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: Мустафаев А.Р

Преподаватель: Бахарев В.Д

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 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 mkk 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 <stddef.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <dlfcn.h>
#include <sys/mman.h>

typedef struct Allocator
{
    void *(*allocator_create)(void *addr, size_t size);
    void *(*allocator_alloc)(void *allocator, size_t size);
    void (*allocator_free)(void *allocator, void *ptr);
    void (*allocator_destroy)(void *allocator);
} Allocator;

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

void *standard_allocator_alloc(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 standard_allocator_free(void *allocator, void *memory)
{
    if (memory == NULL)
```

```
        return;
    uint32_t *mem = (uint32_t *)memory - 1;
    munmap(mem, *mem);
}

void standard_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 shared library\n";
        write(STDERR_FILENO, message, sizeof(message) - 1);
        Allocator *allocator = malloc(sizeof(Allocator));
        allocator->allocator_create = standard_allocator_create;
        allocator->allocator_alloc = standard_allocator_alloc;
        allocator->allocator_free = standard_allocator_free;
        allocator->allocator_destroy = standard_allocator_destroy;
        return allocator;
    }

    void *library = dlopen(library_path, RTLD_LOCAL | RTLD_NOW);
    if (!library)
    {
        char message[] = "WARNING: failed to load shared library\n";
        write(STDERR_FILENO, message, sizeof(message) - 1);
        Allocator *allocator = malloc(sizeof(Allocator));
        allocator->allocator_create = standard_allocator_create;
        allocator->allocator_alloc = standard_allocator_alloc;
        allocator->allocator_free = standard_allocator_free;
        allocator->allocator_destroy = standard_allocator_destroy;
        return allocator;
    }

    Allocator *allocator = malloc(sizeof(Allocator));
    allocator->allocator_create = dlsym(library, "allocator_create");
    allocator->allocator_alloc = dlsym(library, "allocator_alloc");
    allocator->allocator_free = dlsym(library, "allocator_free");
    allocator->allocator_destroy = dlsym(library, "allocator_destroy");

    if (!allocator->allocator_create || !allocator->allocator_alloc || !
```

```
allocator->allocator_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 main(int argc, char **argv)
{
    const char *library_path = (argc > 1) ? argv[1] : NULL;
    Allocator *allocator_api = load_allocator(library_path);
    if (!allocator_api)
    {
        char message[] = "Failed to load allocator API\n";
        write(STDERR_FILENO, message, sizeof(message) - 1);
        return EXIT_FAILURE;
    }

    size_t size = 4096;
    void *addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED)
    {
        char message[] = "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[] = "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);

    void *block1 = allocator_api->allocator_alloc(allocator, 64);
    void *block2 = allocator_api->allocator_alloc(allocator, 40);

    if (block1 == NULL || block2 == NULL)
    {
        char alloc_fail_message[] = "Memory allocation failed\n";
        write(STDERR_FILENO, alloc_fail_message,
sizeof(alloc_fail_message) - 1);
    }
    else
    {
        char alloc_success_message[] = "Memory allocated successfully\n";
        write(STDOUT_FILENO, alloc_success_message,
sizeof(alloc_success_message) - 1);
    }

    char buffer[64];
    snprintf(buffer, sizeof(buffer), "Block 1 address: %p\n", block1);
    write(STDOUT_FILENO, buffer, strlen(buffer));
    snprintf(buffer, sizeof(buffer), "Block 2 address: %p\n", block2);
    write(STDOUT_FILENO, buffer, strlen(buffer));

    allocator_api->allocator_free(allocator, block1);
    allocator_api->allocator_free(allocator, block2);

    char free_message[] = "Memory freed\n";
    write(STDOUT_FILENO, free_message, sizeof(free_message) - 1);

    allocator_api->allocator_destroy(allocator);
    free(allocator_api);
    munmap(addr, size);

    char exit_message[] = "Program exited successfully\n";
    write(STDOUT_FILENO, exit_message, sizeof(exit_message) - 1);

    return EXIT_SUCCESS;
}
```

buddy.c

```
#include <stdlib.h>

#include <unistd.h>
```

```
#include <math.h>

#include <string.h>

#include <sys/mman.h>

#include <unistd.h>


#ifdef _MSC_VER

#define EXPORT __declspec(dllexport)

#else

#define EXPORT

#endif


typedef struct BuddyNode

{

    int size;                // Размер блока

    int free;                // Свободен ли блок

    struct BuddyNode *left;  // Левый "двойник"

    struct BuddyNode *right; // Правый "двойник"

} BuddyNode;


typedef struct Allocator

{

    BuddyNode *root; // Корневой узел

    void *memory;    // Указатель на начало области памяти

    int totalSize;   // Общий размер памяти

    int offset;      // Текущий смещённый указатель для работы с памятью

} Allocator;
```

```
int is_power_of_two(unsigned int n)

