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-213Б-23

Студент: Солодова С. М.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата:_____

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). 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 (см. пример на GitHub Gist). 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

Блоки по 2^n и алгоритм двойников.

Общий метод и алгоритм решения

В данной работе использованы следующие системные вызовы и функции для работы с разделяемой памятью:

- **dlopen**(): открывает динамическую библиотеку
- **mmap()**: отображение объекта разделяемой памяти в адресное пространство процесса.
- **munmap()**: удаление отображения объекта разделяемой памяти из адресного
- **dlsym()**: Получает адрес символа (функции или переменной) из открытой библиотеки.
- **dlclose()**: Закрывает ранее открытое отображение библиотеки.
- **mprotect()**: изменяет защиту для страниц памяти вызывающего процесса

Аллокатор с алгоритмом двойников:

1. Инициализация Аллокатора (allocator_create)
2. Проверка Степени Двойки: Убедиться, что общий размер пула памяти (size) является степенью двойки.
3. Инициализация Структуры Аллокатора:

- Установить memory как указатель на начало выделенного пула памяти.
- Установить total_size как общий размер пула за вычетом размера структуры Allocator.
- Инициализировать offset нулём.

4. Создание Корневого Блока:
   - Вычислить наибольшую степень двойки, меньшую или равную total_size.
   - Создать корневой блок с размером root_size.
   - Установить root как указатель на корневой блок.
5. Выделение Памяти (my_malloc)
6. Проверка Параметров: Убедиться, что аллокатор и размер запроса валидны.
7. Округление До Степени Двойки: Округлить запрашиваемый размер до ближайшей степени двойки (power_size).
8. Выделение Блока:
   - Рекурсивно проходить по дереву блоков, начиная с корневого.
   - Если текущий блок свободен и его размер больше или равен запрашиваемому, попытаться разбить его на два дочерних блока.
   - Продолжать разбиение до достижения необходимого размера.
   - Пометить подходящий блок как занятый (free = 0) и вернуть указатель на память после метаданных блока.
9. Освобождение Памяти (my_free)
   - Проверка Параметров: Убедиться, что аллокатор и указатель валидны.
   - Идентификация Блока: Преобразовать пользовательский указатель обратно в указатель на структуру Block путем вычитания размера Block из пользовательского указателя.
   - Освобождение Блока: Установить статус блока как свободного (free = 1).
   - Попытка Объединения: Если оба дочерних блока свободны, удалить ссылки на них, объединяя их в родительский блок.
10. Деинициализация Аллокатора (allocator_destroy)
    - Освобождение Памяти: Использовать системный вызов munmap для освобождения выделенного пула памяти.
    - Завершение Работы: При возникновении ошибки при освобождении, вывести сообщение об ошибке и завершить программу.

Аллокатор с блоками по 2^n:

1. Инициализация Аллокатора (allocator_create)
2. Инициализация Структуры Аллокатора:
   - Установить memory как указатель на начало выделенного пула памяти.
   - Установить total_size как общий размер пула за вычетом размера структуры Allocator.
   - Инициализировать массив freeLists как пустые списки.
3. Создание Начального Блока:
   - Вычислить наибольшую степень двойки, меньшую или равную total_size.
   - Создать начальный блок с размером initial_size.
   - Добавить начальный блок в соответствующий список свободных блоков.
4. Выделение Памяти (my_malloc)
5. Проверка Параметров: Убедиться, что аллокатор и размер запроса валидны.
6. Округление До Степени Двойки: Округлить запрашиваемый размер до ближайшей степени двойки (actual_size), соответствующей минимальному размеру блока.
7. Поиск Свободного Блока:
   - Определить индекс соответствующего списка свободных блоков.
   - Если в этом списке нет свободных блоков, искать в списках больших размеров.

