

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: Шведов А.И.

Преподаватель: Бахарев В.Д.

Оценка: \_\_\_\_\_

Дата: 20.12.24

Москва, 2024

# Постановка задачи

## Вариант 7.

Требуется создать две динамические библиотеки, реализующие два аллокатора: списки свободных блоков (наиболее подходящее) и блоки по  $2^n$ .

## Общий метод и алгоритм решения

Использованные системные вызовы:

1. **\*int munmap(void addr, size\_t length);** - Удаляет отображения, созданные с помощью mmap.
2. **\*int dlclose(void handle);** - Закрывает динамическую библиотеку, открытую с помощью dlopen, и освобождает ресурсы, связанные с этим дескриптором.
3. **\*\*void dlopen(const char filename, int flag);** - Открывает динамическую библиотеку и возвращает дескриптор для последующего использования.
4. **\*\*void mmap(void addr, size\_t length, int prot, int flags, int fd, off\_t offset);** – создает новое отображение памяти или изменяет существующее.
5. **int write(int \_Filehandle, const void \*\_Buf, unsigned int \_MaxCharCount)** – выводит информацию в Filehandle.

## Описание программы

### 1. main.c

Открывает динамические библиотеки и получает нужные функции. Если в библиотеке не нашлось нужных функций, то вместо них будут использоваться аварийные оберточные функции. Далее как пример функция выделяет и освобождает память массива.

### 2. degree2.c

Файл в котором реализована логика работы аллокатора блоками по  $2^n$ .

- 1) Вся память при инициализации разбивается на блоки которые равны степени двойки.
- 1) Все блоки хранятся в списке свободных элементов.
- 2) Каждый блок хранит указатель на следующий свободный блок.
- 3) При освобождении нужно добавить этот блок в список свободных элементов в нужную позицию.
- 4) Для выделения памяти выбираем блок  $N[\log_2(\text{size})]$  и возвращаем указатель на первый элемент, помечая блок занятым.

### 3. List\_allocator.c

Файл в котором реализована логика работы аллокатора на списках свободных блоков (наиболее подходящее).

- 1) Все свободные блоки организованы в список
- 2) В блоке хранится его размер и указатель на следующий свободный блок
- 3) Для выделения памяти проходимся по всему списку свободных блоков и выбираем минимальный блок, который больше или равен по размеру нужного блока. Помечаем блок, как занятый и убираем из списка.
- 4) При освобождении памяти возвращаем блок в список свободных элементов. И при возможности сливаем рядом стоящие блоки.

## Код программы

### main.c

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

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

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

void *standard_allocator_alloc(void *allocator, size_t size) {
    (void)allocator;
    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 standard_allocator_free(void *allocator, void *memory) {
    (void)allocator;
    if (memory == NULL) return;
    uint32_t *mem = (uint32_t *)memory - 1;
    munmap(mem, *mem);
}

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

void load_allocator(const char *library_path, Allocator *allocator) {
    void *library = dlopen(library_path, RTLD_LOCAL | RTLD_NOW);
    if (library_path == NULL || library_path[0] == '\\0' || !library) {
        char message[] = "WARNING: failed to load shared library\n";
        write(STDERR_FILENO, message, sizeof(message) - 1);
        allocator->allocator_create = standard_allocator_create;
    }
}
```

```

    allocator->allocator_alloc = standard_allocator_alloc;
    allocator->allocator_free = standard_allocator_free;
    allocator->allocator_destroy = standard_allocator_destroy;
    return;
}

allocator->allocator_create = dlsym(library, "allocator_create");
allocator->allocator_alloc = dlsym(library, "allocator_alloc");
allocator->allocator_free = dlsym(library, "allocator_free");
allocator->allocator_destroy = dlsym(library, "allocator_destroy");

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

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

    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[] = "mmap failed\n";
        write(STDERR_FILENO, message, sizeof(message) - 1);
        return EXIT_FAILURE;
    }

    void *allocator = allocator_api.allocator_create(addr, size);

    if (!allocator) {
        char message[] = "Failed to initialize allocator\n";
        write(STDERR_FILENO, message, sizeof(message) - 1);
        munmap(addr, size);
        return EXIT_FAILURE;
    }

    void *blocks[10];
    size_t block_sizes[12] = {12, 13, 13, 24, 40, 56, 100, 120, 400, 120, 120, 120};

    int alloc_failed = 0;
    for (int i = 0; i < 12; ++i) {
        blocks[i] = allocator_api.allocator_alloc(allocator, block_sizes[i]);
        if (blocks[i] == NULL) {
            alloc_failed = 1;
            char alloc_fail_message[] = "Memory allocation failed\n";
            write(STDERR_FILENO, alloc_fail_message,
                  sizeof(alloc_fail_message) - 1);

```