{

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

}



BuddyNode *create_node(Allocator *allocator, int size)

{

    if (allocator->offset + sizeof(BuddyNode) > allocator->totalSize)

    {

        return NULL;

    }



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

    allocator->offset += sizeof(BuddyNode);

    node->size = size;

    node->free = 1;

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

    return node;

}



EXPORT Allocator *allocator_create(void *mem, size_t size)

{

    if (!is_power_of_two(size))

    {

        const char msg[] = "This allocator initialize a power of two\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);
    if (!allocator->root)
    {
        return NULL;
    }

    return allocator;
}

void split_node(Allocator *allocator, BuddyNode *node)
{
    int newSize = node->size / 2;

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

BuddyNode *allocate_recursive(Allocator *allocator, BuddyNode *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_recursive(allocator, node->left, size);
    if (allocated == NULL)
    {
        allocated = allocate_recursive(allocator, node->right, size);
    }

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

```
EXPORT void *allocator_alloc(Allocator *allocator, size_t size)

{

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

    {

        return NULL;

    }



    while (!is_power_of_two(size))

    {

        size++;

    }



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

}



EXPORT void allocator_free(Allocator *allocator, void *ptr)

{

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

    {

        return;

    }

    BuddyNode *node = (BuddyNode *)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;
    }
}


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);
    }
}
```

```
}
```

mkk.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;

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)
        {
            exit(EXIT_FAILURE);
        }
    }
}
```

Протокол работы программы

Тестирование:

```
$ ./example
WARNING: failed to load shared library
Allocator initialized
Memory allocated successfully
Block 1 address: 0x7fa81ca56004
Block 2 address: 0x7fa81ca55004
Memory freed
Program exited successfully

$ ./main.out buddy.so
Allocator initialized
Memory allocated successfully
Block 1 address: 0x78f8153b2120
Block 2 address: 0x78f8153b2138
```

```
Memory freed
Program exited successfully

$ ./main.out mkk.so
Allocator initialized
Memory allocated successfully
Use memory
Block 1 usage 0x7a76d10a480e
Block 2 usage 0x7a76d10a484e
Memory freed
Program exited successfully
```

Strace:

```
execve("./example", ["./example"], 0x7ffd0479d978 /* 82 vars */) = 0
brk(NULL)                               = 0x57c9575d9000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x77affbb93000
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
fstat(3, {st_mode=S_IFREG|0644, st_size=74139, ...}) = 0
mmap(NULL, 74139, PROT_READ, MAP_PRIVATE, 3, 0) = 0x77affbb80000
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"...,
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) = 0x77affb800000
mmap(0x77affb828000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|
MAP_DENYWRITE, 3, 0x28000) = 0x77affb828000
mmap(0x77affb9b0000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x1b0000) = 0x77affb9b0000
mmap(0x77affb9ff000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
MAP_DENYWRITE, 3, 0x1fe000) = 0x77affb9ff000
mmap(0x77affba05000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
MAP_ANONYMOUS, -1, 0) = 0x77affba05000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x77affbb7d000
arch_prctl(ARCH_SET_FS, 0x77affbb7d740) = 0
set_tid_address(0x77affbb7da10)         = 50134
set_robust_list(0x77affbb7da20, 24)     = 0
rseq(0x77affbb7e060, 0x20, 0, 0x53053053) = 0
mprotect(0x77affb9ff000, 16384, PROT_READ) = 0
mprotect(0x57c9556c3000, 4096, PROT_READ) = 0
mprotect(0x77affbbcb000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY})
= 0
munmap(0x77affbb80000, 74139)           = 0
write(2, "WARNING: failed to load shared l"..., 39WARNING: failed to load shared
library
) = 39
getrandom("\xf0\xb3\x8f\x2a\x79\x4a\xd4\x74", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x57c9575d9000
brk(0x57c9575fa000)                     = 0x57c9575fa000
```

```
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x77affbb92000
write(1, "Allocator initialized\n", 22Allocator initialized
) = 22
mmap(NULL, 68, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0) =
0x77affbb91000
mmap(NULL, 44, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0) =
0x77affbb90000
write(1, "Memory allocated successfully\n", 30Memory allocated successfully
) = 30
write(1, "Block 1 address: 0x77affbb91004\n", 32Block 1 address: 0x77affbb91004
) = 32
write(1, "Block 2 address: 0x77affbb90004\n", 32Block 2 address: 0x77affbb90004
) = 32
munmap(0x77affbb91000, 68)              = 0
munmap(0x77affbb90000, 44)              = 0
write(1, "Memory freed\n", 13Memory freed
)          = 13
munmap(0x77affbb92000, 4096)            = 0
write(1, "Program exited successfully\n", 28Program exited successfully
) = 28
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