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-211Б-23

Студент: Тремель Д.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 08.01.24

Москва, 2024

Постановка задачи

Вариант 5.

Исследовать два аллокатора памяти: необходимо реализовать два алгоритма аллокации памяти. Алгоритм Мак-Кьюзика-Кэрелса и алгоритм двойников.

Общий метод и алгоритм решения

Использованные системные вызовы:

int munmap(void addr, size_t length) - Удаляет отображения, созданные с помощью mmap.

int dlclose(void handle) - Закрывает динамическую библиотеку, открытую с помощью dlopen, и освобождает ресурсы, связанные с этим дескриптором.

void exit(int status) - Завершает выполнение программы и возвращает статус выхода в операционную систему.

char dlerror(void) - Возвращает строку, описывающую последнюю ошибку, возникшую при вызове функций dlopen, dlsym, dlclose.

void dlopen(const char filename, int flag) - Открывает динамическую библиотеку и возвращает дескриптор для последующего использования.

void mmap(void addr, size_t length, int prot, int flags, int fd, off_t offset) - Создает новое отображение памяти или изменяет существующее.

○ **Реализация алгоритма Мак-Кьюзика-Кэрелса**: Память управляется с помощью списка свободных блоков. При выделении памяти алгоритм тщательно ищет наиболее подходящий свободный блок, чтобы минимизировать количество неиспользуемой памяти. Этот подход основан на динамическом управлении списками свободных блоков, которые позволяют оптимально распределять память для различных запросов, сохраняя её фрагментацию на минимальном уровне.

○ **Реализация алгоритма двойников**: Память выделяется блоками, размеры которых являются степенями двойки. Это делает процесс управления выделением более структурированным, поскольку размеры блоков легко масштабируются и соответствуют большинству стандартных запросов. Однако такой подход может приводить к внутренней фрагментации памяти, поскольку выделенный блок может оказаться больше необходимого. Алгоритм особенно полезен в системах, где важна скорость выделения памяти, а оптимизация использования менее критична.

Сравнение

Фактор использования памяти

Алгоритм Мак-Кьюзика-Кареса: Управление памятью осуществляется через списки свободных блоков. Для каждого размера выделенного блока поддерживается отдельный список, из которого выбирается наиболее подходящий блок при запросе на выделение памяти. Высокий уровень эффективности при использовании памяти достигается за счёт минимизации внутренней фрагментации. Блоки подбираются таким образом, чтобы размер максимально соответствовал запросу, что ведёт к оптимальному использованию доступной памяти.

Алгоритм двойников:Память выделяется блоками, размерами которых являются степени двойки. Это подразумевает использование системы buddy’s (двойников) для управления памятью и объединения блоков при их освобождении. Склонен к значительной внутренней фрагментации, поскольку выделенные блоки часто превышают необходимый размер. Однако уровень внешней фрагментации обычно ниже, так как память управляется более структурированно.

Скорость выделения блоков

Алгоритм Мак-Кьюзика-Кареса: Может быть замедлена из-за необходимости поиска свободных блоков для нахождения наиболее подходящего.

Алгоритм двойников: Обычно выше, так как необходимо лишь выбрать ближайший размер блока, соответствующий степени двойки.

Скорость освобождения блоков

Алгоритм Мак-Кьюзика-Кареса: Может потребовать больше времени из-за необходимости обновления списков и потенциального объединения смежных свободных блоков.

Алгоритм двойников: Освобождение происходит быстро, поскольку освобождённый блок просто возвращается в соответствующий список двойников, а объединение выполняется по простой логике.

Простота использования аллокатора

Алгоритм Мак-Кьюзика-Кареса: Преимущественно прост в реализации, так как основная логика заключается в поддержке списков и управлении ими.

Алгоритм двойников: Сложнее в реализации и использовании из-за необходимости поддержки системы двойников и более сложного управления процессом объединения блоков.

