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-210Б-23

Студент: Стаценко В.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 9.01.25

Москва, 2024

Постановка задачи

Вариант 5.

Реализовать два алгоритма аллокации памяти: Мак-Кьюзика-Кэрелса и алгоритм двойников.

Общий метод и алгоритм решения

Использованные системные вызовы:

- void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset) - отображает объект разделяемой памяти в адресное пространство процесса;
- int munmap(void *start, size_t length) - освобождает область памяти, которая была выделена с помощью mmap;
- int write(int fd, const void* buffer, int count) - записывает по дескриптору fd count байт из buffer;
- void *dlopen(const char *filename, int flag) - открывает динамическую библиотеку;
- void *dlsym(void *handle, const char *symbol) - извлекает адрес функции или переменной symbol из открытой библиотеки handle;
- int dlclose(void *handle) - закрывает динамическую библиотеку.

Алгоритм работы аллокатора, реализованного с помощью метода Мак-Кьюзика-Кэрелса:

1) Инициализация аллокатора

Выделение памяти для структуры аллокатора:

- Используется mmap для выделения памяти под структуру Allocator.
- Если выделение не удалось, возвращается NULL.

Выделение памяти для массива блоков:

- Выделяется память для массива блоков (Block) с помощью mmap.
- Если выделение не удалось, освобождается память, выделенная для структуры аллокатора, и возвращается NULL.

Инициализация списков свободных блоков:

- Массив free_lists инициализируется NULL для каждого списка.
- Каждый блок в массиве блоков инициализируется: устанавливается индекс блока, флаг is_free устанавливается в true (блок свободен), размер блока устанавливается в BLOCK_SIZE, указатели next и prev устанавливаются в NULL.
- Каждый блок добавляется в соответствующий список свободных блоков на основе его размера.

2) Выделение памяти

Проверка входных данных:

- Если аллокатор не инициализирован или запрашиваемый размер равен 0, возвращается NULL.

Поиск подходящего списка свободных блоков:

- Определяется индекс списка на основе запрашиваемого размера с помощью функции get_list_index.
- Если в текущем списке нет свободных блоков, поиск продолжается в списках для блоков большего размера.

Выбор блока для выделения:

- Берётся первый блок из найденного списка свободных блоков.
- Флаг is_free устанавливается в false (блок занят).

Обновление списка свободных блоков:

- Указатель free_lists[list_index] перемещается на следующий блок в списке.
- Если следующий блок существует, его указатель prev устанавливается в NULL.

Возврат выделенного блока:

- Возвращается указатель на выделенный блок.

3) Освобождение памяти

Проверка входных данных:

- Если аллокатор не инициализирован, блок не существует или блок уже свободен, функция завершается без изменений.

Помечение блока как свободного:

- Флаг is_free устанавливается в true.

Проверка соседних блоков:

- Получаются указатели на предыдущий и следующий блоки с помощью функций get_previous_block и get_next_block.
- Если соседний блок свободен, выполняется слияние: размер текущего блока увеличивается на размер соседнего блока, указатели next и prev обновляются для поддержания целостности списка, соседний блок помечается как занятый (он больше не существует как отдельный блок).

Добавление блока в список свободных:

- Определяется индекс списка на основе размера блока.
- Блок добавляется в начало соответствующего списка свободных блоков.
- Указатели next и prev обновляются для поддержания целостности списка.

4) Уничтожение аллокатора

Проверка входных данных:

- Если аллокатор не инициализирован, функция завершается без изменений.

Освобождение памяти для массива блоков:

- Если массив блоков был выделен, он освобождается с помощью munmap.

Освобождение памяти для структуры аллокатора:

- Память, выделенная для структуры Allocator, освобождается с помощью munmap.

Алгоритм работы аллокатора, реализованного с помощью метода двойников:

1) Инициализация аллокатора

Проверка размера памяти:

- Если размер памяти меньше минимального размера блока (MIN_BLOCK_SIZE), аллокатор не создаётся, и возвращается NULL.

