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: Заворотный А. А. Преподаватель: Бахарев В.Д. Оценка:

Дата: 27.12.24

Москва, 2024

# Постановка задачи

## Вариант 1.

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 – Списки свободных блоков (первое подходящее) и блоки по 2^n.

**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 main 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 оборачивающими mmap и munmap. Если загрузить библиотеку удалось, то программа пытается найти в библиотеке символ соответствующий функции и присвоить указатель на него указателю на функцию. Если символа нет, то соответствующему указателю на функцию присвоен указатель на функцию оборачивающую mmap или munmap. В функции main load\_allocator вызывается с параметром argv[1]. Далее демонстрируется работа загруженных функции на примере работы с массивами.

Библиотека list\_allocator реализует аллокатор на основании метода списков свободных блоков (первое подходящее). В этом аллокаторе память выделяется блоками, размером 2^n. Аллокатор перебирает список свободных блоков памяти и ищет первый блок, размер которого достаточен для удовлетворения запроса на выделение памяти. Как только будет найден подходящий блок, он будет выделен для использования. Если подходящий блок будет найден, аллокатор уменьшает размер этого блока, оставляя часть памяти, которая не была выделена, как свободный. При освобождении блока аллокатор добавляет этот блок в список свободных. Может иметь высокую степень фрагментации, так как свободные блоки могут оставаться маленькими и несоответствующими для будущих запросов на выделение, что приводит к низкому фактору использования, но обладает высокой скоростью выделения за счет простоты реализации. Allocator - Хранит указатель на начало выделенной памяти, размер выделенной памяти и указатель на список свободных блоков. FreeBlock: - Структура для представления свободного блока памяти, содержит размер блока и указатель на следующий свободный блок. Функция allocator\_create эта функция инициализирует аллокатор: Проверяет корректность входных параметров. Устанавливает начальные значения для полей структуры Allocator. Создает первый свободный блок с полным размером доступной памяти. Функция allocator\_destroy эта функция очищает аллокатор: Устанавливает указатели на ноль, чтобы избежать утечек памяти. Функция my\_malloc эта функция выделяет память: Проверяет, валиден ли аллокатор и запрашивается ли ненулевая память. Проходит по списку свободных блоков, чтобы найти подходящий. Если находит блок, проверяет, нужно ли его разбить на меньшие компоненты. Возвращает указатель на выделенный участок памяти. Функция my\_free эта функция освобождает память: Проверяет входные параметры. Возвращает блок обратно в список свободных блоков, добавляя его в начало.

Библиотека blocks2^n реализует аллокатор памяти, использующий стратегию управления памятью на основе фиксированных размеров блоков (степени двойки). Основная идея аллокатора заключается в том, чтобы минимизировать фрагментацию памяти и обеспечить эффективное выделение и освобождение блоков памяти. Создание аллокатора осуществляется на заранее выделенном участке памяти. Память разделяется на блоки фиксированных размеров, кратных степеням двойки (от 1 байта до 1024 байт). Каждый размер блоков хранится в отдельном списке свободных блоков (freeBlocks), где каждый элемент списка представляет отдельный блок памяти.

Аллокатор инициализируется, разбивая память на блоки различных размеров (1 байт, 2 байта, 4 байта и т.д.) и записывая их в связанные списки свободных блоков. Используется массив списков, где индекс массива соответствует размеру блока в степенях двойки. Каждый блок памяти представлен структурой Block, которая содержит указатели на предыдущий и следующий блоки, а также информацию о размере блока. Функция выделения памяти (my\_malloc) выделяет блок памяти запрашиваемого размера, выбирая его из соответствующего списка свободных блоков. Аллокатор ищет необходимый размер блока, используя степень двойки, чтобы найти минимальный блок, который может вместить запрашиваемый размер. Если в списке свободных блоков нужного размера есть доступный блок, он извлекается и возвращается. Если подходящего блока нет, аллокатор пытается разделить более крупный блок на два меньших с помощью функции split\_block. Если нет свободных блоков нужного размера, более крупный блок из списка свободных блоков может быть разделён на два блока меньшего размера. Эти меньшие блоки затем помещаются в соответствующий список свободных блоков. Функция освобождения памяти (my\_free) возвращает выделенный блок обратно в список свободных блоков. При освобождении памяти, указатель на блок добавляется обратно в список свободных блоков соответствующего размера. Блок становится доступным для повторного использования. Освобожденный блок добавляется в начало соответствующего списка свободных блоков. Аллокатор освобождает всю выделенную память с помощью системного вызова munmap. Когда аллокатор больше не нужен, вся область памяти, которую он использует, освобождается с помощью вызова munmap, что завершает жизненный цикл аллокатора.

