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.

1. 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.
    2. Все блоки хранятся в списке свободных элементов.
    3. Каждый блок хранит указатель на следующий свободный блок.
    4. При освобождении нужно добавить этот блок в список свободных элементов в нужную позицию.
    5. Для выделения памяти выбираем блок N[log2(size)] и возвращаем указатель на первый элемент, помечая блок занятым.

1. List\_allocator.c

Файл в котором реализована логика работы аллокатора на спискач 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 log2(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 log2, 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\0\0\3\0>\0\1\0\0\0P\237\2\0\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

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\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\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\0\3\0>\0\1\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 сравнениб их работоспособности.