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: Савинов Н. О.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 26.12.24

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 двойников и алгоритм Мак-Кьюзика-Кэрелса.

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 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 (mem) и её размер (size), проверяет, что size — степень двойки, выделяет часть памяти для структуры аллокатора (Allocator) и корневого узла (BuddyNode), 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 allocator_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.

Библиотека allocator реализует аллокатор на основе алгоритма Мак-Кьюзика-Кэрелса. Это одна из реализаций механизмов управления памятью, основанная на разделении памяти на фиксированные размеры блоков (slabs). Основная идея состоит в том, чтобы минимизировать фрагментацию памяти и обеспечить быструю аллокацию и освобождение. Память организована в пулы (pools), называемые "аренами", которые содержат блоки фиксированного размера. Для каждого размера блока создается отдельный пул. Для управления состоянием блоков в пуле используется битовая карта (bitmap), где каждый бит указывает, занят блок или свободен. Если для запрашиваемого размера уже существует пул, аллокация происходит из него. Если нет подходящего пула, система создает новый, резервируя область памяти для него. Создание аллокатора(allocator_create): инициализирует аллокатор на предоставленном участке памяти, делит память на несколько пулов, каждый из которых предназначен для блоков определенного размера,распределяет битовые карты и области для блоков в каждом пуле. Выделение памяти (allocator_alloc): ищет подходящий пул, минимальный по размеру, который способен вместить запрошенный блок памяти, проверяет битовую карту пула, чтобы найти свободный блок, если свободный блок найден, отмечает его как занятый в битовой карте и возвращает указатель на него,если подходящий пул не найден или все блоки заняты, возвращается NULL. Освобождение памяти (allocator_free): находит пул, к которому относится переданный указатель, рассчитывает индекс блока в пуле по переданному указателю, помечает соответствующий бит в битовой карте как свободный. Уничтожение аллокатора (allocator_destroy): освобождает память, переданную аллокатору, с использованием системного вызова munmap.

Код программы

main.c

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

#define MEMORY_SIZE 1024 * 1024

typedef struct Allocator {
    void* memory_start;
    size_t memory_size;
    void* free_list;
} Allocator;

Allocator* allocator_create(void* const memory, const size_t size);
void allocator_destroy(Allocator* const allocator);
void* allocator_alloc(Allocator* const allocator, const size_t size);
void allocator_free(Allocator* const allocator, void* const memory);

void my_write(const char* message) {
    write(STDERR_FILENO, message, strlen(message));
}

void my_write_hex(void* ptr) {
    char buffer[64];
    unsigned long addr = (unsigned long)ptr;
    size_t i;
    for (i = 0; i < sizeof(buffer) - 1 && addr; ++i) {
        unsigned char byte = addr & 0xF;
```

```
        buffer[i] = (byte < 10) ? '0' + byte : 'a' + (byte - 10);
        addr >>= 4;
    }
    buffer[i] = '\0';
    my_write(buffer);
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        my_write("Usage: <path_to_allocator_library>\n");
        return 1;
    }

    void* handle = dlopen(argv[1], RTLD_LAZY);
    if (!handle) {
        my_write("Failed to load library: ");
        my_write(dlerror());
        my_write("\n");
        return 1;
    }

    Allocator* (*allocator_create)(void*, size_t) = dlsym(handle, "allocator_create");
    void (*allocator_destroy)(Allocator*) = dlsym(handle, "allocator_destroy");
    void* (*allocator_alloc)(Allocator*, size_t) = dlsym(handle, "allocator_alloc");
    void (*allocator_free)(Allocator*, void*) = dlsym(handle, "allocator_free");

    char* error;
    if ((error = dlerror()) != NULL) {
        my_write("Error resolving symbols: ");
        my_write(error);
```

```
        my_write("\n");
        dlclose(handle);
        return 1;
    }

    void* memory = mmap(NULL, MEMORY_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
    if (memory == MAP_FAILED) {
        my_write("mmap failed\n");
        dlclose(handle);
        return 1;
    }