```

        break;
    }
}

if (!alloc_failed) {
    char alloc_success_message[] = "Memory allocated successfully\n";
    write(STDOUT_FILENO, alloc_success_message,
        sizeof(alloc_success_message) - 1);

    for (int i = 0; i < 12; ++i) {
        char buffer[64];
        snprintf(buffer, sizeof(buffer), "Block %d address: %p\n", i + 1,
            blocks[i]);
        write(STDOUT_FILENO, buffer, strlen(buffer));
    }
}

for (int i = 0; i < 12; ++i) {
    if (blocks[i] != NULL)
        allocator_api.allocator_free(allocator, blocks[i]);
}

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

allocator_api.allocator_destroy(allocator);

char exit_message[] = "Program exited successfully\n";
write(STDOUT_FILENO, exit_message, sizeof(exit_message) - 1);

return EXIT_SUCCESS;
}

```

## list\_allocator.c

```

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

#define MIN_BLOCK_SIZE 16

// Структура для метаданных блока
typedef struct BlockHeader {
    size_t size; // Размер блока данных
    struct BlockHeader *next; // Указатель на следующий блок
    bool is_free; // Флаг, указывающий, свободен ли блок
} BlockHeader;

```

```

// Структура аллокатора
typedef struct Allocator {
    BlockHeader *free_list; // Указатель на начало списка свободных блоков
    void *memory_start;     // Начало выделенной памяти
    size_t total_size;      // Общий размер выделенной памяти
    void *base_addr;        // Указатель на начало выделенной памяти mmap
} Allocator;

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

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

    // Инициализируем метаданные первого блока
    allocator->free_list->size = allocator->total_size - sizeof(BlockHeader);
    allocator->free_list->next = NULL;
    allocator->free_list->is_free = true;

    return allocator;
}

// Функция выделения памяти
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;

    BlockHeader *best_fit = NULL;
    BlockHeader *prev_best = NULL;
    BlockHeader *current = allocator->free_list;
    BlockHeader *prev = NULL;

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

    if (best_fit) {

```

```

        size_t remaining_size = best_fit->size - size;
        if (remaining_size >= sizeof(BlockHeader) + MIN_BLOCK_SIZE) {
            BlockHeader *new_block =
                (BlockHeader *)((char *)best_fit + sizeof(BlockHeader) +
                                size);
            new_block->size = remaining_size - sizeof(BlockHeader);
            new_block->is_free = true;
            new_block->next = best_fit->next;

            best_fit->next = new_block;
            best_fit->size = size;
        }

        best_fit->is_free = false;
        if (prev_best == NULL) {
            allocator->free_list = best_fit->next;
        } else {
            prev_best->next = best_fit->next;
        }

        return (void *)((char *)best_fit + sizeof(BlockHeader));
    }

    return NULL;
}

// Функция освобождения памяти
void allocator_free(Allocator *allocator, void *ptr) {
    if (!allocator || !ptr) {
        return;
    }

    BlockHeader *header = (BlockHeader *)((char *)ptr - sizeof(BlockHeader));
    if (!header) return;
    header->is_free = true;

    // Добавляем в начало списка свободных
    header->next = allocator->free_list;
    allocator->free_list = header;

    // Пытаемся слить соседние блоки
    BlockHeader *current = allocator->free_list;
    while (current && current->next) {
        if (((char *)current + sizeof(BlockHeader) + current->size) ==
            (char *)current->next) {
            current->size += current->next->size + sizeof(BlockHeader);
            current->next = current->next->next;
        } else {
            current = current->next;
        }
    }
}

// Функция уничтожения аллокатора

```

```

void allocator_destroy(Allocator *allocator) {
    if (allocator) {
        munmap(allocator->base_addr, allocator->total_size + sizeof(Allocator));
    }
}

```

## degree2.c

```

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

#define MIN_BLOCK_SIZE 16 // Минимальный размер блока

int log2s(int n) {
    if (n == 0) {
        return -1;
    }
    int result = 0;
    while (n > 1) {
        n >>= 1;
        result++;
    }
    return result;
}

typedef struct BlockHeader {
    struct BlockHeader *next;
} BlockHeader;

// Структура аллокатора
typedef struct Allocator {
    BlockHeader **free_lists;
    size_t num_lists;
    void *base_addr;
    size_t total_size;
} Allocator;

