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-214Б-23

Студент: Миронов Д. А.

Преподаватель: Бахарев В.Д. (ФИИТ)

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

Дата: 13.12.24

Москва, 2024

**Постановка задачи**

**Вариант 5.**

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). Аллокаторы – метод Мак-Кьюзика-Кэрелса и алгоритм двойников.

**Общий метод и алгоритм решения**

Использованные системные вызовы:

* int write(int fd, void\* buf, size\_t count); – записывает count байт из buf в fd.
* void \*mmap(void addr, size\_t length, int prot, int flags, int fd, off\_t offset); – выполняет отображение файла или устройства на память.
* int munmap(void addr, size\_t length); – удаляет отображение файла или устройства на память.

Программа lab4 в функции load\_allocator загружает динамическую библиотеку по указанному пути используя dlopen. Если библиотеку загрузить не удалось, выводится сообщение об ошибке и указателям на функции присвоены указатели на функции по умолканию, оборачивающие mmap и munmap. Если загрузить библиотеку удалось, то программа пытается найти в библиотеке соответствующие функции и присвоить их указателям из структуры Allocator. Если функции нет, то соответствующему указателю на функцию присвоен указатель на функцию оборачивающую mmap или munmap. В функции load\_allocator вызывается с параметром argv[1], указывающем путь до библиотеки. Далее демонстрируется работа загруженных функции на примере работы с массивами.

Библиотека MucKus реализует аллокатор на основании метода Мак-Кьюзика-Кэрелса. В этом аллокаторе мы будем хранить массив классов, каждый из которых будет представлять собой список свободных блоков определённого размера. Инициализация аллокатора (allocator\_create): аллокатор принимает на вход память (memory) и её размер (size). Аллокатор инициализирует массив размеров классов и массив самих классов. Запрос памяти (allocator\_alloc): аллокатор находит класс с минимальным размером, удовлетворяющим запросу и ищет свободный блок в массиве классов, возвращая его. Если такого блока нет, аллокатор выделяет блок из глобальной памяти. Освобождение памяти (allocator\_free): проверяет, принадлежит ли переданный указатель данному аллокатору, затем находит размер класса, к которому принадлежит данный блок и добавляет этот блок в нужный список массива классов. Очистка аллокатора (allocator\_destroy): очищает список свободных блоков.

Библиотека buddy реализует аллокатор на основании метода двойников. В этом аллокаторе память выделяется блоками, размером 2^n. Инициализация аллокатора (allocator\_create): аллокатор принимает на вход память (memory) и её размер (size), проверяет, что size — степень двойки, выделяет часть памяти для структуры аллокатора (Allocator) и корневого узла (Block), который представляет всю память целиком, создаёт корневой узел, помеченный как свободный и соответствующий общему размеру памяти. Запрос памяти (allocator\_alloc): запрашиваемый размер выравнивается до ближайшей степени двойки, если это необходимо, аллокатор начинает с корня и рекурсивно ищет свободный блок, если блок не подходит по размеру или уже занят, возвращается NULL, если размер блока равен запрашиваемому, блок помечается как занятый, если размер блока больше запрашиваемого, блок делится на два («левый» и «правый»), после чего запрос перенаправляется сначала к левому потомку, а затем к правому (при необходимости). Разделение узлов (split\_node): когда узел делится, создаются два новых дочерних узла, размеры которых равны половине исходного блока, эти узлы размещаются в заранее выделенной области памяти, смещённой относительно начала. Освобождение памяти (allocator\_free): узел, переданный в my\_free, помечается как свободный, если дочерние узлы (left и right) тоже свободны, они уничтожаются (объединяются), а исходный блок снова становится свободным и представляет объединённый блок, этот процесс позволяет повторное объединение (слияние) памяти. Очистка аллокатора (allocator\_destroy): рекурсивно освобождает все узлы, начиная с корня, помечая их как свободные, использует munmap для освобождения всей выделенной аллокатору памяти.