8. Разбиение Блоков:
    - Если найденный свободный блок больше необходимого размера, разбить его на два меньших блоки.
    - Добавить полученные блоки в соответствующие списки свободных блоков.
    - Повторять процесс до достижения необходимого размера блока.
9. Выделение Блока: Удалить подходящий блок из списка свободных блоков и вернуть указатель на выделенную память.
10. Освобождение Памяти (my_free)
11. Проверка Параметров: Убедиться, что аллокатор и указатель валидны.
12. Добавление Блока в Свободный Список: Добавить освободенный блок в соответствующий список свободных блоков.
13. Попытка Объединения Блоков:
    - Найти "buddy" (двойника) текущего блока.
    - Если "buddy" свободен, удалить его из списка свободных блоков и объединить оба блока в один больший блок.
    - Добавить объединённый блок обратно в соответствующий список свободных блоков.
    - Повторять процесс до достижения максимального размера блока или отсутствия доступных "buddy".
14. Деинициализация Аллокатора (allocator_destroy)
15. Освобождение Памяти: Использовать системный вызов munmap для освобождения выделенного пула памяти.
16. Завершение Работы: При возникновении ошибки при освобождении, вывести сообщение об ошибке и завершить программу.

Код программы

blocks_allocator.c

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

#define MAX_BLOCK_SIZE_EXTENT 10
#define MIN_BLOCK_SIZE_EXTENT 0
#define NUM_LISTS 11

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;

void write_error(const char *msg) {
    write(STDERR_FILENO, msg, strlen(msg));
}

size_t power_of_two(int base, int exp) {
    size_t result = 1;
    for(int i = 0; i < exp; i++) {
        result *= base;
    }
    return result;
}

int get_index(size_t size) {
    int index = 0;
    size_t block_size = 1;
    while (block_size < size && index < NUM_LISTS -1) {
        block_size <<= 1;
        index++;
    }
    return index;
}

void add_to_free_list(Allocator *allocator, Block *block) {
    int index = get_index(block->size);
    block->next = allocator->freeLists[index];
    block->prev = NULL;
    if (allocator->freeLists[index] != NULL) {
        allocator->freeLists[index]->prev = block;
```

```
    }
    allocator->freeLists[index] = block;
}

void remove_from_free_list(Allocator *allocator, Block *block) {
    int index = get_index(block->size);
    if (block->prev != NULL) {
        block->prev->next = block->next;
    } else {
        allocator->freeLists[index] = block->next;
    }
    if (block->next != NULL) {
        block->next->prev = block->prev;
    }
    block->next = block->prev = NULL;
}

void split_block(Allocator *allocator, Block* block) {
    if (block->size <= power_of_two(2, MIN_BLOCK_SIZE_EXTENT)) return;

    size_t half_size = block->size / 2;
    Block *left = (Block *)((char *)block);
    Block *right = (Block *)((char *)block + half_size);

    left->size = half_size;
    left->next = left->prev = NULL;
    right->size = half_size;
    right->next = right->prev = NULL;

    remove_from_free_list(allocator, block);

    add_to_free_list(allocator, left);
    add_to_free_list(allocator, right);
}
```

```
void* allocator_create(void* mem, size_t size) {
    Allocator* allocator = (Allocator*)mem;
    allocator->total_size = size - sizeof(Allocator);
    allocator->memory = (char*)mem + sizeof(Allocator);

    for (int i = 0; i < NUM_LISTS; ++i) {
        allocator->freeLists[i] = NULL;
    }

    size_t initial_size = power_of_two(2, MAX_BLOCK_SIZE_EXTENT);
    if (initial_size > allocator->total_size) {
        initial_size = allocator->total_size;
    }

    Block *initial_block = (Block *)allocator->memory;
    initial_block->size = initial_size;
    initial_block->next = NULL;
    initial_block->prev = NULL;

    add_to_free_list(allocator, initial_block);

    return (void *)allocator;
}

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

    Allocator *allocator = (Allocator *)allocator_ptr;

    size_t actual_size = 1;
    while (actual_size < size) {
        actual_size <<= 1;
    }
```

```
int index = get_index(actual_size);
if (index >= NUM_LISTS) {
    write_error("ERROR: Requested size exceeds maximum block size\n");
    return NULL;
}

int current_index = index;
while (current_index < NUM_LISTS && allocator->freeLists[current_index] == NULL) {
    current_index++;
}

if (current_index == NUM_LISTS) {
    write_error("ERROR: No available blocks for allocation\n");
    return NULL;
}

while (current_index > index) {
    Block *block_to_split = allocator->freeLists[current_index];
    if (block_to_split == NULL) {
        write_error("ERROR: No block available to split\n");
        return NULL;
    }
    split_block(allocator, block_to_split);
    current_index--;
}