Allocator *allocator_create(void *memory, size_t size) {
    if (!memory || size < sizeof(Allocator)) {
        return NULL;
    }
    Allocator *allocator = (Allocator *)memory;
    allocator->base_addr = memory;
    allocator->total_size = size;

    size_t min_usable_size = sizeof(BlockHeader) + MIN_BLOCK_SIZE;

```



```

size_t max_block_size = (size < 32) ? 32 : size;

allocator->num_lists = (size_t)floor(log2s(max_block_size) / 2) + 3;
allocator->free_lists =
    (BlockHeader **)((char *)memory + sizeof(Allocator));

for (size_t i = 0; i < allocator->num_lists; i++) {
    allocator->free_lists[i] = NULL;
}

void *current_block = (char *)memory + sizeof(Allocator) +
    allocator->num_lists * sizeof(BlockHeader *);
size_t remaining_size =
    size - sizeof(Allocator) - allocator->num_lists * sizeof(BlockHeader *);

size_t block_size = MIN_BLOCK_SIZE;
while (remaining_size >= min_usable_size) {
    if (block_size > remaining_size) {
        break;
    }

    if (block_size > max_block_size) {
        break;
    }

    if (remaining_size >= (block_size + sizeof(BlockHeader)) * 2) {
        for (int i = 0; i < 2; i++) {
            BlockHeader *header = (BlockHeader *)current_block;
            size_t index = (size == 0) ? 0 : (size_t)log2s(block_size);
            header->next = allocator->free_lists[index];
            allocator->free_lists[index] = header;

            current_block = (char *)current_block + block_size;
            remaining_size -= block_size;
        }
    } else {
        BlockHeader *header = (BlockHeader *)current_block;
        size_t index = (size == 0) ? 0 : (size_t)log2s(block_size);
        header->next = allocator->free_lists[index];
        allocator->free_lists[index] = header;

        current_block = (char *)current_block + remaining_size;
        remaining_size = 0;
    }

    block_size <<= 1;
}

return allocator;
}

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

```

```

}

size_t index = (size == 0) ? 0 : log2s(size) + 1;
if (index >= allocator->num_lists) {
    index = allocator->num_lists;
}
bool flag = false;
if (allocator->free_lists[index] == NULL) {
    while (index <= allocator->num_lists) {
        if (allocator->free_lists[index] != NULL) {
            flag = true;
            break;
        } else {
            ++index;
        }
    }
    if (!flag) return NULL;
}

BlockHeader *block = allocator->free_lists[index];
allocator->free_lists[index] = block->next;

return (void *)((char *)block + sizeof(BlockHeader));
}

// Функция освобождения памяти
void allocator_free(Allocator *allocator, void *ptr) {
    if (!allocator || !ptr) {
        return;
    }

    BlockHeader *block = (BlockHeader *)((char *)ptr - sizeof(BlockHeader));
    size_t temp_size =
        (char *)block + sizeof(BlockHeader) - (char *)allocator->base_addr;
    size_t temp = 32;

    while (temp <= temp_size) {
        size_t next_size = temp << 1;
        if (next_size > temp_size) {
            break;
        }
        temp = next_size;
    }

    size_t index = (temp_size == 0) ? 0 : (size_t)log2s(temp);
    if (index >= allocator->num_lists) {
        index = allocator->num_lists - 1;
    }

    block->next = allocator->free_lists[index];
    allocator->free_lists[index] = block;
}

// Функция уничтожения аллокатора

```

```
void allocator_destroy(Allocator *allocator) {  
    if (allocator) {  
        munmap(allocator->base_addr, allocator->total_size);  
    }  
}
```

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

### **1. Аллокатор блоками по $2^n$ :**

#### **Скорость выделения блоков:**

Быстрый поиск подходящего блока: Индекс списка свободных блоков вычисляется на основе  $\log_2(\text{size})$ .

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

#### **Скорость освобождения блоков:**

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

Размер блока определяется по адресу, а затем  $\log_2$ , что также очень быстро.

#### **Простота использования:**

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

### **2. Аллокатор на связном списке с выбором наиболее подходящего участка памяти (best-fit)**

#### **Скорость выделения блоков:**

Выбирается наилучший блок: При поиске выбирается блок, размер которого наиболее близок к запрошенному, что минимизирует внутреннюю фрагментацию.

Требуется просмотр всего списка свободных блоков: Поиск “лучшего” блока может занять значительное время, так как нужно просмотреть весь связный список свободных блоков.

#### **Скорость освобождения блоков:**

При освобождении можно объединять освобождаемый блок с соседними, что уменьшает фрагментацию.

Если слияние не требуется, то освобождение происходит за константу.

#### **Простота использования:**

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

## Strace:

