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: Аксельрод А.М.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 06.01.25

Москва, 2025

Постановка задачи

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.

Каждый аллокатор должен обладать следующим интерфейсом (могут быть отличия в зависимости от особенностей алгоритма):

- Allocator* allocator_create(void *const memory, const size_t size) (инициализация аллокатора на памяти memory размера size);
- void allocator_destroy(Allocator *const allocator) (деинициализация структуры аллокатора);
- void* allocator_alloc(Allocator *const allocator, const size_t size) (выделение памяти аллокатором памяти размера size);
- void allocator_free(Allocator *const allocator, void *const memory) (возвращает выделенную память аллокатору);

Вариант 5. Алгоритм Мак-Кьюзика-Кэрелса и алгоритм двойников;

Общий метод и алгоритм решения

Использованные системные вызовы:

- void* dlopen(const char* filename, int flag); Загружает динамическую библиотеку и возвращает указатель на её начало.
- int dlclose(void* handle); Уменьшает счётчик ссылок на указатель динамической библиотеки.
- void* mmap(void* start, size_t length, int prot, int flags, int fd, off_t offset); Отражает length байтов, начиная со смещения offset файла в память, начиная с адреса start.
- int munmap(void* start, size_t length); Освобождает память.

Реализовано два аллокатора:

1. allocator1.c – Мак-Кьюзика-Кэрелса.
2. allocator2.c – алгоритм двойников.

Оба используют одинаковый интерфейс.

В алгоритме МКК память делится на блоки фиксированных размеров, равных степени 2. Создаётся зона, где хранятся блоки одного размера. В каждой зоне создаётся список свободных блоков памяти. Устанавливается минимальный и максимальный размеры блоков. При выделении памяти размер запроса округляется до ближайшей степени двойки, и в соответствующей зоне ищется свободный блок. При освобождении памяти блок возвращается в список свободных блоков.

В алгоритме двойников память также делится на большие блоки размеров степени 2. Если блок слишком большой для запроса, он делится на два блока с размером в два раза меньше(двойники), пока он не станет достаточно маленьким для запроса. Когда блок освобождается, он ищет свободного двойника и объединяется с ним.

Сравнение алгоритмов по времени:

Число байт	Действие	Системный аллокатор(сек)	Алгоритм МКК(сек)	Алгоритм двойников(сек)
10	Выделение	0.000013	0.000003	0.000002
	Очистка	0.000032	0.000020	0.000001
1585	Выделение	0.000009	0.000003	0.000002
	Очистка	0.000025	0.000013	0.000001
2048	Выделение	0.000007	0.000003	0.000002
	Очистка	0.000020	0.000013	0.000001