Block *allocated_block = allocator->freeLists[index];
if (allocated_block == NULL) {
    write_error("ERROR: Failed to allocate block\n");
    return NULL;
}
remove_from_free_list(allocator, allocated_block);

return (void *)allocated_block;
```

```
}

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

    Allocator *allocator = (Allocator *)allocator_ptr;
    Block *block = (Block*)ptr;
    add_to_free_list(allocator, block);

    while (block->size < power_of_two(2, MAX_BLOCK_SIZE_EXTENT)) {
        size_t size = block->size;
        size_t offset = (char*)block - (char*)allocator->memory;
        size_t buddy_offset = offset ^ size;
        Block *buddy = (Block*)((char*)allocator->memory + buddy_offset);

        int index = get_index(size);
        Block *current = allocator->freeLists[index];
        int found = 0;
        while (current != NULL) {
            if (current == buddy) {
                found = 1;
                break;
            }
            current = current->next;
        }

        if (!found) {
            break;
        }

        remove_from_free_list(allocator, buddy);

        Block *merged_block = (buddy < block) ? buddy : block;
```

```
        merged_block->size = size * 2;

        remove_from_free_list(allocator, block);

        block = merged_block;
        add_to_free_list(allocator, merged_block);
    }
}

void allocator_destroy(void *allocator_ptr, size_t size) {
    if (!allocator_ptr) {
        return;
    }

    Allocator *allocator = (Allocator *)allocator_ptr;
    void *full_memory = (char *)allocator->memory - sizeof(Allocator);

    if (munmap(full_memory, size) == -1) {
        const char msg[] = "ERROR: munmap failed\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        _exit(EXIT_FAILURE);
    }
}
```

buddy_allocator.c

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

typedef struct Block {
    size_t size;
    int free;
    struct Block *left;
    struct Block *right;
} Block;

typedef struct Allocator {
    Block *root;
```

```
    void *memory;
    size_t total_size;
    size_t offset;
} Allocator;

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

void write_error(const char *msg) {
    write(STDERR_FILENO, msg, strlen(msg));
}

size_t largest_power_of_two(size_t n) {
    size_t power = 1;
    while (power << 1 <= n) {
        power <<= 1;
    }
    return power;
}

Block *create_node(Allocator *allocator, size_t size) {
    if (allocator->offset + sizeof(Block) > allocator->total_size) {
        write_error("ERROR: Not enough memory to create a new block\n");
        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;
}

void* allocator_create(void *mem, size_t size) {
    if (!is_power_of_two(size)) {
        write_error("ERROR: Allocator initializes memory of size not power of two\n");
        return NULL;
    }

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

    size_t root_size = largest_power_of_two(allocator->total_size);
    allocator->root = create_node(allocator, root_size);
    if (!allocator->root) {
        write_error("ERROR: Unable to create root block\n");
```

```
        return NULL;
    }

    return (void *)allocator;
}

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

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

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

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

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

    Block *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(void *allocator_ptr, size_t size) {
    if (allocator_ptr == NULL || size == 0) {
        return NULL;
    }

    Allocator *allocator = (Allocator *)allocator_ptr;

    size_t power_size = 1;
    while (power_size < size) {
        power_size <<= 1;
    }
```

```
    Block *allocated_block = allocate_block(allocator, allocator->root, power_size);
    if (allocated_block == NULL) {
        return NULL;
    }

    return (void *)((char *)allocated_block + sizeof(Block));
}

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

    Allocator *allocator = (Allocator *)allocator_ptr;

    Block *node = (Block *)((char *)ptr - sizeof(Block));
    node->free = 1;

    while (node->left != NULL && node->left->free && node->right->free) {
        node->left = node->right = NULL;
        node->size *= 2;

        break;
    }
}

void allocator_destroy(void *allocator_ptr, size_t size) {
    if (!allocator_ptr) {
        return;
    }

    Allocator *allocator = (Allocator *)allocator_ptr;
    void *full_memory = (char *)allocator->memory - sizeof(Allocator);

    if (munmap(full_memory, size) == -1) {
        const char msg[] = "ERROR: munmap failed\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        _exit(EXIT_FAILURE);
    }
}
```