Код программы

allocator_1.c

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

#define PAGE_SIZE 4096
#define MIN_BLOCK_SIZE 16

typedef struct {
    unsigned char *memory_region;
    size_t total_memory_size;
    size_t total_blocks;
    unsigned char *allocation_map;
    size_t map_size;
```

```
} MemoryAllocator;

static size_t calculate_allocation_map_size(size_t block_count) {
    return (block_count + 7) / 8;
}

MemoryAllocator *allocator_create(void *const memory, size_t requested_size) {
    if (requested_size == 0) {
        fprintf(stderr, "Error: Requested size is zero\n");
        return NULL;
    }

    requested_size = (requested_size + PAGE_SIZE - 1) & ~(PAGE_SIZE - 1);

    size_t block_count = requested_size / MIN_BLOCK_SIZE;
    size_t allocation_map_size = calculate_allocation_map_size(block_count);

    MemoryAllocator *allocator = mmap(NULL, sizeof(MemoryAllocator), PROT_READ
 | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (allocator == MAP_FAILED) {
        perror("Error mmap for allocator structure");
        return NULL;
    }

    allocator->memory_region = mmap(NULL, requested_size, PROT_READ |
 PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (allocator->memory_region == MAP_FAILED) {
        perror("Error mmap for memory region");
        munmap(allocator, sizeof(MemoryAllocator));
        return NULL;
    }

    allocator->allocation_map = mmap(NULL, allocation_map_size, PROT_READ |
 PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (allocator->allocation_map == MAP_FAILED) {
        perror("Error mmap for allocation map");
        munmap(allocator->memory_region, requested_size);
        munmap(allocator, sizeof(MemoryAllocator));
        return NULL;
    }

    allocator->total_memory_size = requested_size;
    allocator->total_blocks = block_count;
    allocator->map_size = allocation_map_size;
    memset(allocator->allocation_map, 0xFF, allocation_map_size);

    return allocator;
}
```

```
void *allocator_alloc(MemoryAllocator *allocator, size_t size) {
    if (!allocator || size == 0) {
        fprintf(stderr, "Error: Invalid allocation request\n");
        return NULL;
    }

    if (size > allocator->total_memory_size) {
        fprintf(stderr, "Error: Requested size exceeds total memory\n");
        return NULL;
    }

    size_t required_blocks = (size + MIN_BLOCK_SIZE - 1) / MIN_BLOCK_SIZE;

    for (size_t i = 0; i < allocator->total_blocks - required_blocks + 1;) {
        size_t j;
        for (j = 0; j < required_blocks; j++) {
            size_t byte_index = (i + j) / 8;
            size_t bit_offset = (i + j) % 8;
            if (!(allocator->allocation_map[byte_index] & (1 << bit_offset))) {
                break;
            }
        }

        if (j == required_blocks) {
            for (j = 0; j < required_blocks; j++) {
                size_t byte_index = (i + j) / 8;
                size_t bit_offset = (i + j) % 8;
                allocator->allocation_map[byte_index] &= ~(1 << bit_offset);
            }
            return allocator->memory_region + i * MIN_BLOCK_SIZE;
        }
        i += j + 1;
    }

    fprintf(stderr, "Error: Not enough free memory\n");
    return NULL;
}

void allocator_free(MemoryAllocator *allocator, void *ptr, size_t size) {
    if (!allocator || !ptr || size == 0) {
        fprintf(stderr, "Error: Invalid free request\n");
        return;
    }

    unsigned char *aligned_ptr = (unsigned char *)ptr;
    if (aligned_ptr < allocator->memory_region ||
        aligned_ptr >= allocator->memory_region + allocator->total_memory_size) {
        fprintf(stderr, "Error: Pointer out of allocator bounds\n");
        return;
```