Код программы

main.c

```
#include <stddef.h>

#include <unistd.h>

#include <string.h>

#include <stdlib.h>

#include <dlfcn.h>

#include <sys/mman.h>
```

```
#include <time.h>

typedef struct {
    void* (*allocator_create)(void* memory, size_t size);
    void (*allocator_destroy)(void* allocator);
    void* (*allocator_alloc)(void* allocator, size_t size);
    void (*allocator_free)(void* allocator, void* memory);
} AllocatorAPI;

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

void std_allocator_destroy(void* const allocator) {
    (void)allocator;
}

void* std_allocator_alloc(void* const allocator, const size_t size) {
    u_int32_t* memory = mmap(NULL, size, PROT_READ | PROT_WRITE,
                             MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (memory == MAP_FAILED) {
        return NULL;
    }
    *memory = (u_int32_t)(size + sizeof(u_int32_t));
    return memory + 1;
}
```

```
void std_allocator_free(void* const allocator, void* const memory) {
    if (memory == NULL) {
        return;
    }
    u_int32_t* mem = (u_int32_t*)memory - 1;
    munmap(mem, *mem);
}

void my_itoa(size_t value, char* buffer, int base) {
    char* digits = "0123456789";
    char temp[20];
    int i = 0;

    if (value == 0) {
        buffer[0] = '0';
        buffer[1] = '\0';
        return;
    }

    while (value > 0) {
        temp[i++] = digits[value % base];
        value /= base;
    }

    for (int j = 0; j < i; j++) {
        buffer[j] = temp[i - j - 1];
    }
    buffer[i] = '\0';
}
```

```
void my_ftoa(double value, char* buffer, int precision) {
    size_t int_part = (size_t)value;
    my_itoa(int_part, buffer, 10);

    while (*buffer) {
        buffer++;
    }

    *buffer++ = '.';

    double frac_part = value - (double)int_part;
    for (int i = 0; i < precision; i++) {
        frac_part *= 10;
        int digit = (int)frac_part;
        *buffer++ = '0' + digit;
        frac_part -= digit;
    }

    *buffer = '\0';
}

void write_message(const char* msg1, size_t number, const char* msg2, double time) {
    char buffer[256];
    char number_str[20];
    char time_str[20];

    my_itoa(number, number_str, 10);
    my_ftoa(time, time_str, 6);
```

```
    size_t len1 = strlen(msg1);
    size_t len2 = strlen(number_str);
    size_t len3 = strlen(msg2);
    size_t len4 = strlen(time_str);

    memcpy(buffer, msg1, len1);
    memcpy(buffer + len1, number_str, len2);
    memcpy(buffer + len1 + len2, msg2, len3);
    memcpy(buffer + len1 + len2 + len3, time_str, len4);

    buffer[len1 + len2 + len3 + len4] = '\n';
    write(STDOUT_FILENO, buffer, len1 + len2 + len3 + len4 + 1);
}

int main(int argc, char *argv[]) {
    void* handle = NULL;
    AllocatorAPI api;

    if (argc > 1) {
        handle = dlopen(argv[1], RTLD_LAZY);
        if (!handle) {
            char* msg = "cannot open library\n";
            write(STDOUT_FILENO, msg, strlen(msg));
            exit(EXIT_FAILURE);
        }

        api.allocator_create = dlsym(handle, "allocator_create");
        api.allocator_destroy = dlsym(handle, "allocator_destroy");
```

```
            api.allocator_alloc = dlsym(handle, "allocator_alloc");
            api.allocator_free = dlsym(handle, "allocator_free");

            if (!api.allocator_create || !api.allocator_destroy || !api.allocator_alloc ||
!api.allocator_free) {
                char* msg = "standart allocator error\n";
                write(STDOUT_FILENO, msg, strlen(msg));
                exit(EXIT_FAILURE);
            }
        } else {
            api.allocator_create = std_allocator_create;
            api.allocator_destroy = std_allocator_destroy;
            api.allocator_alloc = std_allocator_alloc;
            api.allocator_free = std_allocator_free;
        }

        size_t memory_size = 4096;
        size_t size_data = 2048;

        void *memory = mmap(NULL, memory_size, PROT_READ | PROT_WRITE,
MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
        if (memory == MAP_FAILED) {
            char* msg = "mmap error\n";
            write(STDOUT_FILENO, msg, strlen(msg));
            exit(EXIT_FAILURE);
        }

        void *allocator = api.allocator_create(memory, memory_size);
        if (!allocator) {
            char* msg = "mmap error\n";
```

```
    write(STDOUT_FILENO, msg, strlen(msg));
    munmap(memory, memory_size);
    exit(EXIT_FAILURE);
}

clock_t start, end;
double time_used;

start = clock();
void *ptr1 = api.allocator_alloc(allocator, size_data);
end = clock();
time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
write_message("time to allocate ", size_data, " bytes: ", time_used);

start = clock();
api.allocator_free(allocator, ptr1);
end = clock();
time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
write_message("time to free ", size_data, " bytes: ", time_used);

api.allocator_destroy(allocator);

munmap(memory, memory_size);

if (handle) {
    dlclose(handle);
}

return 0;
```

```
}
```

