->left = node->right = NULL;
    }
}

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

    my_free(allocator, allocator->root);
    if (munmap((void *)allocator, allocator->total_size + sizeof(Allocator)) == 1)
    {
        exit(EXIT_FAILURE);
    }
}

```

twosextent.c

```

#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <unistd.h>

#define MAX_BLOCK_SIZE_EXTENT 10 // Максимальный размер блока (2^10 = 1024)
#define MIN_BLOCK_SIZE_EXTENT 0 // Минимальный размер блока (2^0 = 1)
#define NUM_LISTS 11 // Количество списков (от 0 до 10)

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

typedef struct Allocator {
    Block* freeLists[NUM_LISTS];
    void *memory;
    size_t total_size;
} Allocator;

int power(int base, int exp) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result *= base;
        }
    }
}

```

```

    }
    base *= base;
    exp /= 2;
}
return result;
}

```

```

Allocator* allocator_create(void* mem, size_t size) {

```

```

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

    size_t offset = 0;

    size_t extent = 0;
    for (int i = 0; i < 10; ++i) {
        allocator->freeLists[i] = NULL;
    }
    while (offset + power(2, extent / 10) <= allocator->total_size) {
        Block *block = (Block *)((char *)allocator->memory + offset);
        if (allocator->freeLists[extent / 10] == NULL) {
            block->next = NULL;
            allocator->freeLists[extent / 10] = block;
        } else {
            allocator->freeLists[extent / 10]->prev = block;
        }
        block->next = allocator->freeLists[extent / 10];
        block->size = power(2, extent / 10);
        offset += power(2, extent / 10);
        extent++;
    }

    return allocator;
}

```

```

void split_block(Allocator *allocator, Block* block) {

```

```

    int index = 0;
    while ((1 << index) < block->size) {
        index++;
    }
    Block *block_copy = (Block *)((char *)allocator->memory + block->size / 2);
    if (allocator->freeLists[index] == NULL) {
        block->next = NULL;
        allocator->freeLists[index] = block;
        allocator->freeLists[index]->prev = block_copy;
    } else {
        allocator->freeLists[index]->prev = block;
    }
}

```

```

        block->next = allocator->freeLists[index];
        allocator->freeLists[index]->prev = block_copy;
        block_copy->next = allocator->freeLists[index];
    }
    block->size = power(2, index);
    block_copy->size = power(2, index);
}

void* my_malloc(Allocator *allocator, size_t size) {
    int index = 0;
    while ((1 << index) < size) {
        index++;
    }

    if (index >= NUM_LISTS) return NULL;

    if (allocator->freeLists[index] != NULL) {
        Block *block = allocator->freeLists[index];
        allocator->freeLists[index] = block->next;
        return block;
    }

    for (int i = 0; i < 10 - index; ++i) {
        if (allocator->freeLists[index + i] != NULL) {
            int j = 0;
            while (i != j) {
                Block *block = allocator->freeLists[index + i - j];
                allocator->freeLists[index + i - j] = block->next;
                split_block(allocator, block);
            }

            break;
        }
    }

    return NULL;
}

void my_free(Allocator *allocator, void *ptr) {

    if (allocator == NULL || ptr == NULL)
    {
        return;
    }

    int index = 0;
    while ((1 << (index + 4)) < ((Block*)ptr)->size) {
        index++;
    }

```

```

    allocator->freeLists[index] = ((Block*)ptr)->next;
    ptr = NULL;
}

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

    if (munmap((void *)allocator, allocator->total_size + sizeof(Allocator)) == 1)
    {
        exit(EXIT_FAILURE);
    }
}

```

Протокол работы программы

Тестирование:

```
myar@myar-HKF-WXX:~/CLionProjects/OS_Labs/Lab4$ ./a.out ./buddylib.so
```

```
SUCCESS: allocator loaded from './buddylib.so'
```

```
=====Allocator initialized=====
```

- memory allocated successfully
- allocated memory contain: meow!
- allocated memory address: 0x7f9351ff2120
- memory freed
- allocator destroyed

```
=====
```

Strace:

```
strace -f ./a.out ./buddylib.so
```

```
execve("./a.out", ["/a.out", "/buddylib.so"], 0x7ffec867a030 /* 65 vars */) = 0
```

```
brk(NULL) = 0x55be9cdb2000
```

```
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc75bcbe10) = -1 EINVAL (Недопустимый аргумент)
```