    Allocator* allocator = allocator_create(memory, MEMORY_SIZE);
    if (!allocator) {
        my_write("Failed to create allocator\n");
        munmap(memory, MEMORY_SIZE);
        dlclose(handle);
        return 1;
    }

    void* block = allocator_alloc(allocator, 128);
    if (block) {
        my_write("Allocated block at ");
        my_write_hex(block);
        my_write("\n");
    } else {
        my_write("Failed to allocate block\n");
    }
```

```
    allocator_free(allocator, block);
    my_write("Freed block\n");

    allocator_destroy(allocator);
    munmap(memory, MEMORY_SIZE);
    dlclose(handle);

    return 0;
}
```

buddy.c

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

#ifdef _MSC_VER
#define EXPORT __declspec(dllexport)
#else
#define EXPORT
#endif

typedef struct BuddyNode {
    int size;
    int free;
    struct BuddyNode *left;
    struct BuddyNode *right;
} BuddyNode;

typedef struct Allocator {
    BuddyNode *root;
    void *memory;
    int totalSize;
    int offset;
} Allocator;

// Проверка, является ли число степенью двойки
int is_power_of_two(unsigned int n) {
    return (n > 0) && ((n & (n - 1)) == 0);
```

```
}

// Создание нового узла
BuddyNode *create_node(Allocator *allocator, int size) {
    if ((size_t)(allocator->offset + sizeof(BuddyNode)) > (size_t)(allocator->totalSize)) {
        return NULL;
    }

    BuddyNode *node = (BuddyNode *)((char *)allocator->memory + allocator->offset);
    allocator->offset += sizeof(BuddyNode);

    node->size = size;
    node->free = 1;
    node->left = NULL;
    node->right = NULL;

    return node;
}

// Создание аллокатора
EXPORT Allocator *allocator_create(void *mem, size_t size) {
    if (!is_power_of_two(size)) {
        const char msg[] = "This allocator requires a power of two size\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return NULL;
    }

    Allocator *allocator = (Allocator *)mem;
    allocator->memory = (char *)mem + sizeof(Allocator);
    allocator->totalSize = size - sizeof(Allocator);
```

```
    allocator->offset = 0;

    allocator->root = create_node(allocator, size);
    return allocator->root ? allocator : NULL;
}

// Разделение узла на два "двойника"
void split_node(Allocator *allocator, BuddyNode *node) {
    int newSize = node->size / 2;

    node->left = create_node(allocator, newSize);
    node->right = create_node(allocator, newSize);
}

// Рекурсивное выделение памяти
BuddyNode *allocate_recursive(Allocator *allocator, BuddyNode *node, int size) {
    if (!node || node->size < size || !node->free) {
        return NULL;
    }

    if (node->size == size) {
        node->free = 0;
        return node;
    }

    if (!node->left) {
        split_node(allocator, node);
    }

    BuddyNode *allocated = allocate_recursive(allocator, node->left, size);
```

```
    if (!allocated) {
        allocated = allocate_recursive(allocator, node->right, size);
    }

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

// Выделение памяти
EXPORT void *allocator_alloc(Allocator *allocator, size_t size) {
    if (!allocator || size <= 0) {
        return NULL;
    }

    while (!is_power_of_two(size)) {
        size++;
    }

    return allocate_recursive(allocator, allocator->root, size);
}

// Освобождение памяти
EXPORT void allocator_free(Allocator *allocator, void *ptr) {
    if (!allocator || !ptr) {
        return;
    }

    BuddyNode *node = (BuddyNode *)ptr;
    if (!node) {
        return;
```

```
    }

    node->free = 1;

    if (node->left && node->left->free && node->right && node->right->free) {
        allocator_free(allocator, node->left);
        allocator_free(allocator, node->right);
        node->left = NULL;
        node->right = NULL;
    }
}

// Уничтожение аллокатора
EXPORT void allocator_destroy(Allocator *allocator) {
    if (!allocator) {
        return;
    }

    allocator_free(allocator, allocator->root);

    if (munmap((void *)allocator, allocator->totalSize + sizeof(Allocator)) == -1) {
        exit(EXIT_FAILURE);
    }
}
```