Выделение памяти для структуры аллокатора:

- Используется mmap для выделения памяти под структуру Allocator.
- Если выделение не удалось, возвращается NULL.

Инициализация полей аллокатора:

- Поле memory указывает на переданную память.
- Поле size устанавливается в размер переданной памяти.
- Поле num_levels вычисляется как количество уровней в иерархии блоков, начиная с минимального размера блока (MIN_BLOCK_SIZE) до размера всей памяти.

Выделение памяти для списков свободных блоков:

- Используется mmap для выделения памяти под массив free_lists, который хранит указатели на списки свободных блоков для каждого уровня.
- Если выделение не удалось, освобождается память, выделенная для структуры аллокатора, и возвращается NULL.

Инициализация списков свободных блоков:

- Все списки инициализируются NULL.

- Единственный свободный блок на верхнем уровне (самый большой блок) добавляется в список свободных блоков.

2) Выделение памяти

Проверка входных данных:

- Если аллокатор не инициализирован или запрашиваемый размер больше размера всей памяти, возвращается NULL.

Определение размера блока:

- Запрашиваемый размер округляется вверх до ближайшей степени двойки с помощью функции round_up_pow2.

Определение уровня блока:

- Вычисляется уровень блока на основе его размера с помощью функции get_level.

Поиск подходящего блока:

- Начиная с текущего уровня, аллокатор ищет свободный блок в списках свободных блоков.
- Если на текущем уровне нет свободных блоков, поиск продолжается на более высоких уровнях.

Разделение блока:

- Если найденный блок больше запрашиваемого, он разделяется на два двойника.
- Один двойник добавляется в список свободных блоков на более низком уровне, а другой используется для дальнейшего разделения или выделения.

Возврат выделенного блока:

- Когда найден блок подходящего размера, он удаляется из списка свободных блоков и возвращается.

3) **Освобождение памяти**

Проверка входных данных:

- Если аллокатор не инициализирован или указатель на память равен NULL, функция завершается без изменений.

Определение смещения блока:

- Вычисляется смещение блока относительно начала памяти аллокатора.

Определение уровня блока:

- Вычисляется уровень блока на основе его размера и смещения.

Поиск двойника:

- Вычисляется адрес двойника освобождаемого блока.
- Проверяется, свободен ли двойник.

Объединение блоков:

- Если двойник свободен, блоки объединяются в один блок большего размера.
- Процесс повторяется, пока возможно объединение с двойником.

Добавление блока в список свободных:

- Если объединение невозможно, освобождённый блок добавляется в список свободных блоков на соответствующем уровне.

4) Уничтожение аллокатора

Проверка входных данных:

- Если аллокатор не инициализирован, функция завершается без изменений.

Освобождение памяти для списков свободных блоков:

- Если массив free_lists был выделен, он освобождается с помощью munmap.

Освобождение памяти для структуры аллокатора:

- Память, выделенная для структуры Allocator, освобождается с помощью munmap.

Код программы

Алгоритм Мак-Кьюзика-Кэрелса:

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

#define BLOCK_SIZE 512
#define TOTAL_BLOCKS 1000
#define NUM_LISTS 10
#define MAP_ANONYMOUS 0x20

typedef struct Block {
    uint32_t index;
    bool is_free;
    size_t size;
    struct Block* next;
    struct Block* prev;
} Block;

typedef struct Allocator {
    Block* blocks;
    Block* free_lists[NUM_LISTS];
    uint32_t total_blocks;
} Allocator;

int get_list_index(size_t size) {
    int index = 0;
    while (size > BLOCK_SIZE) {
        size /= 2;
        index++;
    }
    return index;
}

Block* get_previous_block(Allocator* allocator, Block* block) {
    if (block->index > 0) {
        return &allocator->blocks[block->index - 1];
    }
    return NULL;
}
```