Allocator1.c

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

#define MAX_BLOCK_SIZE 4096
#define MIN_BLOCK_SIZE 16
#define SIZE_DATA 32

typedef struct FreeBlock {
    struct FreeBlock* next;
} FreeBlock;

typedef struct {
    FreeBlock* free_lists[SIZE_DATA];
    void* memory_start;
    size_t total_size;
    size_t cur_pos;
} MKKAllocator;

int get_free_list_index(size_t size) {
    int idx = 0;
    size_t cur_size = MIN_BLOCK_SIZE;
    while (cur_size < size && cur_size <= MAX_BLOCK_SIZE) {
        cur_size *= 2;
        idx++;
```

```
    }
    return idx;
}

MKKAllocator* allocator_create(void *const memory, size_t size) {
    if (size > MAX_BLOCK_SIZE) {
        char* msg = "enter smaller size\n";
        write(STDOUT_FILENO, msg, strlen(msg));
        return NULL;
    }

    MKKAllocator* allocator = mmap(NULL, sizeof(MKKAllocator), PROT_READ
| PROT_WRITE,
                                   MAP_PRIVATE | MAP_ANONYMOUS, -1,
0);

    if (allocator == MAP_FAILED) {
        char* msg = "mmap error, cannot create allocator\n";
        write(STDOUT_FILENO, msg, strlen(msg));
        return NULL;
    }

    allocator->memory_start = (MKKAllocator*)memory;
    allocator->total_size = size - sizeof(MKKAllocator);
    allocator->cur_pos = 0;

    memset(allocator->free_lists, 0, sizeof(allocator->free_lists));
    return allocator;
}
```

```
void* allocator_alloc(MKKAllocator* allocator, size_t size) {
    if (!allocator || size <= 0 || size > MAX_BLOCK_SIZE) {
        return NULL;
    }

    int idx = get_free_list_index(size);
    if (!allocator->free_lists[idx]) {
        size_t block_size = MIN_BLOCK_SIZE << idx;
        if (allocator->cur_pos + block_size > allocator->total_size) {
            char* msg = "not enough memory in allocator\n";
            write(STDOUT_FILENO, msg, strlen(msg));
            return NULL;
        }

        void *block = (char*)allocator->memory_start + allocator->cur_pos;
        allocator->cur_pos += block_size;

        return block;
    }

    FreeBlock* block = allocator->free_lists[idx];
    allocator->free_lists[idx] = block->next;

    return (void*)block;
}

void allocator_free(MKKAllocator* allocator, void* memory) {
    if (!allocator || !memory) {
        return;
```

```
        }

        size_t size = ((FreeBlock*)memory)->next ? MAX_BLOCK_SIZE :
MIN_BLOCK_SIZE;
        int idx = get_free_list_index(size);
        FreeBlock* block = (FreeBlock*)memory;
        block->next = allocator->free_lists[idx];
        allocator->free_lists[idx] = block;
    }

    void allocator_destroy(MKKAllocator* allocator) {
        if (!allocator) {
            return;
        }
        allocator->memory_start = NULL;
        allocator->total_size = 0;