main.c

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

```
typedef struct AllocatorAPI {
    void* (*allocator_create)(void *mem, size_t size);
    void* (*my_malloc)(void *allocator, size_t size);
    void (*my_free)(void *allocator, void *ptr);
    void (*allocator_destroy)(void *allocator, size_t size);
} AllocatorAPI;

void write_message(int fd, const char *msg) {
    write(fd, msg, strlen(msg));
}

AllocatorAPI* load_allocator(const char *library_path) {
    if (library_path == NULL) {
        return NULL;
    }

    void *handle = dlopen(library_path, RTLD_LAZY);
    if (!handle) {
        write_message(STDERR_FILENO, "ERROR: Failed to load library\n");
        return NULL;
    }

    dlerror();

    AllocatorAPI *api = (AllocatorAPI *)mmap(NULL, sizeof(AllocatorAPI), PROT_READ |
PROT_WRITE,
                                             MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (api == MAP_FAILED) {
        write_message(STDERR_FILENO, "ERROR: mmap failed for AllocatorAPI\n");
        dlclose(handle);
        return NULL;
    }

    api->allocator_create = (void* (*)(void *, size_t))dlsym(handle, "allocator_create");
    api->my_malloc = (void* (*)(void *, size_t))dlsym(handle, "my_malloc");
    api->my_free = (void (*)(void *, void *))dlsym(handle, "my_free");
    api->allocator_destroy = (void (*)(void *, size_t))dlsym(handle, "allocator_destroy");

    char *error = dlerror();
    if (error != NULL) {
        write_message(STDERR_FILENO, "ERROR: Failed to load symbols from library\n");
        munmap(api, sizeof(AllocatorAPI));
        dlclose(handle);
        return NULL;
    }

    return api;
}

void* fallback_allocator_create(void *mem, size_t size) {
```

```
    (void)mem;
    (void)size;
    return NULL;
}

void* fallback_my_malloc(void *allocator, size_t size) {
    (void)allocator;
    return malloc(size);
}

void fallback_my_free(void *allocator, void *ptr) {
    (void)allocator;
    free(ptr);
}

void fallback_allocator_destroy(void *allocator, size_t size) {
    (void)allocator;
    (void)size;
}

AllocatorAPI* get_allocator_api(const char *library_path) {
    AllocatorAPI *api = load_allocator(library_path);
    if (api == NULL) {
        api = (AllocatorAPI *)mmap(NULL, sizeof(AllocatorAPI), PROT_READ | PROT_WRITE,
                                   MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
        if (api == MAP_FAILED) {
            write_message(STDERR_FILENO, "ERROR: mmap failed for fallback AllocatorAPI\n");
            return NULL;
        }
        api->allocator_create = fallback_allocator_create;
        api->my_malloc = fallback_my_malloc;
        api->my_free = fallback_my_free;
        api->allocator_destroy = fallback_allocator_destroy;
    }
    return api;
}

size_t addr_to_hex(char *buffer, size_t buffer_size, void *ptr) {
    const char *hex_chars = "0123456789abcdef";
    char temp[20];
    size_t len = 0;
    unsigned long addr = (unsigned long)ptr;

    temp[len++] = '0';
    temp[len++] = 'x';

    int started = 0;
    for(int i = sizeof(addr)*8 - 4; i >= 0; i -=4 ){
        unsigned long digit = (addr >> i) & 0xF;
        if (digit !=0 || started || i ==0) {
```

```
            temp[len++] = hex_chars[digit];
            started = 1;
        }
    }

    if (len >= buffer_size) {
        len = buffer_size -1;
    }

    memcpy(buffer, temp, len);
    buffer[len] = '\0';
    return len;
}

int test_allocator(const char *library_path) {
    AllocatorAPI *allocator_api = get_allocator_api(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) {
        write_message(STDERR_FILENO, "ERROR: mmap failed\n");
        munmap(allocator_api, sizeof(AllocatorAPI));
        return EXIT_FAILURE;
    }

    void *allocator = NULL;
    if (allocator_api->allocator_create != fallback_allocator_create) {
        allocator = allocator_api->allocator_create(addr, size);
        if (!allocator) {
            write_message(STDERR_FILENO, "ERROR: Failed to initialize allocator\n");
            munmap(addr, size);
            munmap(allocator_api, sizeof(AllocatorAPI));
            return EXIT_FAILURE;
        }
    } else {
        allocator = NULL;
    }

    write_message(STDOUT_FILENO, "==========Allocator
initialized==========\n");

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

    if (allocated_memory == NULL) {
        write_message(STDERR_FILENO, "ERROR: Memory allocation failed\n");
    } else {
```

```
        write_message(STDOUT_FILENO, "- Memory allocated successfully\n");
    }

    if (allocated_memory != NULL) {
        write_message(STDOUT_FILENO, "- Allocated memory contains: ");

        strcpy((char*)allocated_memory, "Hello, Allocator!\n");

        write(STDOUT_FILENO, allocated_memory, strlen((char*)allocated_memory));

        char buffer[64];
        size_t len = 0;
        const char *addr_msg = "- Allocated memory address: ";
        write(STDOUT_FILENO, addr_msg, strlen(addr_msg));

        len = addr_to_hex(buffer, sizeof(buffer), allocated_memory);
        write(STDOUT_FILENO, buffer, len);
        write(STDOUT_FILENO, "\n", 1);

        allocator_api->my_free(allocator, allocated_memory);
        write_message(STDOUT_FILENO, "- Memory freed\n");
    }

    if (allocator_api->allocator_destroy != fallback_allocator_destroy) {
        allocator_api->allocator_destroy(allocator, size);
    }

    munmap(allocator_api, sizeof(AllocatorAPI));
    munmap(addr, size);

    write_message(STDOUT_FILENO, "- Allocator
destroyed\n==========================================\n");
    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;
    } else {
        return EXIT_SUCCESS;
    }
}
```