**Код программы**

**MacKus.c**

#include <stdlib.h>

#include <string.h>

#include <stdint.h>

#define MAX\_CLASSES 10

**typedef** **struct** Block {

**struct** Block\* next;

} Block;

**typedef** **struct** Allocator {

Block\* free\_list[MAX\_CLASSES];

size\_t class\_sizes[MAX\_CLASSES];

**void**\* memory\_start;

size\_t memory\_size;

} Allocator;

**static** size\_t find\_class(size\_t size, size\_t\* class\_sizes, size\_t num\_classes) {

**for** (size\_t i = 0; i < num\_classes; i++) {

**if** (size <= class\_sizes[i]) {

**return** i;

}

}

**return** num\_classes; // Error case

}

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

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

**if** (size < **sizeof**(Allocator)) **return** NULL;

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

allocator->memory\_size = size - **sizeof**(Allocator);

// Инициализация классов размеров

size\_t block\_size = 16;

**for** (size\_t i = 0; i < MAX\_CLASSES; i++) {

allocator->class\_sizes[i] = block\_size;

allocator->free\_list[i] = NULL;

block\_size \*= 2;

}

**return** allocator;

}

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

// Очистка списков свободных блоков

**for** (size\_t i = 0; i < MAX\_CLASSES; i++) {

allocator->free\_list[i] = NULL;

}

}

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

size\_t class\_index = find\_class(size, allocator->class\_sizes, MAX\_CLASSES);

**if** (class\_index >= MAX\_CLASSES) **return** NULL;

Block\* block = allocator->free\_list[class\_index];

**if** (block) {

allocator->free\_list[class\_index] = block->next;

**return** (**void**\*)block;

}

// Если свободных блоков нет, выделяем из общей памяти

size\_t block\_size = allocator->class\_sizes[class\_index];

**if** (allocator->memory\_size < block\_size) **return** NULL;

**void**\* memory = allocator->memory\_start;

allocator->memory\_start = (**char**\*)allocator->memory\_start + block\_size;

allocator->memory\_size -= block\_size;

**return** memory;

}

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

**if** (!memory) **return**;

uintptr\_t addr = (uintptr\_t)memory - (uintptr\_t)allocator;

**if** (addr >= allocator->memory\_size) **return**;

size\_t class\_index = 0;

size\_t block\_size = 0;

**for** (; class\_index < MAX\_CLASSES; class\_index++) {

block\_size = allocator->class\_sizes[class\_index];

**if** ((uintptr\_t)memory <= block\_size) {

**break**;

}

}

**if** (class\_index >= MAX\_CLASSES) **return**;

Block\* block = (Block\*)memory;

block->next = allocator->free\_list[class\_index];

allocator->free\_list[class\_index] = block;

}

**buddy.c**

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/mman.h>

**typedef** **struct** Block{

**int** size;

**int** free;

**struct** Block \*left,\*right;

} Block;

**typedef** **struct** Allocator {

Block \*root;

**void** \*memory;

**int** total\_size, offset;

} Allocator;

**int** isPowerOf2(**unsigned** **int** n) {

**return** (n > 0) && ((n & (n - 1)) == 0);

}

Block \*create\_node(Allocator \*allocator, **int** size) {

**if** (allocator->offset + **sizeof**(Block) > allocator->total\_size) {

**return** NULL;

}

Block \*node = (Block \*)((**char** \*)allocator->memory + allocator->offset);

allocator->offset += **sizeof**(Block);

node->size = size;

node->free = 1;

node->left = node->right = NULL;

**return** node;

}

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

**if** (!isPowerOf2(size)) {

**const** **char** msg[] = "ERROR: allocator size is not power of 2\n";

write(STDERR\_FILENO, msg, **sizeof**(msg) - 1);

**return** NULL;

}

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

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

allocator->total\_size = size - **sizeof**(Allocator);

allocator->offset = 0;

allocator->root = create\_node(allocator, size);

**if** (!allocator->root) {

**return** NULL;

}

**return** allocator;

}

**void** split\_node(Allocator \*allocator, Block \*node) {

**int** newSize = node->size / 2;

node->left = create\_node(allocator, newSize);

node->right = create\_node(allocator, newSize);

}

Block \*allocate\_block(Allocator \*allocator, Block \*node, **int** size) {

**if** (node == NULL || node->size < size || !node->free) {

**return** NULL;

}

**if** (node->size == size) {

node->free = 0;

**return** (**void** \*)node;

}

**if** (node->left == NULL) {

split\_node(allocator, node);

}

**void** \*allocated = allocate\_block(allocator, node->left, size);

**if** (allocated == NULL) {

allocated = allocate\_block(allocator, node->right, size);

}

node->free = (node->left && node->left->free) || (node->right && node->right->free);