```
Block* get_next_block(Allocator* allocator, Block* block) {
    if (block->index < allocator->total_blocks - 1) {
        return &allocator->blocks[block->index + 1];
    }
    return NULL;
}

void merge_blocks(Allocator* allocator, Block* block1, Block* block2) {
    block1->size += block2->size;
    block1->next = block2->next;
    if (block2->next) {
        block2->next->prev = block1;
    }
    block2->is_free = false;
}

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

    allocator->total_blocks = TOTAL_BLOCKS;

    allocator->blocks = (Block*)mmap(NULL, allocator->total_blocks * sizeof(Block), PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0)
    if (allocator->blocks == MAP_FAILED) {
        perror("mmap for blocks failed");
        munmap(allocator, sizeof(Allocator));
        return NULL;
    }

    for (int i = 0; i < NUM_LISTS; i++) {
        allocator->free_lists[i] = NULL;
    }
```

```
    for (uint32_t i = 0; i < allocator->total_blocks; i++) {
        allocator->blocks[i].index = i;
        allocator->blocks[i].is_free = true;
        allocator->blocks[i].size = BLOCK_SIZE;
        allocator->blocks[i].next = NULL;
        allocator->blocks[i].prev = NULL;

        int list_index = get_list_index(allocator->blocks[i].size);
        if (allocator->free_lists[list_index]) {
            allocator->free_lists[list_index]->prev = &allocator->blocks[i];
        }
        allocator->blocks[i].next = allocator->free_lists[list_index];
        allocator->free_lists[list_index] = &allocator->blocks[i];
    }

    return allocator;
}

Block* allocator_alloc(Allocator* allocator, size_t size) {
    if (!allocator || size == 0) {
        return NULL;
    }

    int list_index = get_list_index(size);
    while (list_index < NUM_LISTS && !allocator->free_lists[list_index]) {
        list_index++;
    }

    if (list_index >= NUM_LISTS) {
        return NULL;
    }

    Block* block = allocator->free_lists[list_index];
    block->is_free = false;

    allocator->free_lists[list_index] = block->next;
    if (allocator->free_lists[list_index]) {
        allocator->free_lists[list_index]->prev = NULL;
    }

    return block;
}
```

```c
void allocator_free(Allocator* allocator, Block* block) {
    if (!allocator || !block || block->is_free) {
        return;
    }

    block->is_free = true;

    Block* prev_block = get_previous_block(allocator, block);
    Block* next_block = get_next_block(allocator, block);

    if (prev_block && prev_block->is_free) {
        merge_blocks(allocator, prev_block, block);
        block = prev_block;
    }

    if (next_block && next_block->is_free) {
        merge_blocks(allocator, block, next_block);
    }

    int list_index = get_list_index(block->size);
    block->next = allocator->free_lists[list_index];
    block->prev = NULL;
    if (allocator->free_lists[list_index]) {
        allocator->free_lists[list_index]->prev = block;
    }
    allocator->free_lists[list_index] = block;
}

void allocator_destroy(Allocator* allocator) {
    if (allocator) {
        if (allocator->blocks) {
            munmap(allocator->blocks, allocator->total_blocks * sizeof(Block));
        }
        munmap(allocator, sizeof(Allocator));
    }
}
```

Алгоритм двойников:

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

#define MIN_BLOCK_SIZE 8
#define MAX_BLOCK_SIZE (1 << 20)
#define MAP_ANONYMOUS 0x20

typedef struct Allocator {
    void *memory;
    size_t size;
    void **free_lists;
    size_t num_levels;
} Allocator;

size_t round_up_pow2(size_t size) {
    size_t power = MIN_BLOCK_SIZE;
    while (power < size) {
        power *= 2;
    }
    return power;
}

size_t get_level(size_t block_size) {
    return (size_t)(log2(block_size) - log2(MIN_BLOCK_SIZE));
}

Allocator* allocator_create(void *const memory, const size_t size) {
    if (size < MIN_BLOCK_SIZE) {
        fprintf(stderr, "Memory size too small\n");
        return NULL;
    }

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