        memset(allocator->free_lists, 0, sizeof(allocator->free_lists));
    }
```

Allocator2.c

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

#define SIZE_DATA 32

typedef struct BuddyBlock {
```

```
    size_t size;
    struct BuddyBlock *next;
    struct BuddyBlock *prev;
    int is_free;
} BuddyBlock;

typedef struct {
    void* memory;
    size_t total_size;
    BuddyBlock* free_lists[SIZE_DATA];
} BuddyAllocator;

void* allocator_create(void* const memory, const size_t size) {
    BuddyAllocator* allocator = (BuddyAllocator*)mmap(NULL,
sizeof(BuddyAllocator),
                                    PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
    if (allocator == MAP_FAILED) {
        char* msg = "mmap error, cannot create allocator\n";
        write(STDOUT_FILENO, msg, strlen(msg));
        return NULL;
    }

    allocator->memory = memory;
    allocator->total_size = size;

    for (int i = 0; i < SIZE_DATA; i++)
        allocator->free_lists[i] = NULL;

    BuddyBlock* initial_block = (BuddyBlock*)memory;
```

```
    initial_block->size = size;
    initial_block->next = NULL;
    initial_block->prev = NULL;
    initial_block->is_free = 1;

    int idx = 0;
    size_t block_size = 1;
    while (block_size < size) {
        block_size <<= 1;
        idx++;
    }

    initial_block->next = allocator->free_lists[idx];
    if (allocator->free_lists[idx]) {
        allocator->free_lists[idx]->prev = initial_block;
    }
    allocator->free_lists[idx] = initial_block;

    return allocator;
}

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

    if (munmap(allocator, sizeof(BuddyAllocator)) == -1) {
        char* msg = "munmap error\n";
        write(STDOUT_FILENO, msg, strlen(msg));
```

```
        exit(EXIT_FAILURE);
    }
}

void* allocator_alloc(void* const allocator, const size_t size) {
    BuddyAllocator* buddy_allocator = (BuddyAllocator*)allocator;
    size_t block_size = 1;
    int order = 0;

    while (block_size < size) {
        block_size <<= 1;
        order++;
    }

    BuddyBlock* block = NULL;
    for (int i = order; i < SIZE_DATA; i++) {
        if (buddy_allocator->free_lists[i]) {
            block = buddy_allocator->free_lists[i];
            order = i;
            break;
        }
    }

    if (!block) {
        return NULL;
    }

    if (block->next) {
        block->next->prev = block->prev;
```

```
    }
    if (block->prev) {
        block->prev->next = block->next;
    } else {
        buddy_allocator->free_lists[order] = block->next;
    }

    while (order > 0 && block_size > size) {
        order--;
        block_size >>= 1;

        BuddyBlock* buddy = (BuddyBlock*)((char*)block + block_size);
        buddy->size = block_size;
        buddy->next = buddy_allocator->free_lists[order];
        buddy->prev = NULL;
        buddy->is_free = 1;

        if (buddy_allocator->free_lists[order]) {
            buddy_allocator->free_lists[order]->prev = buddy;
        }
        buddy_allocator->free_lists[order] = buddy;
    }

    block->is_free = 0;
    return block;
}

void allocator_free(void* const allocator, void* const memory) {
    BuddyAllocator* buddy_allocator = (BuddyAllocator*)allocator;
```

```
    BuddyBlock* block = (BuddyBlock*)memory;
    block->is_free = 1;

    while (1) {
        size_t block_size = block->size;
        char* block_addr = (char*)block;
        ptrdiff_t offset = (block_addr - (char*)buddy_allocator->memory);
        char* buddy_addr = (char*)buddy_allocator->memory + (offset ^ block_size);

        if (buddy_addr < (char*)buddy_allocator->memory || buddy_addr >=
(char*)buddy_allocator->memory + buddy_allocator->total_size)
            break;

        BuddyBlock* buddy = (BuddyBlock*)buddy_addr;

        if (buddy->is_free && buddy->size == block_size) {
            if (buddy->next)
                buddy->next->prev = buddy->prev;
            if (buddy->prev)
                buddy->prev->next = buddy->next;
            else {
                int order = 0;
                size_t temp_size = block_size;
                while (temp_size >>= 1) {
                    order++;
                }
                buddy_allocator->free_lists[order] = buddy->next;
            }

            if (block < buddy) {
```

```
            block->size <<= 1;
        } else {
            buddy->size <<= 1;
            block = buddy;
        }
    } else {
        break;
    }
}

int order = 0;
size_t block_size = block->size;
while (block_size >>= 1) {
    order++;
}

block->next = buddy_allocator->free_lists[order];
block->prev = NULL;
if (buddy_allocator->free_lists[order]) {
    buddy_allocator->free_lists[order]->prev = block;
}
buddy_allocator->free_lists[order] = block;
}
```

Протокол работы программы

Тестирование:

1. Маленькая память
   $ ./main
   time to allocate 10 bytes: 0.000013
   time to free 10 bytes: 0.000032
   ./main ./allocator1.so
   time to allocate 10 bytes: 0.000003
   time to free 10 bytes: 0.000020

$ ./main ./allocator2.so
time to allocate 10 bytes: 0.000002
time to free 10 bytes: 0.000001

