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: Иванченко Макар Дмитриевич

Преподаватель: Бахарев В.Д. (ФИИТ)

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 12.12.24

Москва, 2024

**Постановка задачи**

**Постановка задачи**

**Вариант 3.**

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]).

Списки свободных блоков (первое подходящее) и алгоритм двойников.

**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**

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. Buddy\_allocator.c

В файле реализован аллокатор при помощи метода двойников.

* + 1. При запросе памяти размер округляется до степени двойки
    2. Аллокатор делит свободную память на две равные части, и продолжает так до тех пор пока не найдет блок искомого размера
    3. При освобождении памяти проверяется, свободен ли близнец освобождаемого блока памяти, и если он свободен, то блоки рекурсивно сливаются

1. List\_allocator.c

Файл в котором реализована логика работы аллокатора на спискач свободных блоков (первое 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 первый блок, размер которого больше требуемого размера.
    4. При освобождении памяти возвращаем блок в список свободных элементов. И при возможности сливаем рядом стоящие блоки.

**Код программы**

**Allocators.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>

#include <bits/mman-linux.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 = (uint32\_t\*)

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 = (void\*(\*)(void\*, size\_t size))(dlsym(library, "allocator\_create"));

allocator->allocator\_alloc = (void\*(\*)(void\*, size\_t size))(dlsym(library, "allocator\_alloc"));

allocator->allocator\_free = (void(\*)(void\*, void\* size))(dlsym(library, "allocator\_free"));

allocator->allocator\_destroy = (void(\*)(void\*))(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) {

printf("1g");

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 };

return 0;

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;

}

Buddy\_allocator.c

#include <dlfcn.h>

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <stddef.h>

#include <stdbool.h>

#include <string.h>

#define MIN\_SIZE\_BLOCK 32

typedef struct BlockHeader {

size\_t size;

struct BlockHeader\* next;

struct BlockHeader\* buddy;

bool is\_free;

} BlockHeader;

typedef struct Allocator {

void\* memory;

size\_t size;

uint8\_t\* bitmap;

size\_t block\_size;

BlockHeader\*\* free\_lists;

size\_t max\_pow;

size\_t total\_free\_lists\_size;

} Allocator;

size\_t log2s(size\_t n) {

if (n == 0) {

return -1;

}

size\_t result = 0;

while (n > 1) {

n >>= 1;

result++;

}

return result;

}

Allocator\* allocator\_create(void\* const memory, const size\_t size) {

if (!memory || size == 0) return NULL;

Allocator\* allocator = (Allocator\*)memory;

allocator->memory = (void\*)((char\*)memory + sizeof(Allocator));

allocator->size = size - sizeof(Allocator);

allocator->block\_size = 32;

allocator->bitmap = (uint8\_t\*)allocator->memory;

size\_t bitmap\_size = allocator->size / allocator->block\_size / 8;

memset(allocator->bitmap, 0, bitmap\_size);

allocator->memory = (uint8\_t\*)allocator->bitmap + bitmap\_size;

allocator->free\_lists = (BlockHeader\*\*)allocator->memory;

size\_t max\_pow = 0;

size\_t cur = MIN\_SIZE\_BLOCK;

while (cur < size) {

cur \*= 2;

max\_pow++;

}

allocator->max\_pow = max\_pow;

allocator->total\_free\_lists\_size = (max\_pow + 1) \* sizeof(BlockHeader);

memset(allocator->free\_lists, 0, (max\_pow + 1) \* sizeof(BlockHeader));

allocator->free\_lists[max\_pow]->is\_free = true;

allocator->free\_lists[max\_pow]->size = size;

allocator->free\_lists[max\_pow]->next = NULL;

return allocator;

}

void allocator\_destroy(Allocator\* const allocator) {

if (allocator) {

munmap(allocator->memory, allocator->size + sizeof(Allocator));

}

}

void add\_to\_list(BlockHeader\*\* free\_list\_first, BlockHeader\* to\_insert) {