```
    allocator->memory = memory;
    allocator->size = size;
    allocator->num_levels = get_level(size) + 1;

    allocator->free_lists = (void **)mmap(
        NULL, allocator->num_levels * sizeof(void *), PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
    if (allocator->free_lists == MAP_FAILED) {
        perror("mmap for free_lists failed");
        munmap(allocator, sizeof(Allocator));
        return NULL;
    }

    memset(allocator->free_lists, 0, allocator->num_levels * sizeof(void *));
    allocator->free_lists[allocator->num_levels - 1] = memory;
    return allocator;
}

void* allocator_alloc(Allocator *const allocator, const size_t size) {
    if (!allocator || size > allocator->size) {
        return NULL;
    }

    size_t block_size = round_up_pow2(size);
    size_t level = get_level(block_size);

    for (size_t i = level; i < allocator->num_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 + (1 << (i + (size_t)log2(MIN_BLOCK_SIZE))));
                *(void **)buddy = allocator->free_lists[i];
                allocator->free_lists[i] = buddy;
            }

            return block;
        }
    }
```

```
    return NULL;
}

void allocator_free(Allocator *const allocator, void *const memory) {
    if (!allocator || !memory) {
        return;
    }

    size_t offset = (char *)memory - (char *)allocator->memory;
    if (offset >= allocator->size) {
        return;
    }

    size_t level = 0;
    size_t block_size = MIN_BLOCK_SIZE;
    while (block_size < allocator->size && (offset % (block_size * 2)) == 0) {
        block_size *= 2;
        level++;
    }

    void *buddy = (void *)((char *)allocator->memory + (offset ^ block_size));
    void **current = &allocator->free_lists[level];
    while (*current) {
        if (*current == buddy) {
            *current = *(void **)(*current);
            allocator_free(allocator, (offset < (char *)buddy - (char *)allocator->memory) ? memory : buddy);
            return;
        }
        current = (void **)*current;
    }

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

```c
void allocator_destroy(Allocator *const allocator) {
    if (!allocator) {
        return;
    }

    if (allocator->free_lists) {
        munmap(allocator->free_lists, allocator->num_levels * sizeof(void *));
    }
    munmap(allocator, sizeof(Allocator));
}
```

main.c:

```c
#include <dlfcn.h>
#include <math.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>
#include <bits/mman-linux.h>

#define MAP_ANONYMOUS 0x20

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)memory;
    (void)size;
    return memory;
}

void *standard_allocator_alloc(void *allocator, size_t size) {
    (void)allocator;
    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;
}
```

```c
void standard_allocator_free(void *allocator, void *memory) {
    (void)allocator;
    if (memory == NULL) return;
    uint32_t *mem = (uint32_t *)memory - 1;
    munmap(mem, *mem);
}

void standard_allocator_destroy(void *allocator) {
    (void)allocator;
}

void load_allocator(const char *library_path, Allocator *allocator) {
    if (library_path == NULL || library_path[0] == '\0') {
        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;
    }

    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_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_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) {
        char msg[] = "Error: failed to load all allocator functions\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        dlclose(library);
        allocator->allocator_create = standard_allocator_create;
        allocator->allocator_alloc = standard_allocator_alloc;
        allocator->allocator_free = standard_allocator_free;
        allocator->allocator_destroy = standard_allocator_destroy;
    }
}

size_t round_up_to_power_of_two(size_t size) {
    if (size < 8) {
        return 8;
    }
    size_t power = 1;
    while (power < size) {
        power <<= 1;
    }
    return power;
}

int main(int argc, char **argv) {
    const char *library_path = (argc > 1) ? argv[1] : NULL;
    Allocator allocator_api;
    load_allocator(library_path, &allocator_api);

    size_t size = 4096;
    size_t block_size = 16;

    void *addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED) {
        perror("mmap failed");
        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);
        return EXIT_FAILURE;
    }

    void *blocks[12];
    size_t block_sizes[12] = {16, 16, 16, 32, 48, 64, 96, 128, 256, 128, 128, 128};

    int alloc_failed = 0;
    for (int i = 0; i < 12; ++i) {
        blocks[i] = allocator_api.allocator_alloc(allocator, block_sizes[i]);
        if (blocks[i] == NULL) {
            alloc_failed = 1;
            char alloc_fail_message[] = "Memory allocation failed\n";
            write(STDERR_FILENO, alloc_fail_message, sizeof(alloc_fail_message) - 1);
            break;
        }
    }

    if (!alloc_failed) {
        char alloc_success_message[] = "Memory allocated successfully\n";
        write(STDOUT_FILENO, alloc_success_message, sizeof(alloc_success_message) - 1);
        for (int i = 0; i < 12; ++i) {
            char buffer[64];
            snprintf(buffer, sizeof(buffer), "Block %d address: %p\n", i + 1,
                     blocks[i]);
            write(STDOUT_FILENO, buffer, strlen(buffer));
        }
    }
```

```c
    for (int i = 0; i < 12; ++i) {
        if (blocks[i] != NULL) {
            allocator_api.allocator_free(allocator, blocks[i]);
        }
    }
    char free_message[] = "Memory freed\n";
    write(STDOUT_FILENO, free_message, sizeof(free_message) - 1);

    allocator_api.allocator_destroy(allocator);
    munmap(addr, size);

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

Протокол работы программы

Тестирование