Протокол работы программы

Тестирование:

```
wabisabi@wabisabi:/mnt/c/Users/zlata/OneDrive/Рабочий стол/labs/sofa/os_lab4$ gcc -fPIC -shared -o libblocks.so blocks_allocator.c
wabisabi@wabisabi:/mnt/c/Users/zlata/OneDrive/Рабочий стол/labs/sofa/os_lab4$ gcc -fPIC -shared -o libbuddy.so buddy_allocator.c
wabisabi@wabisabi:/mnt/c/Users/zlata/OneDrive/Рабочий стол/labs/sofa/os_lab4$ gcc -o allocator_main main.c -ldl
wabisabi@wabisabi:/mnt/c/Users/zlata/OneDrive/Рабочий стол/labs/sofa/os_lab4$ ./allocator_main ./libblocks.so
============Allocator initialized============
- Memory allocated successfully
- Allocated memory contains: Hello, Allocator!
- Allocated memory address: 0x7ff12ccac428
- Memory freed
- Allocator destroyed
=============================================
wabisabi@wabisabi:/mnt/c/Users/zlata/OneDrive/Рабочий стол/labs/sofa/os_lab4$ ./allocator_main ./libbuddy.so
============Allocator initialized============
- Memory allocated successfully
- Allocated memory contains: Hello, Allocator!
- Allocated memory address: 0x7f6c26373160
- Memory freed
- Allocator destroyed
=============================================
```