```
    }

    if ((aligned_ptr - allocator->memory_region) % MIN_BLOCK_SIZE != 0) {
        fprintf(stderr, "Error: Pointer is not aligned to block boundary\n");
        return;
    }

    size_t blocks_to_free = (size + MIN_BLOCK_SIZE - 1) / MIN_BLOCK_SIZE;
    size_t starting_block_index = (aligned_ptr - allocator->memory_region) /
 MIN_BLOCK_SIZE;

    if (starting_block_index >= allocator->total_blocks ||
        starting_block_index + blocks_to_free > allocator->total_blocks) {
        fprintf(stderr, "Error: Blocks to free exceed memory bounds\n");
        return;
    }

    for (size_t i = starting_block_index; i < starting_block_index +
 blocks_to_free; i++) {
        allocator->allocation_map[i / 8] |= (1 << (i % 8));
    }
}

void allocator_destroy(MemoryAllocator *allocator) {
    if (!allocator) {
        fprintf(stderr, "Error: Attempt to destroy a non-existent
allocator\n");
        return;
    }

    if (allocator->allocation_map) {
        munmap(allocator->allocation_map, allocator->map_size);
        allocator->allocation_map = NULL;
    }

    if (allocator->memory_region) {
        munmap(allocator->memory_region, allocator->total_memory_size);
        allocator->memory_region = NULL;
    }

    munmap(allocator, sizeof(MemoryAllocator));
}

}
```

allocator_2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/mman.h>
#include <string.h>
#include <stdint.h>

typedef struct BuddyAllocator {
    void *memory_region;
    size_t total_memory_size;
    size_t min_block_size;
    size_t max_block_size;
    int max_levels;
    void **free_lists;
} BuddyAllocator;

BuddyAllocator *allocator_create(void *const memory, const size_t size) {
    size_t total_size = pow(2, ceil(log2(size)));
    size_t min_block_size = 64;
    size_t max_block_size = total_size;
    int max_levels = log2(max_block_size / min_block_size) + 1;

    BuddyAllocator *allocator = mmap(NULL, sizeof(BuddyAllocator),
                                     PROT_READ | PROT_WRITE,
                                     MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (allocator == MAP_FAILED) {
        perror("mmap");
        return NULL;
    }

    allocator->memory_region = mmap(NULL, total_size, PROT_READ | PROT_WRITE,
                                    MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (allocator->memory_region == MAP_FAILED) {
        perror("mmap");
        munmap(allocator, sizeof(BuddyAllocator));
        return NULL;
    }

    allocator->total_memory_size = total_size;
    allocator->min_block_size = min_block_size;
    allocator->max_block_size = max_block_size;
    allocator->max_levels = max_levels;

    allocator->free_lists = mmap(NULL, max_levels * sizeof(void *),
                                 PROT_READ | PROT_WRITE,
                                 MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (allocator->free_lists == MAP_FAILED) {
        perror("mmap");
        munmap(allocator->memory_region, total_size);
```

```
        munmap(allocator, sizeof(BuddyAllocator));
        return NULL;
    }

    memset(allocator->free_lists, 0, max_levels * sizeof(void *));
    allocator->free_lists[max_levels - 1] = allocator->memory_region;

    return allocator;
}

void *allocator_alloc(BuddyAllocator *const allocator, const size_t size) {
    size_t adjusted_size = size < allocator->min_block_size ?
 allocator->min_block_size : size;
    int level = log2(allocator->max_block_size / adjusted_size);

    for (int i = level; i < allocator->max_levels; i++) {
        if (allocator->free_lists[i] != NULL) {
            void *block = allocator->free_lists[i];