```
victoria@victoria:~/laba/os/OSlabs/laba4$ cc -o mc.so -fPIC -shared mc.c -lm
victoria@victoria:~/laba/os/OSlabs/laba4$ cc -o Main -ldl main.c
victoria@victoria:~/laba/os/OSlabs/laba4$ ./Main ./mc.so
Memory allocated successfully
Block 1 address: 0x7f0468119ce0
Block 2 address: 0x7f0468119cc0
Block 3 address: 0x7f0468119ca0
Block 4 address: 0x7f0468119c80
Block 5 address: 0x7f0468119c60
Block 6 address: 0x7f0468119c40
Block 7 address: 0x7f0468119c20
Block 8 address: 0x7f0468119c00
Block 9 address: 0x7f0468119be0
Block 10 address: 0x7f0468119bc0
Block 11 address: 0x7f0468119ba0
Block 12 address: 0x7f0468119b80
Memory freed
Program exited successfully
```

```
victoria@victoria:~/laba/os/OSlabs/laba4$ cc -o buddy.so -fPIC -shared buddy.c -lm
victoria@victoria:~/laba/os/OSlabs/laba4$ cc -o Main -ldl main.c
victoria@victoria:~/laba/os/OSlabs/laba4$ ./Main ./buddy.so
Memory allocated successfully
Block 1 address: 0x7f4a38ff3000
Block 2 address: 0x7f4a38ff3010
Block 3 address: 0x7f4a38ff3020
Block 4 address: 0x7f4a38ff3040
Block 5 address: 0x7f4a38ff3080
Block 6 address: 0x7f4a38ff30c0
Block 7 address: 0x7f4a38ff3100
Block 8 address: 0x7f4a38ff3180
Block 9 address: 0x7f4a38ff3200
Block 10 address: 0x7f4a38ff3300
Block 11 address: 0x7f4a38ff3380
Block 12 address: 0x7f4a38ff3400
Memory freed
Program exited successfully
```

Сравнение алгоритмов

1) Фактор использования памяти

Алгоритм Мак-Кьюзика-Кэрелса:

- Ниже из-за фиксированного размера блоков.

Алгоритм двойников:

- Выше, так как блоки разделяются на двойники только до нужного размера.

Итог: Алгоритм двойников обеспечивает более высокий фактор использования памяти благодаря гибкости в разделении блоков.

2) Скорость выделения блоков

Алгоритм Мак-Кьюзика-Кэрелса:

- Медленнее, так как требует поиска подходящего блока в списке свободных блоков
- Сложность: O(n), где n — количество блоков в списке.

Алгоритм двойников:

- Быстрее, так как блоки строго организованы по размерам и адресам.
- Сложность: O(log n), где n — количество уровней в иерархии блоков.

Итог: Алгоритм двойников обеспечивает более высокую скорость выделения блоков благодаря строгой организации блоков и рекурсивному разделению.

3) Скорость освобождения блоков

Алгоритм Мак-Кьюзика-Кэрелса:

- Медленнее, так как требует проверки соседних блоков для слияния.
- Сложность: O(n), где n — количество соседних блоков.

Алгоритм двойников:

- Быстрее, так как требует только поиска двойника и его объединения.
- Сложность: O(1) для поиска двойника и O(log n) для рекурсивного объединения.

Итог: Алгоритм двойников обеспечивает более высокую скорость освобождения блоков благодаря строгой организации блоков и быстрому поиску двойника.

4) Простота использования аллокатора

Алгоритм Мак-Кьюзика-Кэрелса:

- Проще в реализации и понимании, так как использует базовые структуры данных (списки) и не требует сложной логики для разделения и объединения блоков.
- Менее гибкий, так как все блоки имеют фиксированный размер, что ограничивает его применение.

Алгоритм двойников:

- Сложнее в реализации и понимании, так как требует рекурсивного разделения и объединения блоков, а также строгой организации блоков по адресам.
- Более гибкий, так как поддерживает блоки разного размера и минимизирует фрагментацию.

Итог: Алгоритм Мак-Кьюзика-Кэрелса проще в использовании, но менее гибкий. Алгоритм двойников сложнее в реализации, но обеспечивает большую гибкость и эффективность.

Strace

```
victoria@victoria:~/laba/os/OSlabs/laba4$ strace ./Main ./buddy.so
execve("./Main", ["./Main", "./buddy.so"], 0x7fff735c5d78 /* 35 vars */) = 0
brk(NULL)                               = 0x559d0800b000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd847f0710) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f14a3d21000
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
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=17915, ...}, AT_EMPTY_PATH) = 0
```