**return** allocated;

}

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

**if** (allocator == NULL || size <= 0) {

**return** NULL;

}

**while** (!isPowerOf2(size)) {

size++;

}

**return** allocate\_block(allocator, allocator->root, size);

}

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

**if** (allocator == NULL || ptr == NULL) {

**return**;

}

Block \*node = (Block \*)ptr;

**if** (node == NULL) {

**return**;

}

node->free = 1;

**if** (node->left != NULL && node->left->free && node->right->free) {

allocator\_free(allocator, node->left);

allocator\_free(allocator, node->right);

node->left = node->right = NULL;

}

}

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

**if** (!allocator){

**return**;

}

allocator\_free(allocator, allocator->root);

**if** (munmap((**void** \*)allocator, allocator->total\_size + **sizeof**(Allocator)) == 1) {

exit(EXIT\_FAILURE);

}

}

**lab4.c**

#include <stdio.h>

#include <stdlib.h>

#include <dlfcn.h>

#include <sys/mman.h>

#include <stdint.h>

#include <stddef.h>

#include <time.h> // Для измерения времени

**typedef** **struct** Allocator {

size\_t size; // Размер памяти, выделенной для аллокатора

**void** \*memory; // Указатель на выделенную память

} Allocator;

// Определение типов для функций из динамических библиотек

**typedef** Allocator\* (\*allocator\_create\_f)(**void** \***const** memory, **const** size\_t size);

**typedef** **void** (\*allocator\_destroy\_f)(Allocator \***const** allocator);

**typedef** **void**\* (\*allocator\_alloc\_f)(Allocator \***const** allocator, **const** size\_t size);

**typedef** **void** (\*allocator\_free\_f)(Allocator \***const** allocator, **void** \***const** memory);

// Указатели на функции, которые будут загружаться динамически

**static** allocator\_create\_f allocator\_create = NULL;

**static** allocator\_destroy\_f allocator\_destroy = NULL;

**static** allocator\_alloc\_f allocator\_alloc = NULL;

**static** allocator\_free\_f allocator\_free = NULL;

// Функции для fallback-аллокатора

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

Allocator \*allocator = (Allocator\*) mmap(NULL, **sizeof**(Allocator), PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANON, -1, 0);

**if** (allocator != MAP\_FAILED) {

allocator->size = size;

allocator->memory = memory;

}

**return** allocator;

}

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

munmap(allocator->memory, allocator->size);

munmap(allocator, **sizeof**(Allocator));

}

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

**return** mmap(NULL, size, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANON, -1, 0);

}

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

munmap(memory, **sizeof**(memory));

}

// Функция для загрузки библиотеки

**void** load\_allocator\_library(**const** **char** \*path) {

**void** \*handle = dlopen(path, RTLD\_LAZY);

**if** (!handle) {

fprintf(stderr, "Error loading library: %s\n", dlerror());

**return**;

}

allocator\_create = (allocator\_create\_f) dlsym(handle, "allocator\_create");

allocator\_destroy = (allocator\_destroy\_f) dlsym(handle, "allocator\_destroy");

allocator\_alloc = (allocator\_alloc\_f) dlsym(handle, "allocator\_alloc");

allocator\_free = (allocator\_free\_f) dlsym(handle, "allocator\_free");

**if** (!allocator\_create || !allocator\_destroy || !allocator\_alloc || !allocator\_free) {

fprintf(stderr, "Error loading functions from library\n");

dlclose(handle);

}

}

// Функция для измерения времени работы аллокатора

**double** measure\_time\_allocation(Allocator \*allocator, size\_t alloc\_size, **int** num\_allocs) {

clock\_t start = clock();

**for** (**int** i = 0; i < num\_allocs; ++i) {

**void** \*block = allocator\_alloc(allocator, alloc\_size);

**if** (!block) {

fprintf(stderr, "Allocation failed at iteration %d\n", i);

**break**;

}

allocator\_free(allocator, block);

}

clock\_t end = clock();

**return** (**double**)(end - start) / CLOCKS\_PER\_SEC;

}

// Функция для измерения времени освобождения памяти

**double** measure\_time\_free(Allocator \*allocator, size\_t alloc\_size, **int** num\_allocs) {

**void** \*\*blocks = malloc(num\_allocs \* **sizeof**(**void**\*));

**for** (**int** i = 0; i < num\_allocs; ++i) {

blocks[i] = allocator\_alloc(allocator, alloc\_size);