            allocator->free_lists[i] = *(void **)block;

            while (i > level) {
                i--;
                void *buddy = (void *)((char *)block + (allocator->min_block_size
<< i));
                *(void **)buddy = allocator->free_lists[i];
                allocator->free_lists[i] = buddy;
            }

            return block;
        }
    }

    return NULL;
}

void allocator_destroy(BuddyAllocator *const allocator) {
    munmap(allocator->free_lists, allocator->max_levels * sizeof(void *));
    munmap(allocator->memory_region, allocator->total_memory_size);
    munmap(allocator, sizeof(BuddyAllocator));
}

void allocator_free(BuddyAllocator *const allocator, void *const memory) {
    size_t size = allocator->min_block_size;
    int level = log2(allocator->max_block_size / size);
    void *ptr = memory;

    while (level < allocator->max_levels - 1) {
        void *buddy = (void *)((uintptr_t)ptr ^ (allocator->min_block_size <<
level));
        void **prev = &allocator->free_lists[level];
        while (*prev != NULL && *prev != buddy) {
```

```
            prev = (void **)*prev;
        }

        if (*prev == buddy) {
            *prev = *(void **)buddy;
            ptr = (ptr < buddy) ? ptr : buddy;
            level++;
        } else {
            break;
        }
    }

    *(void **)ptr = allocator->free_lists[level];
    allocator->free_lists[level] = ptr;
}
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <sys/mman.h>
#include <time.h>

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

void* fallback_allocator_create(void *const memory, const size_t size) {
    printf("%li\n", size);
    return memory;
}

void fallback_allocator_destroy(void *const allocator) {
    if (allocator)
        printf("\n");
}

void* fallback_allocator_alloc(void *const allocator, const size_t size) {
    printf("%li\n", size);
    if (allocator)
        printf("\n");
    return mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
}
```

```
void fallback_allocator_free(void *const allocator, void *const memory, size_t
size) {
    munmap(memory, 4096);
    if (allocator)
        printf("\n");
}

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



    if (argc > 1) {
        handle = dlopen(argv[1], RTLD_LAZY);
        if (!handle) {
            fprintf(stderr, "%s\n", dlerror());
            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) {
            fprintf(stderr, "%s\n", dlerror());
            exit(EXIT_FAILURE);
        }
    } else {
        api.allocator_create = fallback_allocator_create;
        api.allocator_destroy = fallback_allocator_destroy;
        api.allocator_alloc = fallback_allocator_alloc;
        api.allocator_free = fallback_allocator_free;
    }

    size_t memory_size = 4096;
    size_t size_data = 128;

    void *memory = mmap(NULL, memory_size, PROT_READ | PROT_WRITE, MAP_PRIVATE
| MAP_ANONYMOUS, -1, 0);
    if (memory == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }

    void *allocator = api.allocator_create(memory, memory_size);

    clock_t start, end;
    double cpu_time_used;
```

```
    start = clock();
    void *ptr1 = api.allocator_alloc(allocator, size_data);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time to allocate %li bytes: %f seconds\n", size_data,
 cpu_time_used);

    start = clock();
    api.allocator_free(allocator, ptr1, size_data);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Time to free %li bytes: %f seconds\n", size_data, cpu_time_used);

    api.allocator_destroy(allocator);

    munmap(memory, memory_size);

    if (handle) {
        dlclose(handle);
    }

    return 0;
}
```