Strace:

```
strace ./allocator_main ./libbuddy.so
execve("./allocator_main", ["./allocator_main", "./libbuddy.so"],
0x7ffede7018b8 /* 35 vars */) = 0
brk(NULL)                               = 0x56538afee000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f02e05bf000
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=20167, ...}) = 0
mmap(NULL, 20167, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f02e05ba000
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) =
0x7f02e03a8000
mmap(0x7f02e03d0000, 1605632, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f02e03d0000
mmap(0x7f02e0558000, 323584, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x7f02e0558000
mmap(0x7f02e05a7000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x7f02e05a7000
mmap(0x7f02e05ad000, 52624, PROT_READ|PROT_WRITE,
```

```
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f02e05ad000
      close(3)                                  = 0
      mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7f02e03a5000
      arch_prctl(ARCH_SET_FS, 0x7f02e03a5740) = 0
      set_tid_address(0x7f02e03a5a10)           = 103639
      set_robust_list(0x7f02e03a5a20, 24)       = 0
      rseq(0x7f02e03a6060, 0x20, 0, 0x53053053) = 0
      mprotect(0x7f02e05a7000, 16384, PROT_READ) = 0
      mprotect(0x56536dffc000, 4096, PROT_READ) = 0
      mprotect(0x7f02e05f7000, 8192, PROT_READ) = 0
      prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0
      munmap(0x7f02e05ba000, 20167)             = 0
      getrandom("\x55\x30\xde\x41\x9c\xb8\x75\x4d", 8, GRND_NONBLOCK) = 8
      brk(NULL)                                 = 0x56538afee000
      brk(0x56538b00f000)                       = 0x56538b00f000
      openat(AT_FDCWD, "./libbuddy.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\0"..., 832) = 832
      fstat(3, {st_mode=S_IFREG|0777, st_size=16208, ...}) = 0
      getcwd("/mnt/c/Users/zlata/OneDrive/\320\240\320\260\320\261\320\276\321\2
07\320\270\320\271 \321\201\321\202\320\276\320\273/labs/sofa/os_lab4", 128) =
70
      mmap(NULL, 16488, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f02e05ba000
      mmap(0x7f02e05bb000, 4096, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7f02e05bb000
      mmap(0x7f02e05bc000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x2000) = 0x7f02e05bc000
      mmap(0x7f02e05bd000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f02e05bd000
      close(3)                                  = 0
      mprotect(0x7f02e05bd000, 4096, PROT_READ) = 0
      mmap(NULL, 32, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f02e03a4000
      mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f02e03a3000
      write(1, "=============Allocator initializ"..., 48=============Allocator
initialized=============
      ) = 48
      write(1, "- Memory allocated successfully\n", 32- Memory allocated
successfully
      ) = 32
      write(1, "- Allocated memory contains: ", 29- Allocated memory contains: )
= 29
      write(1, "Hello, Allocator!\n", 18Hello, Allocator!
      )     = 18
      write(1, "- Allocated memory address: ", 28- Allocated memory address: ) =
28
      write(1, "0x7f02e03a3160", 140x7f02e03a3160)          = 14
      write(1, "\n", 1
      )                         = 1
      write(1, "- Memory freed\n", 15- Memory freed
```

```
)        = 15
munmap(0x7f02e03a3000, 4096)            = 0
munmap(0x7f02e03a4000, 32)              = 0
munmap(0x7f02e03a3000, 4096)            = 0
write(1, "- Allocator destroyed\n==========..., 70- Allocator destroyed
==============================================
) = 70
exit_group(0)                           = ?
+++ exited with 0 +++
```

Вывод

В данной программе реализованы два алгоритма управления памятью: Блоки по 2^n и. Оба аллокатора используют системные вызовы mmap и munmap для управления пулом памяти, а также функции write для вывода информационных и ошибочных сообщений, исключая использование библиотеки <stdio.h>.

алгоритм двойников ориентирован на дерево блоков, что обеспечивает эффективное управление памятью с низкой степенью фрагментации.

Аллокатор с Блоками Размеров 2^n использует массив списков свободных блоков для каждого возможного размера, что упрощает поиск и управление блоками. Этот аллокатор демонстрирует корректное выделение и освобождение памяти без возникновения рекурсии.

Сравнение двух алгоритмов показало, что алгоритм двойников более эффективно справляется с уменьшением фрагментации за счёт структуры дерева блоков, но требует более сложного управления метаданными. Аллокатор с блоками размеров 2^n проще в реализации и управлении, но может страдать от более высокой степени внутренней фрагментации.

Использование системных вызовов mmap, munmap, write, dlopen, dlsym и _exit обеспечило эффективное взаимодействие программы с операционной системой, а также динамическую загрузку и управление аллокаторами памяти.

.