```
        mmap(NULL, 17915, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f14a3d1c000
        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) =
0x7f14a3af3000
        mprotect(0x7f14a3b1b000, 2023424, PROT_NONE) = 0
        mmap(0x7f14a3b1b000, 1658880, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f14a3b1b000
        mmap(0x7f14a3cb0000, 360448, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7f14a3cb0000
        mmap(0x7f14a3d09000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x7f14a3d09000
        mmap(0x7f14a3d0f000, 52816, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f14a3d0f000
        close(3)                        = 0
        mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7f14a3af0000
        arch_prctl(ARCH_SET_FS, 0x7f14a3af0740) = 0
        set_tid_address(0x7f14a3af0a10)         = 686566
        set_robust_list(0x7f14a3af0a20, 24)     = 0
        rseq(0x7f14a3af10e0, 0x20, 0, 0x53053053) = 0
        mprotect(0x7f14a3d09000, 16384, PROT_READ) = 0
        mprotect(0x559d07192000, 4096, PROT_READ) = 0
        mprotect(0x7f14a3d5b000, 8192, PROT_READ) = 0
        prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY})
= 0
        munmap(0x7f14a3d1c000, 17915)           = 0
        getrandom("\x75\x5e\xfd\x44\xef\x8b\xc9\xbe", 8, GRND_NONBLOCK) = 8
        brk(NULL)                               = 0x559d0800b000
        brk(0x559d0802c000)                     = 0x559d0802c000
        openat(AT_FDCWD, "./buddy.so", O_RDONLY|O_CLOEXEC) = 3
        read(3, "\177ELF\2\1\1\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|0755, st_size=16136, ...}, AT_EMPTY_PATH) = 0
        getcwd("/home/victoria/laba/os/OSlabs/laba4", 128) = 36
        mmap(NULL, 16488, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f14a3d1c000
        mmap(0x7f14a3d1d000, 4096, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7f14a3d1d000
        mmap(0x7f14a3d1e000, 4096, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f14a3d1e000
        mmap(0x7f14a3d1f000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f14a3d1f000
        close(3)                        = 0
        openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
        newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=17915, ...}, AT_EMPTY_PATH) = 0
        mmap(NULL, 17915, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f14a3aeb000
        close(3)                        = 0
        openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libm.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\0\0\0\0\0\0\0"..., 832) = 832
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=940560, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 942344, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f14a3a04000