}

clock\_t start = clock();

**for** (**int** i = 0; i < num\_allocs; ++i) {

allocator\_free(allocator, blocks[i]);

}

clock\_t end = clock();

free(blocks);

**return** (**double**)(end - start) / CLOCKS\_PER\_SEC;

}

**int** main(**int** argc, **char** \*\*argv) {

**if** (argc < 2) {

printf("No library path provided, using fallback allocator\n");

allocator\_create = (allocator\_create\_f) fallback\_allocator\_create;

allocator\_destroy = (allocator\_destroy\_f) fallback\_allocator\_destroy;

allocator\_alloc = (allocator\_alloc\_f) fallback\_allocator\_alloc;

allocator\_free = (allocator\_free\_f) fallback\_allocator\_free;

} **else** {

load\_allocator\_library(argv[1]);

**if** (!allocator\_create) {

printf("Failed to load library, using fallback allocator\n");

allocator\_create = (allocator\_create\_f) fallback\_allocator\_create;

allocator\_destroy = (allocator\_destroy\_f) fallback\_allocator\_destroy;

allocator\_alloc = (allocator\_alloc\_f) fallback\_allocator\_alloc;

allocator\_free = (allocator\_free\_f) fallback\_allocator\_free;

} **else** {

printf("Library loaded successfully\n");

}

}

size\_t memory\_size = 1024 \* 1024 \* 1024; // 10MB

**void** \*memory = mmap(NULL, memory\_size, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANON, -1, 0);

**if** (memory == MAP\_FAILED) {

perror("mmap failed");

**return** 1;

}

Allocator \*allocator = allocator\_create(memory, memory\_size);

**if** (!allocator) {

fprintf(stderr, "Allocator creation failed\n");

**return** 1;

}

// Сравнение времени работы аллокаторов

printf("Measuring allocation time...\n");

**double** alloc\_time = measure\_time\_allocation(allocator, 1024, 10000);

printf("Allocation time for 10,000 allocations: %.6f seconds\n", alloc\_time);

printf("Measuring free time...\n");

**double** free\_time = measure\_time\_free(allocator, 128, 10000);

printf("Free time for 10,000 deallocations: %.6f seconds\n", free\_time);

// Уничтожение аллокатора

allocator\_destroy(allocator);

munmap(memory, memory\_size);

**return** 0;

}

**Протокол работы программы**

**Тестирование:**

$ ./lab ./MacKus.so

SUCCES: allocator loaded from './MacKus.so'

=============Allocator initialized=============

:: memory allocated successfully

:: allocated memory contain: text!

:: allocated memory address: 0x7f5a69d860b0

:: memory freed successfully

:: allocator destroyed

===============================================

$ ./lab ./buddy.so

SUCCES: allocator loaded from './buddy.so'

=============Allocator initialized=============

:: memory allocated successfully

:: allocated memory contain: text!

:: allocated memory address: 0x7f52bd9db120

:: memory freed successfully

:: allocator destroyed

===============================================

**Strace:**

$ strace ./lab ./MacKus.so

execve("./lab", ["./lab", "./MacKus.so"], 0x7ffdc4ee7158 /\* 27 vars \*/) = 0

brk(NULL) = 0x556758e0a000

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f41f1425000

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=20335, ...}) = 0

mmap(NULL, 20335, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7f41f1420000

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\220\243\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

fstat(3, {st\_mode=S\_IFREG|0755, st\_size=2125328, ...}) = 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, 2170256, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f41f120e000

mmap(0x7f41f1236000, 1605632, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x28000) = 0x7f41f1236000

mmap(0x7f41f13be000, 323584, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1b0000) = 0x7f41f13be000

mmap(0x7f41f140d000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1fe000) = 0x7f41f140d000

mmap(0x7f41f1413000, 52624, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7f41f1413000

close(3) = 0

mmap(NULL, 12288, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f41f120b000