```
execve("./main", [ "./main", "./degree2.so" ], 0x7ffe92139060 /* 27 vars */) = 0

brk(NULL)                               = 0x562632b20000

arch_prctl(0x3001 /* ARCH_??? */, 0x7ffeaff466a0) = -1 EINVAL (Invalid argument)

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7f361a403000

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

newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=37379, ...}, AT_EMPTY_PATH) = 0

mmap(NULL, 37379, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f361a3f9000

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\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) =
0x7f361a1d0000

mprotect(0x7f361a1f8000, 2023424, PROT_NONE) = 0

mmap(0x7f361a1f8000, 1658880, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f361a1f8000

mmap(0x7f361a38d000, 360448, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7f361a38d000

mmap(0x7f361a3e6000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x7f361a3e6000

mmap(0x7f361a3ec000, 52816, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f361a3ec000

close(3)                                = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7f361a1cd000
```

```

arch_prctl(ARCH_SET_FS, 0x7f361a1cd740) = 0
set_tid_address(0x7f361a1cda10)      = 40532
set_robust_list(0x7f361a1cda20, 24)  = 0
rseq(0x7f361a1ce0e0, 0x20, 0, 0x53053053) = 0
mprotect(0x7f361a3e6000, 16384, PROT_READ) = 0
mprotect(0x5625f727d000, 4096, PROT_READ) = 0
mprotect(0x7f361a43d000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f361a3f9000, 37379)        = 0
getrandom("\x79\xc3\x82\x52\xdc\xc4\x6a\x92", 8, GRND_NONBLOCK) = 8
brk(NULL)                            = 0x562632b20000
brk(0x562632b41000)                  = 0x562632b41000
openat(AT_FDCWD, "./degree2.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\0\0\0\0\0\0"..., 832) = 832
newfstatat(3, "", {st_mode=S_IFREG|0777, st_size=15744, ...}, AT_EMPTY_PATH) = 0
getcwd("/mnt/c/Users/mrshv/OneDrive/Desktop/Os/4", 128) = 41
mmap(NULL, 16440, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f361a3fe000
mmap(0x7f361a3ff000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7f361a3ff000
mmap(0x7f361a400000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f361a400000
mmap(0x7f361a401000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f361a401000
close(3)                            = 0
mprotect(0x7f361a401000, 4096, PROT_READ) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f361a43c000
write(1, "Memory allocated successfully\n", 30Memory allocated successfully
) = 30
write(1, "Block 1 address: 0x7f361a43c080\n", 32Block 1 address: 0x7f361a43c080

```

) = 32

write(1, "Block 2 address: 0x7f361a43c070\n", 32Block 2 address: 0x7f361a43c070

) = 32

write(1, "Block 3 address: 0x7f361a43c650\n", 32Block 3 address: 0x7f361a43c650

) = 32

write(1, "Block 4 address: 0x7f361a43c0b0\n", 32Block 4 address: 0x7f361a43c0b0

) = 32

write(1, "Block 5 address: 0x7f361a43c110\n", 32Block 5 address: 0x7f361a43c110

) = 32

write(1, "Block 6 address: 0x7f361a43c0d0\n", 32Block 6 address: 0x7f361a43c0d0

) = 32

write(1, "Block 7 address: 0x7f361a43c1d0\n", 32Block 7 address: 0x7f361a43c1d0

) = 32

write(1, "Block 8 address: 0x7f361a43c150\n", 32Block 8 address: 0x7f361a43c150

) = 32

write(1, "Block 9 address: 0x7f361a43c650\n", 32Block 9 address: 0x7f361a43c650

) = 32

write(1, "Block 10 address: 0x7f361a43c350"..., 33Block 10 address: 0x7f361a43c350

) = 33

write(1, "Block 11 address: 0x7f361a43c250"..., 33Block 11 address: 0x7f361a43c250

) = 33

write(1, "Block 12 address: 0x7f361a43c450"..., 33Block 12 address: 0x7f361a43c450

) = 33

write(1, "Memory freed\n", 13Memory freed

) = 13

**munmap(0x7f361a43c000, 4096) = 0**

write(1, "Program exited successfully\n", 28Program exited successfully

) = 28

exit\_group(0) = ?

+++ exited with 0 +++

## **Вывод**

В рамках лабораторной работы была разработана программа, демонстрирующая работу аллокатора передаваемого в качестве аргумента при вызове программы. Было реализовано 2 аллокатора и проведена работа по сравнении их работоспособности.