2. Не степень двойки
   $ ./main
   time to allocate 1585 bytes: 0.000009
   time to free 1585 bytes: 0.000025
   $ ./main ./allocator1.so
   time to allocate 1585 bytes: 0.000003
   time to free 1585 bytes: 0.00013
   $ ./main ./allocator2.so
   time to allocate 1585 bytes: 0.000002
   time to free 1585 bytes: 0.000001

3. Степень двойки
   $ ./main
   time to allocate 2048 bytes: 0.000007
   time to free 2048 bytes: 0.000020
   $ ./main ./allocator1.so
   time to allocate 2048 bytes: 0.000003
   time to free 2048 bytes: 0.000013
   $ ./main ./allocator2.so
   time to allocate 2048 bytes: 0.000002
   time to free 2048 bytes: 0.000001

Strace:

$ strace -f ./main ./allocator1.so

execve("./main", ["./main", "./allocator1.so"], 0x7ffeeaeb75f0 /* 68 vars */) = 0

brk(NULL) = 0x5e50f05ef000

arch_prctl(0x3001 /* ARCH_??? */, 0x7ffe57aff9a0) = -1 EINVAL (Invalid argument)

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7cdebc3ef000

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=58791, ...}, AT_EMPTY_PATH) = 0

mmap(NULL, 58791, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7cdebc3e0000

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) = 0x7cdebc000000

mprotect(0x7cdebc028000, 2023424, PROT_NONE) = 0

mmap(0x7cdebc028000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7cdebc028000

mmap(0x7cdebc1bd000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7cdebc1bd000

mmap(0x7cdebc216000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x7cdebc216000

mmap(0x7cdebc21c000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7cdebc21c000

close(3) = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7cdebc3dd000

arch_prctl(ARCH_SET_FS, 0x7cdebc3dd740) = 0

set_tid_address(0x7cdebc3dda10) = 4689

set_robust_list(0x7cdebc3dda20, 24) = 0

rseq(0x7cdebc3de0e0, 0x20, 0, 0x53053053) = 0

mprotect(0x7cdebc216000, 16384, PROT_READ) = 0

mprotect(0x5e50ef5c9000, 4096, PROT_READ) = 0

mprotect(0x7cdebc429000, 8192, PROT_READ) = 0

prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0

munmap(0x7cdebc3e0000, 58791) = 0

getrandom("\x82\xae\xbd\x82\x00\x81\x85\x96", 8, GRND_NONBLOCK) = 8

brk(NULL) = 0x5e50f05ef000

brk(0x5e50f0610000) = 0x5e50f0610000

openat(AT_FDCWD, "./allocator1.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"..., 832) = 832

newfstatat(3, "", {st_mode=S_IFREG|0775, st_size=15912, ...}, AT_EMPTY_PATH) = 0

getcwd("/home/aaxelf/Desktop/os/lab4", 128) = 29

mmap(NULL, 16464, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7cdebc3ea000

mmap(0x7cdebc3eb000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7cdebc3eb000

mmap(0x7cdebc3ec000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7cdebc3ec000

mmap(0x7cdebc3ed000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7cdebc3ed000

close(3) = 0

mprotect(0x7cdebc3ed000, 4096, PROT_READ) = 0

mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7cdebc428000

mmap(NULL, 280, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7cdebc3e9000

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=5424540}) = 0

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=5431015}) = 0

write(1, "time to allocate 10 bytes: 0.000"..., 36time to allocate 10 bytes: 0.000007) = 36

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=5450708}) = 0

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=5467991}) = 0

write(1, "time to free 10 bytes: 0.000017\n", 32time to free 10 bytes: 0.000017) = 32

munmap(0x7cdebc428000, 4096) = 0

munmap(0x7cdebc3ea000, 16464) = 0

exit_group(0) = ?

+++ exited with 0 +++

Вывод

В рамках лабораторной работы была написана программа, демонстрирующая работу динамических аллокаторов. В ходе сравнения скорости выделения и освобождения памяти было выявлено, что системный аллокатор работает медленнее всех в обоих действиях, алгоритм МКК медленно освобождает память, а алгоритм двойников хорошо показал себя во всех случаях.