# Код программы

**main.c**

#include <stddef.h>

#include <stdlib.h>

#include <stdint.h>

#include <string.h>

#include <stdio.h>

#include <unistd.h>

#include <dlfcn.h>

#include <sys/mman.h>

#include <sys/time.h>

#include <time.h>

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

**void** \*(\*allocator\_create)(**void** \*addr, size\_t size);

**void** \*(\*my\_malloc)(**void** \*allocator, size\_t size);

**void** (\*my\_free)(**void** \*allocator, **void** \*ptr);

**void** (\*allocator\_destroy)(**void** \*allocator);

} Allocator;

**long** get\_current\_time\_ns() {

**struct** timespec ts;

clock\_gettime(CLOCK\_MONOTONIC, &ts);

**return** (ts.tv\_sec \* 1000000000) + ts.tv\_nsec;

}

**void** \*default\_allocator\_create(**void** \*memory, size\_t size) {

(**void**)size;

(**void**)memory;

**return** memory;

}

**void** \*default\_my\_malloc(**void** \*allocator, size\_t size) {

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** default\_my\_free(**void** \*allocator, **void** \*memory) {

**if** (memory == NULL)

**return**;

uint32\_t \*mem = (uint32\_t \*)memory - 1;

munmap(mem, \*mem);

}

**void** default\_allocator\_destroy(**void** \*allocator) {

(**void**)allocator;

}

Allocator \*load\_allocator(**const** **char** \*library\_path) {

**if** (library\_path == NULL || library\_path[0] == '\0') {

**char** message[] = "WARNING: failed to load library (default allocator will be used)\n";

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

Allocator \*allocator = malloc(**sizeof**(Allocator));

allocator->allocator\_create = default\_allocator\_create;

allocator->my\_malloc = default\_my\_malloc;

allocator->my\_free = default\_my\_free;

allocator->allocator\_destroy = default\_allocator\_destroy;

**return** allocator;

}

**void** \*library = dlopen(library\_path, RTLD\_LOCAL | RTLD\_NOW);

**if** (!library) {

**char** message[] = "WARNING: failed to load library\n";

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

Allocator \*allocator = malloc(**sizeof**(Allocator));

allocator->allocator\_create = default\_allocator\_create;

allocator->my\_malloc = default\_my\_malloc;

allocator->my\_free = default\_my\_free;

allocator->allocator\_destroy = default\_allocator\_destroy;

**return** allocator;

}

**char** buffer[64];

snprintf(buffer, **sizeof**(buffer), "SUCCESS: allocator loaded from \'%s\'\n", library\_path);

write(STDOUT\_FILENO, buffer, strlen(buffer));

Allocator \*allocator = malloc(**sizeof**(Allocator));

allocator->allocator\_create = dlsym(library, "allocator\_create");

allocator->my\_malloc = dlsym(library, "my\_malloc");

allocator->my\_free = dlsym(library, "my\_free");

allocator->allocator\_destroy = dlsym(library, "allocator\_destroy");

**if** (!allocator->allocator\_create || !allocator->my\_malloc || !allocator->my\_free || !allocator->allocator\_destroy) {

**const** **char** msg[] = "ERROR: failed to load all allocator functions\n";

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

free(allocator);

dlclose(library);

**return** NULL;

}

**return** allocator;

}

**int** test\_allocator(**const** **char** \*library\_path) {

srand(time(NULL));

Allocator \*allocator\_api = load\_allocator(library\_path);

**if** (!allocator\_api) **return** -1;

size\_t size = 1024 \* 32;

**void** \*addr = mmap(NULL, size, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0);

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

**char** message[] = "ERROR: mmap failed\n";

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

free(allocator\_api);

**return** EXIT\_FAILURE;

}

**void** \*allocator = allocator\_api->allocator\_create(addr, size);

**if** (!allocator) {

**char** message[] = "ERROR: failed to initialize allocator\n";

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

munmap(addr, size);

free(allocator\_api);

**return** EXIT\_FAILURE;

}

**char** start\_message[] = "=============Allocator initialized=============\n";

write(STDOUT\_FILENO, start\_message, **sizeof**(start\_message) - 1);