BlockHeader\* current = \*free\_list\_first;

BlockHeader\* prev = NULL;

if (current->next == NULL) {

if (to\_insert > current) {

current->next = to\_insert;

to\_insert->next = current->next;

} else {

\*free\_list\_first = to\_insert;

(\*free\_list\_first)->next = current;

}

} else {

while (current->next != NULL) {

if (current->next > to\_insert) {

BlockHeader\* tmp = current->next;

current->next = to\_insert;

to\_insert->next = tmp;

break;

}

current = current->next;

}

if (current->next == NULL) {

current->next = to\_insert;

to\_insert->next = NULL;

}

}

}

size\_t round\_to\_2\_pow(size\_t num) {

size\_t cur = MIN\_SIZE\_BLOCK;

while (num > cur) {

cur \*= 2;

}

return cur;

}

void\* allocator\_alloc(Allocator\* const allocator, size\_t size) {

if (!allocator || size == 0 || size > allocator->size) return NULL;

size = round\_to\_2\_pow(size);

size\_t pow = (size\_t)log2s(size);

size\_t min\_exist\_pow = pow;

while (allocator->free\_lists[min\_exist\_pow] == NULL) {

min\_exist\_pow++;

if (min\_exist\_pow == allocator->max\_pow) {

return NULL;

}

}

BlockHeader\* result\_block;

BlockHeader\* cur = NULL;

BlockHeader\* block1 = allocator->free\_lists[min\_exist\_pow];

BlockHeader\* block2;

while (min\_exist\_pow != pow) {

cur = allocator->free\_lists[min\_exist\_pow];

allocator->free\_lists[min\_exist\_pow] = allocator->free\_lists[min\_exist\_pow]->next;

block1 = allocator->free\_lists[min\_exist\_pow] + sizeof(BlockHeader);

block2 = allocator->free\_lists[min\_exist\_pow] + sizeof(BlockHeader) + allocator->free\_lists[min\_exist\_pow]->size / 2 + sizeof(BlockHeader);

block1->size = allocator->free\_lists[min\_exist\_pow]->size / 2;

block1->is\_free = true;

block2->size = allocator->free\_lists[min\_exist\_pow]->size / 2;

block2->is\_free = true;

min\_exist\_pow--;

add\_to\_list(&(allocator->free\_lists[min\_exist\_pow]), block1);

add\_to\_list(&(allocator->free\_lists[min\_exist\_pow]), block2);

}

block1 = allocator->free\_lists[min\_exist\_pow];

allocator->free\_lists[min\_exist\_pow] = allocator->free\_lists[min\_exist\_pow]->next;

block1->is\_free = false;

return (char\*)allocator->free\_lists[min\_exist\_pow] + sizeof(BlockHeader);

}

void allocator\_free(Allocator\* const allocator, void\* const memory) {

BlockHeader\* cur = (BlockHeader\*)((char\*)memory - sizeof(BlockHeader));

while (true) {

cur->is\_free = true;

size\_t pow = log2s(cur->size);

BlockHeader\* current = allocator->free\_lists[pow];

BlockHeader\* prev = NULL;

bool has\_merged = false;

while (current != NULL) {

if (current->size == cur->size) {

if (prev == NULL) {

allocator->free\_lists[pow] = allocator->free\_lists[pow]->next;

} else {

prev->next = current->next;

}

BlockHeader\* new\_block = current < cur ? current : cur;

new\_block->is\_free = true;

new\_block->size = current->size \* 2;

has\_merged = true;

cur = new\_block;

break;

}

current = current->next;

}

if (!has\_merged) {

break;

}

}

}

**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 "list\_allocator.h"

#include <unistd.h>

#define MIN\_BLOCK\_SIZE 16

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\_first = (BlockHeader\*)allocator->memory\_start;

allocator->free\_list\_first->size = allocator->total\_size - sizeof(BlockHeader);

allocator->free\_list\_first->next = NULL;

allocator->free\_list\_first->is\_free = true;

return allocator;