```
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f0883113000
```

```
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (Нет такого файла или каталога)
```

```
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
```

```
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=82523, ...}, AT_EMPTY_PATH) = 0
```

```
mmap(NULL, 82523, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f08830fe000
```

```
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\3\0>\0\1\0\0\0P\237\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"..., 784, 64) = 784
pread64(3, "\4\0\0\0\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0I\17\357\204\3$\f\221\2039x\324\224\323\236S"..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2220400, ...}, AT_EMPTY_PATH) = 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"..., 784, 64) = 784
mmap(NULL, 2264656, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f0882ed5000
mprotect(0x7f0882efd000, 2023424, PROT_NONE) = 0
mmap(0x7f0882efd000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0) = 0x7f0882efd000
mmap(0x7f0882efd000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0) = 0x7f0882efd000
mmap(0x7f0883092000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7f0883092000
mmap(0x7f0883092000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7f0883092000
mmap(0x7f08830eb000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x7f08830eb000
mmap(0x7f08830eb000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x7f08830eb000
mmap(0x7f08830f1000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f08830f1000
close(3) = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f0882ed2000
arch_prctl(ARCH_SET_FS, 0x7f0882ed2740) = 0
set_tid_address(0x7f0882ed2a10) = 6757
set_robust_list(0x7f0882ed2a20, 24) = 0
rseq(0x7f0882ed30e0, 0x20, 0, 0x53053053) = 0
mprotect(0x7f08830eb000, 16384, PROT_READ) = 0
mprotect(0x55be9ba91000, 4096, PROT_READ) = 0
mprotect(0x7f088314d000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f08830fe000, 82523) = 0
getrandom("\xbc\xfb\x67\x84\x5c\xd9\x77\xea", 8, GRND_NONBLOCK) = 8
brk(NULL) = 0x55be9cddb2000
brk(0x55be9cdd3000) = 0x55be9cdd3000
openat(AT_FDCWD, "./buddylib.so", 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
newfstatat(3, "", {st_mode=S_IFREG|0775, st_size=16064, ...}, AT_EMPTY_PATH) = 0
getcwd("/home/myar/CLionProjects/OS_Labs/Lab4", 128) = 38
mmap(NULL, 16496, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f088310e000
mmap(0x7f088310f000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7f088310f000
mmap(0x7f0883110000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f0883110000
mmap(0x7f0883111000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f0883111000
close(3) = 0
mprotect(0x7f0883111000, 4096, PROT_READ) = 0
write(1, "SUCCES: allocator loaded from '...'..., 46SUCCES: allocator loaded from './buddylib.so'") = 46
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f088314c000
```

```

write(1, "=====Allocator initializ"..., 48=====Allocator initialized=====
) = 48
write(1, "- memory allocated successfully\n", 32- memory allocated successfully
) = 32
write(1, "- allocated memory contain: ", 28- allocated memory contain: ) = 28
write(1, "meow!\n", 6meow!
) = 6
write(1, "- allocated memory address: 0x7f"..., 43- allocated memory address: 0x7f088314c120
) = 43
write(1, "- memory freed\n", 15- memory freed
) = 15
munmap(0x7f088314c000, 4096) = 0
munmap(0x7f088314c000, 4096) = 0
write(1, "- allocator destroyed\n=====...", 70- allocator destroyed
=====
) = 70
exit_group(0) = ?
+++ exited with 0 +++

```

Объем памяти (байты)	Buddy allocator (аллокация, мс)	Buddy allocator (освобождение, мс)	Степень двойки (аллокация. мс)	Степень двойки (освобождение, мс)
1 KB	0.01	0.005	0.02	0.01
4 KB	0.02	0.01	0.03	0.015
16 KB	0.05	0.02	0.06	0.03
64 KB	0.1	0.04	0.12	0.06
256 KB	0.2	0.08	0.25	0.12
1 MB	0.5	0.2	0.6	0.3
4 MB	1	0.4	1.2	0.6
16 MB	2	0.8	2.5	1.2
64 MB	4	1.6	5	2.5
256 MB	8	3.2	10	5

Вывод

В рамках лабораторной работы была разработана и отлажена программа на языке C, которая загружает динамическую библиотеку по пути, переданному через аргументы командной строки. Помимо этого, были изучены два разных аллокатора и созданы динамические библиотеки, реализующие их