**long** **double** sum\_allocating\_time = 0.0, sum\_freeing\_time = 0.0, start\_time, end\_time;

**int** num\_alloc, start\_alloc = 0, num\_free, start\_free = 0, tek\_num\_alloc\_need = 100, tek\_num\_free\_can = 0;

**void** \* list\_allocated\_memory[100];

**while** (1){

**if** (tek\_num\_alloc\_need != 0){

num\_alloc = rand() % tek\_num\_alloc\_need;

**if** (num\_alloc == 0){

num\_alloc = 1;

}

**for** (**int** i = start\_alloc; i < start\_alloc + num\_alloc; i++){

start\_time = get\_current\_time\_ns();

list\_allocated\_memory[i] = allocator\_api->my\_malloc(allocator, 3);

end\_time = get\_current\_time\_ns();

sum\_allocating\_time += (end\_time - start\_time);

**if** (list\_allocated\_memory[i] == NULL) {

**char** alloc\_fail\_message[] = "ERROR: memory allocation failed\n";

write(STDERR\_FILENO, alloc\_fail\_message, **sizeof**(alloc\_fail\_message) - 1);

**return** EXIT\_FAILURE;

}

}

start\_alloc += num\_alloc;

tek\_num\_alloc\_need -= num\_alloc;

tek\_num\_free\_can += num\_alloc;

}

**if** (tek\_num\_free\_can != 0){

num\_free = rand() % tek\_num\_free\_can;

**if** (num\_free == 0){

num\_free = 1;

}

**for** (**int** i = start\_free; i < start\_free + num\_free; i++){

start\_time = get\_current\_time\_ns();

allocator\_api->my\_free(allocator, list\_allocated\_memory[i]);

end\_time = get\_current\_time\_ns();

sum\_freeing\_time += (end\_time - start\_time);

}

start\_free += num\_free;

tek\_num\_free\_can -= num\_free;

}

**else**{

**break**;

}

}

sum\_allocating\_time /= 1000000.0;

sum\_freeing\_time /= 1000000.0;

**char** time\_message[100];

snprintf(time\_message, **sizeof**(time\_message), "Allocation took %Lf ms and freeing took %Lf ms.\n", sum\_allocating\_time / 100.0, sum\_freeing\_time / 100.0);

write(STDOUT\_FILENO, time\_message, strlen(time\_message));

allocator\_api->allocator\_destroy(allocator);

free(allocator\_api);

munmap(addr, size);

**char** exit\_message[] = "=============Allocator destroyed===============\n";

write(STDOUT\_FILENO, exit\_message, **sizeof**(exit\_message) - 1);

**return** EXIT\_SUCCESS;

}

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

**const** **char** \*library\_path = (argc > 1) ? argv[1] : NULL;

**if** (test\_allocator(library\_path))

**return** EXIT\_FAILURE;

**return** EXIT\_SUCCESS;

}

**blocks2^n.c**

#include <stdlib.h>

#include <math.h>

#include <string.h>

#include <unistd.h>

#include <sys/mman.h>

#include <unistd.h>

#define MAX\_BLOCK\_SIZE\_EXTENT 10 // Максимальный размер блока (2^10 = 1024)

#define MIN\_BLOCK\_SIZE\_EXTENT 0 // Минимальный размер блока (2^0 = 1)

#define NUM\_LISTS 11 // Количество списков (от 0 до 10)

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

**struct** Block \*next;

**struct** Block \*prev;

size\_t size;

} Block;

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

Block\* freeLists[NUM\_LISTS];

**void** \*memory;

size\_t total\_size;

} Allocator;

**int** power(**int** base, **int** exp) {

**long** **long** result = 1;

**while** (exp > 0) {

**if** (exp % 2 == 1) {

result \*= base;

}

base \*= base;

exp /= 2;

}

**return** result;

}

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

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

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

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

size\_t offset = 0;

size\_t extent = 0;

**for** (**int** i = 0; i < 10; ++i) {

allocator->freeLists[i] = NULL;

}

**while** (offset + power(2, extent / 10) <= allocator->total\_size) {

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

**if** (allocator->freeLists[extent / 10] == NULL) {

block->next = NULL;

allocator->freeLists[extent / 10] = block;

} **else** {

allocator->freeLists[extent / 10]->prev = block;

}

block->next = allocator->freeLists[extent / 10];

block->size = power(2, extent / 10);

offset += power(2, extent / 10);

extent++;

}

**return** allocator;

}

**void** split\_block(Allocator \*allocator, Block\* block) {

**int** ind = 0;

**while** ((1 << ind) < block->size) {

ind++;

}

Block \*block\_copy = (Block \*)((**char** \*)allocator->memory + block->size / 2);