mmap(0x7f14a3a12000, 507904, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xe000) = 0x7f14a3a12000
mmap(0x7f14a3a8e000, 372736, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x8a000) = 0x7f14a3a8e000
mmap(0x7f14a3ae9000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xe4000) = 0x7f14a3ae9000
close(3)                                = 0
mprotect(0x7f14a3ae9000, 4096, PROT_READ) = 0
mprotect(0x7f14a3d1f000, 4096, PROT_READ) = 0
munmap(0x7f14a3aeb000, 17915)           = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x7f14a3d5a000
mmap(NULL, 32, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7f14a3aef000
mmap(NULL, 80, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x7f14a3aee000
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x7), ...},
AT_EMPTY_PATH) = 0
write(1, "Memory allocated successfully\n", 30Memory allocated successfully
) = 30
write(1, "Block 1 address: 0x7f14a3d5a000\n", 32Block 1 address: 0x7f14a3d5a000
) = 32
write(1, "Block 2 address: 0x7f14a3d5a010\n", 32Block 2 address: 0x7f14a3d5a010
) = 32
write(1, "Block 3 address: 0x7f14a3d5a020\n", 32Block 3 address: 0x7f14a3d5a020
) = 32
write(1, "Block 4 address: 0x7f14a3d5a040\n", 32Block 4 address: 0x7f14a3d5a040
) = 32
write(1, "Block 5 address: 0x7f14a3d5a080\n", 32Block 5 address: 0x7f14a3d5a080
) = 32
write(1, "Block 6 address: 0x7f14a3d5a0c0\n", 32Block 6 address: 0x7f14a3d5a0c0
) = 32
write(1, "Block 7 address: 0x7f14a3d5a100\n", 32Block 7 address: 0x7f14a3d5a100
) = 32
write(1, "Block 8 address: 0x7f14a3d5a180\n", 32Block 8 address: 0x7f14a3d5a180
) = 32
write(1, "Block 9 address: 0x7f14a3d5a200\n", 32Block 9 address: 0x7f14a3d5a200
) = 32
write(1, "Block 10 address: 0x7f14a3d5a300"..., 33Block 10 address: 0x7f14a3d5a300
) = 33
write(1, "Block 11 address: 0x7f14a3d5a380"..., 33Block 11 address: 0x7f14a3d5a380
) = 33
write(1, "Block 12 address: 0x7f14a3d5a400"..., 33Block 12 address: 0x7f14a3d5a400
) = 33
write(1, "Memory freed\n", 13Memory freed
)       = 13
munmap(0x7f14a3aee000, 80)              = 0
munmap(0x7f14a3aef000, 32)              = 0
munmap(0x7f14a3d5a000, 4096)            = 0
write(1, "Program exited successfully\n", 28Program exited successfully
) = 28
```

```
exit_group(0)                   = ?
+++ exited with 0 +++
```

Вывод

В ходе выполнения лабораторной работы я освоила принципы работы двух алгоритмов аллокации памяти - алгоритма Мак-Кьюзика-Кэрелса и алгоритма двойников, научилась реализовывать их на языке C, а также анализировать их эффективность по таким параметрам, как фактор использования памяти, скорость выделения и освобождения блоков, и простота использования.