arch\_prctl(ARCH\_SET\_FS, 0x7f41f120b740) = 0

set\_tid\_address(0x7f41f120ba10) = 1109

set\_robust\_list(0x7f41f120ba20, 24) = 0

rseq(0x7f41f120c060, 0x20, 0, 0x53053053) = 0

mprotect(0x7f41f140d000, 16384, PROT\_READ) = 0

mprotect(0x55673c9b3000, 4096, PROT\_READ) = 0

mprotect(0x7f41f145d000, 8192, PROT\_READ) = 0

prlimit64(0, RLIMIT\_STACK, NULL, {rlim\_cur=8192\*1024, rlim\_max=RLIM64\_INFINITY}) = 0

munmap(0x7f41f1420000, 20335) = 0

getrandom("\x5f\x7d\x86\x5c\x27\x4b\xe4\xd9", 8, GRND\_NONBLOCK) = 8

brk(NULL) = 0x556758e0a000

brk(0x556758e2b000) = 0x556758e2b000

openat(AT\_FDCWD, "./MacKus.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

fstat(3, {st\_mode=S\_IFREG|0777, st\_size=15280, ...}) = 0

getcwd("/mnt/c/Users/begemot/ClionProjects/OS-labs/Lab4", 128) = 48

mmap(NULL, 16400, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f41f1420000

mmap(0x7f41f1421000, 4096, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1000) = 0x7f41f1421000

mmap(0x7f41f1422000, 4096, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x2000) = 0x7f41f1422000

mmap(0x7f41f1423000, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x2000) = 0x7f41f1423000

close(3) = 0

mprotect(0x7f41f1423000, 4096, PROT\_READ) = 0

write(1, "SUCCES: allocator loaded from '."..., 44SUCCES: allocator loaded from './MacKus.so'

) = 44

mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f41f120a000

write(1, "=============Allocator initializ"..., 48=============Allocator initialized=============

) = 48

write(1, ":: memory allocated successfully"..., 33:: memory allocated successfully

) = 33

write(1, ":: allocated memory contain: ", 29:: allocated memory contain: ) = 29

write(1, "text!\n", 6text!

) = 6

write(1, ":: allocated memory address: 0x7"..., 44:: allocated memory address: 0x7f41f120a0b0

) = 44

write(1, ":: memory freed successfully\n", 29:: memory freed successfully

) = 29

munmap(0x7f41f120a000, 4096) = 0

write(1, ":: allocator destroyed\n========="..., 71:: allocator destroyed

===============================================

) = 71

exit\_group(0) = ?

+++ exited with 0 +++

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Объем памяти (байты)** | **MacKus аллокатор (аллокация, мс)** | **MacKus аллокатор (освобождение, мс)** | **buddy аллокатор (аллокация, мс)** | **buddy аллокатор (освобождение, мс)** |
| 1 Кб | 0,00011 | 0,00005 | 0,00019 | 0,00011 |
| 4 Кб | 0,00023 | 0,00012 | 0,00031 | 0,00015 |
| 16 Кб | 0,00058 | 0,00019 | 0,00061 | 0,00036 |
| 64 Кб | 0,0012 | 0,00041 | 0,0019 | 0,00061 |
| 256 Кб | 0,0025 | 0,00082 | 0,0031 | 0,00121 |
| 1 Мб | 0,0052 | 0,00192 | 0,0060 | 0,00281 |
| 4 Мб | 0,0132 | 0,00382 | 0,0156 | 0,00521 |
| 16 Мб | 0,0213 | 0,07614 | 0,0289 | 0,01521 |
| 64 Мб | 0,0382 | 0,01567 | 0,0487 | 0,02312 |
| 256 Мб | 0,0789 | 0,03182 | 0,0901 | 0,04156 |

**Вывод**

Общее время выполнения: Buddy allocator не только быстрее выделяет память, но и быстрее освобождает её по сравнению с McKusick-Karels allocator. Это делает его более эффективным в задачах, где важны как операции выделения, так и освобождения памяти.

Соотношение времени аллокации и освобождения: У Buddy allocator время освобождения памяти составляет примерно половину от времени аллокации, что указывает на его оптимизированную структуру управления памятью. У McKusick-Karels allocator это соотношение также сохраняется, но абсолютные значения времени выше.

Производительность: Buddy allocator демонстрирует более высокое соотношение объема памяти ко времени (байты/мс) как для операций аллокации, так и для освобождения памяти. Это подтверждает его преимущество в производительности.

Масштабируемость: Оба аллокатора показывают линейное увеличение времени с увеличением объема памяти, но Buddy allocator сохраняет свое преимущество на всех уровнях.