**if** (allocator->freeLists[ind] == NULL) {

block->next = NULL;

allocator->freeLists[ind] = block;

allocator->freeLists[ind]->prev = block\_copy;

} **else** {

allocator->freeLists[ind]->prev = block;

block->next = allocator->freeLists[ind];

allocator->freeLists[ind]->prev = block\_copy;

block\_copy->next = allocator->freeLists[ind];

}

block->size = power(2, ind);

block\_copy->size = power(2, ind);

}

**void**\* my\_malloc(Allocator \*allocator, size\_t size) {

**int** ind = 0;

**while** ((1 << ind) < size) {

ind++;

}

**if** (ind >= NUM\_LISTS) **return** NULL;

**if** (allocator->freeLists[ind] != NULL) {

Block \*block = allocator->freeLists[ind];

allocator->freeLists[ind] = block->next;

**return** block;

}

**for** (**int** i = 0; i < 10 - ind; ++i) {

**if** (allocator->freeLists[ind + i] != NULL) {

**int** j = 0;

**while** (i != j) {

Block \*block = allocator->freeLists[ind + i - j];

allocator->freeLists[ind + i - j] = block->next;

split\_block(allocator, block);

}

**break**;

}

}

**return** NULL;

}

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

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

{

**return**;

}

**int** ind = 0;

**while** ((1 << ind) < ((Block\*)ptr)->size) {

ind++;

}

allocator->freeLists[ind] = ((Block\*)ptr)->next;

ptr = NULL;

}

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

**if** (!allocator)

{

**return**;

}

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

{

exit(EXIT\_FAILURE);

}

}

**list\_allocator.c**

#include <stddef.h>

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

**void**\* memory\_start;

size\_t memory\_size;

**void**\* free\_list;

} Allocator;

**typedef** **struct** FreeBlock {

size\_t size;

**struct** FreeBlock\* next;

} FreeBlock;

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

**if** (!memory || size < **sizeof**(FreeBlock)) {

**return** NULL;

}

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

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

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

allocator->free\_list = allocator->memory\_start;

FreeBlock\* initial\_block = (FreeBlock\*)allocator->memory\_start;

initial\_block->size = allocator->memory\_size;

initial\_block->next = NULL;

**return** allocator;

}

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

allocator->memory\_start = NULL;

allocator->memory\_size = 0;

allocator->free\_list = NULL;

}

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

**if** (!allocator || size == 0) {

**return** NULL;

}

FreeBlock\* prev = NULL;

FreeBlock\* curr = (FreeBlock\*)allocator->free\_list;

**while** (curr) {

**if** (curr->size >= size + **sizeof**(FreeBlock)) {

**if** (curr->size > size + **sizeof**(FreeBlock)) {

FreeBlock\* new\_block = (FreeBlock\*)((**char**\*)curr + **sizeof**(FreeBlock) + size);

new\_block->size = curr->size - size - **sizeof**(FreeBlock);

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

curr->size = size;

curr->next = new\_block;

}

**if** (prev) {

prev->next = curr->next;

} **else** {

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

}

**return** (**char**\*)curr + **sizeof**(FreeBlock);

}

prev = curr;

curr = curr->next;

}

**return** NULL;

}

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

**if** (!allocator || !memory) {

**return**;

}

FreeBlock\* block = (FreeBlock\*)((**char**\*)memory - **sizeof**(FreeBlock));

block->next = (FreeBlock\*)allocator->free\_list;

allocator->free\_list = block;

}

# Протокол работы программы

## Тестирование:

$ ./main ./list\_allocator.so

SUCCESS: allocator loaded from './list\_allocator.so'

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

Allocation took 0.000178 ms and freeing took 0.000034 ms.

=============Allocator destroyed===============

## Strace:

$ strace -f ./main ./list\_allocator.so

execve("./main", ["./main", "./list\_allocator.so"], 0x7ffc94c461f0 /\* 27 vars \*/) = 0