allocator.c

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

#ifdef _MSC_VER
#define EXPORT __declspec(dllexport)
#else
#define EXPORT
#endif

#define MAX_BLOCK_SIZES 4

typedef struct
{
    size_t block_size;     // Размер одного блока
    size_t block_count;    // Количество блоков
    uint8_t *bitmap;       // Битовая карта для управления блоками
    uint8_t *memory_start; // Указатель на начало памяти для этого пула
} MemoryPool;

typedef struct
{
    void *memory;                   // Указатель на переданную память
    size_t memory_size;               // Размер памяти
    MemoryPool pools[MAX_BLOCK_SIZES]; // Пулы для разных размеров блоков
} Allocator;
```

```
static void write_message(const char *message)
{
    write(STDERR_FILENO, message, strlen(message));
}

EXPORT Allocator *allocator_create(void *mem, size_t mem_size)
{
    Allocator *allocator = (Allocator *)malloc(sizeof(Allocator));
    if (!allocator)
        return NULL;

    allocator->memory = mem;
    allocator->memory_size = mem_size;

    size_t block_sizes[MAX_BLOCK_SIZES] = {16, 32, 64, 128};

    uint8_t *current_memory = mem;
    for (int i = 0; i < MAX_BLOCK_SIZES; i++)
    {
        size_t block_size = block_sizes[i];
        size_t block_count = mem_size / block_size / MAX_BLOCK_SIZES;

        allocator->pools[i].block_size = block_size;
        allocator->pools[i].block_count = block_count;
        allocator->pools[i].bitmap = current_memory;
        allocator->pools[i].memory_start = current_memory + block_count / 8;

        memset(allocator->pools[i].bitmap, 0, block_count / 8);

        current_memory += block_count / 8 + block_count * block_size;
```

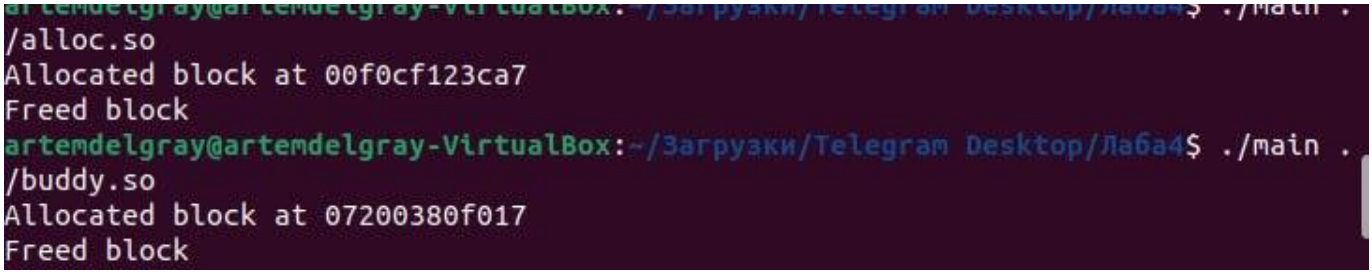
```
    }

    return allocator;
}

EXPORT void *allocator_alloc(Allocator *allocator, size_t size)
{
    for (int i = 0; i < MAX_BLOCK_SIZES; i++)
    {
        MemoryPool *pool = &allocator->pools[i];

        if (size > pool->block_size)
            continue;

        for (size_t j = 0; j < pool->block_count; j++)
        {
            size_t byte_index = j / 8;
            size_t bit_index = j % 8;

            if (!(pool->bitmap[byte_index] & (1 << bit_index)))
            {
                pool->bitmap[byte_index] |= (1 << bit_index);
                return pool->memory_start + j * pool->block_size;
            }
        }
    }

    return NULL;
}
```

```
EXPORT void allocator_free(Allocator *allocator, void *ptr)
{
    for (int i = 0; i < MAX_BLOCK_SIZES; i++)
    {
        MemoryPool *pool = &allocator->pools[i];

        if (ptr >= (void *)pool->memory_start && ptr < (void *)(pool->memory_start + pool->block_count
* pool->block_size))
        {
            size_t offset = (uint8_t *)ptr - pool->memory_start;
            size_t index = offset / pool->block_size;
            size_t byte_index = index / 8;
            size_t bit_index = index % 8;

            pool->bitmap[byte_index] &= ~(1 << bit_index);
            return;
        }
    }
}