Протокол работы программы

```
u@DESKTOP-3U3OER0:/mnt/c/Users/u/CLionProjects/OS/lab_4$ ./main
./allocator_1.so

Time to allocate 128 bytes: 0.000002 seconds

Time to free 128 bytes: 0.000002 seconds


u@DESKTOP-3U3OER0:/mnt/c/Users/u/CLionProjects/OS/lab_4$ ./main
./allocator_2.so

Time to allocate 128 bytes: 0.000004 seconds

Time to free 128 bytes: 0.000001 seconds
```

strace

```
u@DESKTOP-3U3OER0:/mnt/c/Users/u/CLionProjects/OS/lab_4$ strace ./main
./allocator_

1.so

execve("./main", ["./main", "./allocator_1.so"], 0x7ffed4378d88 /* 26
vars */) = 0

brk(NULL)                               = 0x557e87428000

arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc93a71950) = -1 EINVAL (Invalid
argument)

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=61562, ...}) = 0

mmap(NULL, 61562, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fe9f4a13000

close(3)                                = 0

openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libdl.so.2", 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
\22\0\0\0\0\0\0"..., 832) =

 832

fstat(3, {st_mode=S_IFREG|0644, st_size=18848, ...}) = 0

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7fe9f4

a11000
```

```
mmap(NULL, 20752, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7fe9f4a0b000

mmap(0x7fe9f4a0c000, 8192, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE

, 3, 0x1000) = 0x7fe9f4a0c000

mmap(0x7fe9f4a0e000, 4096, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x300

0) = 0x7fe9f4a0e000

mmap(0x7fe9f4a0f000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRIT

E, 3, 0x3000) = 0x7fe9f4a0f000

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\300A\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\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0", 32,

848) = 32

pread64(3,
"\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\7\2C\n\357_\243\335\2449\206V>\237\374\3

04"..., 68, 880) = 68

fstat(3, {st_mode=S_IFREG|0755, st_size=2029592, ...}) = 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

pread64(3,
"\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0", 32,

848) = 32

pread64(3,
"\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\7\2C\n\357_\243\335\2449\206V>\237\374\3

04"..., 68, 880) = 68
```

```
mmap(NULL, 2037344, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7fe9f4819000

mmap(0x7fe9f483b000, 1540096, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWR

ITE, 3, 0x22000) = 0x7fe9f483b000

mmap(0x7fe9f49b3000, 319488, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1

9a000) = 0x7fe9f49b3000

mmap(0x7fe9f4a01000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRI

TE, 3, 0x1e7000) = 0x7fe9f4a01000

mmap(0x7fe9f4a07000, 13920, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMO

US, -1, 0) = 0x7fe9f4a07000

close(3)                                = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7fe9f

4816000

arch_prctl(ARCH_SET_FS, 0x7fe9f4816740) = 0

mprotect(0x7fe9f4a01000, 16384, PROT_READ) = 0

mprotect(0x7fe9f4a0f000, 4096, PROT_READ) = 0

mprotect(0x557e71826000, 4096, PROT_READ) = 0

mprotect(0x7fe9f4a50000, 4096, PROT_READ) = 0

munmap(0x7fe9f4a13000, 61562)           = 0

brk(NULL)                               = 0x557e87428000

brk(0x557e87449000)                     = 0x557e87449000

openat(AT_FDCWD, "./allocator_1.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\340\20\0\0\0\0\0\0"...,
832

) = 832

fstat(3, {st_mode=S_IFREG|0777, st_size=16640, ...}) = 0

getcwd("/mnt/c/Users/u/CLionProjects/OS/lab_4", 128) = 38

mmap(NULL, 16464, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7fe9f4a1e000
```

```
mmap(0x7fe9f4a1f000, 4096, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE

, 3, 0x1000) = 0x7fe9f4a1f000

mmap(0x7fe9f4a20000, 4096, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x200

0) = 0x7fe9f4a20000

mmap(0x7fe9f4a21000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRIT

E, 3, 0x2000) = 0x7fe9f4a21000

close(3)                                = 0

mprotect(0x7fe9f4a21000, 4096, PROT_READ) = 0

mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7fe9f4

a4f000

mmap(NULL, 40, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7fe9f4a1

d000

mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7fe9f4

a1c000

mmap(NULL, 32, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7fe9f4a1

b000

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=7067300}) = 0

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=7098500}) = 0

fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}) = 0

write(1, "Time to allocate 128 bytes: 0.00"..., 45Time to allocate 128
bytes: 0.000

031 seconds

) = 45

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=7250300}) = 0

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=7300100}) = 0

write(1, "Time to free 128 bytes: 0.000050"..., 41Time to free 128
bytes: 0.000050

seconds
```

```
) = 41
munmap(0x7fe9f4a1b000, 32)              = 0
munmap(0x7fe9f4a1c000, 4096)            = 0
munmap(0x7fe9f4a1d000, 40)              = 0
munmap(0x7fe9f4a4f000, 4096)            = 0
munmap(0x7fe9f4a1e000, 16464)           = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Вывод

В процессе выполнения данной лабораторной работы я реализовал программу, которая выделяет память через передаваемый аллокатор. Сравнил данные алгоритмы по разным критериям.