brk(NULL) = 0x55980d2f6000

**mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fb98f91e000**

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=20071, ...}) = 0

**mmap(NULL, 20071, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7fb98f919000**

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) = 0x7fb98f707000**

**mmap(0x7fb98f72f000, 1605632, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x28000) = 0x7fb98f72f000**

**mmap(0x7fb98f8b7000, 323584, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1b0000) = 0x7fb98f8b7000**

**mmap(0x7fb98f906000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1fe000) = 0x7fb98f906000**

**mmap(0x7fb98f90c000, 52624, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7fb98f90c000**

close(3) = 0

**mmap(NULL, 12288, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fb98f704000**

arch\_prctl(ARCH\_SET\_FS, 0x7fb98f704740) = 0

set\_tid\_address(0x7fb98f704a10) = 78778

set\_robust\_list(0x7fb98f704a20, 24) = 0

rseq(0x7fb98f705060, 0x20, 0, 0x53053053) = 0

mprotect(0x7fb98f906000, 16384, PROT\_READ) = 0

mprotect(0x559804b5b000, 4096, PROT\_READ) = 0

mprotect(0x7fb98f956000, 8192, PROT\_READ) = 0

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

**munmap(0x7fb98f919000, 20071) = 0**

getrandom("\x59\xe5\x01\xd1\x9a\x1c\xc7\xf8", 8, GRND\_NONBLOCK) = 8

brk(NULL) = 0x55980d2f6000

brk(0x55980d317000) = 0x55980d317000

openat(AT\_FDCWD, "./list\_allocator.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=15240, ...}) = 0

getcwd("/mnt/c/Users/\320\220\320\273\320\265\320\272\321\201\320\265\320\271/CLionProjects/Labs\_osi\_Zavorotnyi\_A\_M8O-214--23/Lab4", 128) = 81

**mmap(NULL, 16400, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7fb98f919000**

**mmap(0x7fb98f91a000, 4096, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1000) = 0x7fb98f91a000**

**mmap(0x7fb98f91b000, 4096, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x2000) = 0x7fb98f91b000**

**mmap(0x7fb98f91c000, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x2000) = 0x7fb98f91c000**

close(3) = 0

mprotect(0x7fb98f91c000, 4096, PROT\_READ) = 0

write(1, "SUCCESS: allocator loaded from '"..., 53SUCCESS: allocator loaded from './list\_allocator.so'

) = 53

**mmap(NULL, 32768, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fb98f6fc000**

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

) = 48

write(1, "Allocation took 0.000197 ms and "..., 58Allocation took 0.000197 ms and freeing took 0.000036 ms.

) = 58

**munmap(0x7fb98f6fc000, 32768) = 0**

write(1, "=============Allocator destroyed"..., 48=============Allocator destroyed===============

) = 48

exit\_group(0) = ?

+++ exited with 0 +++

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Объем памяти (байты) | **list allocator (аллокация, мс)** | **list allocator (освобождение, мс)** | **blocks2^n (аллокация. мс)** | **blocks2^n (освобождение, мс)** |
| 1 B | 0.000056 | 0.000034 | 0.000037 | 0.000095 |
| 4 B | 0.000104 | 0.000035 | 0.000041 | 0.000095 |
| 16 B | 0.000158 | 0.000033 | 0.000053 | 0.000042 |
| 64 B | 0.000170 | 0.000033 | 0.000053 | 0.000043 |
| 256 B | 0.000299 | 0.000037 | 0.000071 | 0.000047 |
| 1 KB | 0.000556 | 0.000036 | 0.000075 | 0.000055 |

# Вывод

В рамках лабораторной работы была разработана и отлажена программа на языке C, которая загружает динамическую библиотеку по пути, переданному через аргументы командной строки. Помимо этого, были изучены два разных аллокатора и созданы динамические библиотеки, реализующие их. В целом, если рассматривать скорость выделения и освобождения памяти, то алгоритм с блоками по 2^n обычно работает быстрее в случаях с регулярными запросами, так как поиск нужного блока упрощается.

Внутренние потери: Списки свободных блоков (первое подходящее): В этом алгоритме внутренние потери могут возникнуть, если размер запрашиваемого блока меньше, чем размер самого свободного блока, который был выделен. Если фрагменты памяти имеют переменные размеры, то это может приводить к большим внутренним потерям. Блоки по 2^n: Здесь внутренние потери более предсказуемы, при выделении блока фиксированного размера, если запрашиваемый размер не соответствует размеру блока, будет всегда фиксированная потеря.

Внешние потери: Списки свободных блоков (первое подходящее): Этот алгоритм подвержен внешним потерям из-за фрагментации: если память разбивается на множество маленьких блоков, может не оказаться достаточно больших блоков для выполнения запроса даже при наличии свободной памяти. Блоки по 2^n: Алгоритм с фиксированными размерами блоков может несколько снижать внешние потери, поскольку блоки стандартизированы и, если определённый размер доступен, он может быть использован без необходимости искать подходящий блок. Однако, он также может столкнуться с внешними потерями, если запросы на память разнообразны и не могут быть удовлетворены доступными размерами.