EXPORT void allocator_destroy(Allocator *allocator)
{
    if (allocator)
    {
        if (munmap(allocator->memory, allocator->memory_size) == -1)
        {
            const char error_msg[] = "Error: munmap failed\n";
            write_message(error_msg);
            exit(EXIT_FAILURE);
        }
```

```
        free(allocator);
    }
}
```

Протокол работы программы

Тестирование:



Strace:

```
strace -f ./main ./buddy.so

execve("./main", ["./main", "./buddy.so"], 0x7fffa13c3340 /* 46 vars */) = 0

brk(NULL)                               = 0x5a439107f000

arch_prctl(0x3001 /* ARCH_??? */, 0x7fff325440f0) = -1 EINVAL (Недопустимый аргумент)

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x71e8fdb3e000

access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (Нет такого файла или каталога)

openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=58047, ...}, AT_EMPTY_PATH) = 0

mmap(NULL, 58047, PROT_READ, MAP_PRIVATE, 3, 0) = 0x71e8fdb2f000

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) = 0x71e8fd800000

mprotect(0x71e8fd828000, 2023424, PROT_NONE) = 0

mmap(0x71e8fd828000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x28000) = 0x71e8fd828000

mmap(0x71e8fd9bd000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) =
0x71e8fd9bd000

mmap(0x71e8fda16000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x215000) = 0x71e8fda16000

mmap(0x71e8fda1c000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1,
0) = 0x71e8fda1c000

close(3)                                = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x71e8fdb2c000

arch_prctl(ARCH_SET_FS, 0x71e8fdb2c740) = 0

set_tid_address(0x71e8fdb2ca10)         = 4039
```

```
set_robust_list(0x71e8fdb2ca20, 24)     = 0

rseq(0x71e8fdb2d0e0, 0x20, 0, 0x53053053) = 0

mprotect(0x71e8fda16000, 16384, PROT_READ) = 0

mprotect(0x5a438fcd2000, 4096, PROT_READ) = 0

mprotect(0x71e8fdb78000, 8192, PROT_READ) = 0

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

munmap(0x71e8fdb2f000, 58047)           = 0

getrandom("\xd5\xdb\xb7\xaa\x3c\xd1\xf0\x31", 8, GRND_NONBLOCK) = 8

brk(NULL)                               = 0x5a439107f000

brk(0x5a43910a0000)                     = 0x5a43910a0000

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|0775, st_size=16088, ...}, AT_EMPTY_PATH) = 0

getcwd("/home/artemdelgray/\320\227\320\260\320\263\321\200\321\203\320\267\320\272\320\270/
Telegram Desktop/\320\233\320\260\320\261\320\2604", 128) = 63

mmap(NULL, 16496, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x71e8fdb39000

mmap(0x71e8fdb3a000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x1000) = 0x71e8fdb3a000

mmap(0x71e8fdb3b000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) =
0x71e8fdb3b000

mmap(0x71e8fdb3c000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x2000) = 0x71e8fdb3c000

close(3)                                = 0

mprotect(0x71e8fdb3c000, 4096, PROT_READ) = 0

mmap(NULL, 1048576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x71e8fda2c000

write(2, "Allocated block at ", 19Allocated block at )     = 19

write(2, "072c2adf8e17", 12072c2adf8e17)            = 12

write(2, "\n", 1

)                       = 1

write(2, "Freed block\n", 12Freed block

)           = 12

munmap(0x71e8fda2c000, 1048576)         = 0

munmap(0x71e8fda2c000, 1048576)         = 0

munmap(0x71e8fdb39000, 16496)           = 0

exit_group(0)                           = ?
```

```
+++ exited with 0 +++
```

Вывод

В ходе выполнения лабораторной работы была составлена и отлажена программа на языке C, осуществляющая загрузку динамической библиотеки по пути, переданному в аргументах командной строки. Также были изучены два различных аллокатора и написаны динамические библиотеки, реализующие их.