}

void\* allocator\_alloc(Allocator\* allocator, size\_t size) {

if (!allocator || size == 0 || size > allocator->total\_size) {

return NULL;

}

size = (size + MIN\_BLOCK\_SIZE - 1) / MIN\_BLOCK\_SIZE \* MIN\_BLOCK\_SIZE;

BlockHeader\* prev = NULL;

BlockHeader\* current = allocator->free\_list\_first;

while (current != NULL) {

if (current->size >= size) {

current->is\_free = false;

if (current->size == size) {

if (prev != NULL) {

prev->next = current->next;

} else {

allocator->free\_list\_first = allocator->free\_list\_first->next;

}

current->size = size;

} else {

size\_t remain\_size = current->size - size;

current->size = size;

if (remain\_size > MIN\_BLOCK\_SIZE + sizeof(BlockHeader)) {

BlockHeader\* new\_block = (BlockHeader\*)((char\*)current + sizeof(BlockHeader) + size);

new\_block->size = remain\_size - sizeof(BlockHeader);

new\_block->is\_free = true;

new\_block->next = current->next;

current->next = new\_block;

if (prev != NULL) {

prev->next = current->next;

} else {

allocator->free\_list\_first = allocator->free\_list\_first->next;

}

}

}

break;

}

current = current->next;

}

return (char\*)current + sizeof(BlockHeader);

}

void allocator\_free(Allocator\* allocator, void\* ptr) {

if (!allocator || !ptr) {

return;

}

BlockHeader\* block\_header\_to\_free = (BlockHeader\*)((char\*)ptr - sizeof(BlockHeader));

if (!block\_header\_to\_free) return;

block\_header\_to\_free->is\_free = true;

BlockHeader\* current = allocator->free\_list\_first;

BlockHeader\* prev = NULL;

if (current->next == NULL) {

if (block\_header\_to\_free > current) {

current->next = block\_header\_to\_free;

block\_header\_to\_free->next = current->next;

} else {

allocator->free\_list\_first = block\_header\_to\_free;

allocator->free\_list\_first->next = current;

}

} else {

while (current->next != NULL) {

if (current->next > block\_header\_to\_free) {

BlockHeader\* tmp = current->next;

current->next = block\_header\_to\_free;

block\_header\_to\_free->next = tmp;

break;

}

current = current->next;

}

if (current->next == NULL) {

current->next = block\_header\_to\_free;

block\_header\_to\_free->next = NULL;

}

}

current = allocator->free\_list\_first;

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;

}

}

current = allocator->free\_list\_first;

while (current != NULL) {

memset((char\*)(current) + sizeof(BlockHeader), '0', current->size);

current = current->next;

}

}

void allocator\_destroy(Allocator\* allocator) {

if (allocator) {

munmap(allocator->base\_addr, allocator->total\_size + sizeof(Allocator));

}

}

**Сравнение алгоритмов аллокаторов: списки свободных блоков (первое подходящее) и алгоритм близнецов**

**1. Алгоритм близнецов**

**Скорость выделения блоков:**

Если блок нужного размера уже существует, он просто выделяется, что делает операцию выделения **быстрой**. Однако если подходящего блока нет, то процесс может включать разделение более крупных блоков, что может занять дополнительное время.

**Скорость освобождения блоков:**

Низкая, так как алгориитм каждый раз пытается слить блок со своим близнецом. Если операции выделения и освобождения будут чередоваться, то это рпиведет к еще большему падению производительности.

**Простота использования:**

Довольно гибкий и удобный, однако имеет несколько недостатков. Во-первых, может выделять только память, размер которой является степенью двойки. Во-вторых, не поддерживает частичное освобождение блока, только целиком.

**2. Аллокатор на связном списке с выбором первого подходящего участка**

**Скорость выделения блоков:**

Выбирается первый подходящий блок, поэтому скорость высокая.

**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 небольшой

**Протокол работы программы**

**Тестирование:**

**Strace:**

execve("./main", ["./main", "./buddy.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, "./buddy.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.