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: Болдинова В.В.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 01.01.25

Москва, 2024

Постановка задачи

Вариант 2.

Реализовать два алгоритма аллокации памяти: списки свободных блоков (первое подходящее) и алгоритм Мак-Кьюзи Кэрелса.

Общий метод и алгоритм решения

Использованные системные вызовы:

Алгоритмы:

Алгоритм «Списки свободных блоков» (Free Block Allocator)

Этот алгоритм представляет собой простую и гибкую стратегию управления памятью, основанную на поддержке связного списка свободных блоков. Он оптимизирован для минимизации фрагментации и поддержания высокой эффективности выделения и освобождения памяти.

Принципы работы

1. **Инициализация аллокатора (allocator_create)**:
    - Вся выделенная память разбивается на один или несколько свободных блоков.
    - Эти блоки объединяются в связный список.
2. **Выделение памяти (allocator_alloc)**:
    - Аллокатор ищет первый подходящий блок в списке свободных блоков.
    - Если блок больше запрошенного размера, он делится на две части:
        - Первая часть передаётся пользователю.
        - Оставшаяся часть остаётся в списке свободных блоков.
3. **Освобождение памяти (allocator_free)**:
    - Освобождённый блок добавляется обратно в список свободных.
    - Аллокатор проверяет, можно ли объединить освободившийся блок с соседними, чтобы уменьшить фрагментацию.
4. **Объединение блоков**:
    - Если два свободных блока расположены рядом, они объединяются в один большой блок, чтобы уменьшить фрагментацию.

Организация блоков памяти

Каждый блок памяти содержит служебную информацию и полезную память:

- **Служебная часть блока** (заголовок):
    - Размер блока.

    - Указатель на следующий свободный блок.
    - Флаг, указывающий, свободен ли блок.
- **Полезная часть блока**:
    - Место, доступное для использования программой.

Поиск блока

1. **Метод первого подходящего (First Fit)**:
    - Аллокатор начинает с начала списка свободных блоков.
    - Проверяет каждый блок, пока не найдёт первый, который достаточно велик для запрошенного размера.
2. **Уточнение блока**:
    - Если найденный блок больше запрошенного размера, он делится на две части:
        - Один блок используется.
        - Оставшаяся память остаётся в списке свободных.

Алгоритм работы функций

1. allocator_create

- Выделяет всю память в один свободный блок.
- Инициализирует заголовок блока и указывает, что он свободен.

2. allocator_alloc

- Проходит по списку свободных блоков.
- Ищет первый блок, который больше или равен запрошенному размеру.
- Делит блок на две части, если он больше запрошенного размера.

3. allocator_free

- Освобождает блок, возвращая его в список свободных.
- Проверяет соседние блоки для объединения.

Преимущества и недостатки

Преимущества:

1. Простота реализации.
2. Эффективно использует память, уменьшая внутреннюю фрагментацию.
3. Подходит для задач с переменными размерами блоков.

Недостатки:

1. Список свободных блоков может стать длинным, что замедляет поиск.

2. Подвержен внешней фрагментации (несколько небольших блоков, недостаточных для новых запросов).

Алгоритм «Мак-Кьюзи Кэрелса» (McCusIcarel's Algorithm)

Алгоритм Мак-Кьюзи Кэрелса предназначен для управления памятью, организованной в виде страниц. Он основан на разделении памяти на блоки, размеры которых являются степенями двойки. Этот подход позволяет эффективно выделять и освобождать память, сводя к минимуму фрагментацию.

Принципы работы

1. **Инициализация памяти (allocator_create)**:
    - Вся память делится на страницы фиксированного размера (обычно 4 KB).
    - Каждая страница может быть разбита на блоки памяти, размер которых равен степени двойки (16, 32, 64 байта и т. д.).
2. **Выделение памяти (allocator_alloc)**:
    - Алгоритм выбирает минимальный блок, размер которого больше или равен запрашиваемому.
    - Если на странице нет свободных блоков подходящего размера, создаётся новая страница с блоками нужного размера.
3. **Освобождение памяти (allocator_free)**:
    - Освобождённый блок возвращается в список свободных блоков соответствующего размера на своей странице.

Организация блоков памяти

1. Структура блока

Каждый блок включает:

- Размер блока (block_size): хранится в заголовке блока.
- Указатель на следующий свободный блок (next_free): используется для создания списка свободных блоков на странице.
- Флаг «свободен/занят» (is_free): указывает, доступен ли блок для использования.

2. Структура страницы

Каждая страница памяти включает:

- Размер страницы (page_size): общий размер страницы.
- Список свободных блоков (free_blocks): указатель на первый свободный блок в списке.
- Ссылка на следующую страницу (next_free): для управления списком страниц.

3. Структура аллокатора

Аллокатор управляет:

- Общей памятью (memory): вся доступная память.
- Размер памяти (size): общий размер памяти, доступной для распределителя.
- Списком страниц (free_pages): указатель на первую свободную страницу.

Как происходит поиск блоков

1. Поиск подходящей страницы

- Алгоритм начинает с первой страницы из списка free_pages.
- Проверяет каждую страницу, пока не найдёт страницу с подходящими свободными блоками.

2. Выделение блока

- Если свободный блок найден:
    - Он удаляется из списка свободных блоков страницы.
    - Помечается как занятый.
- Если свободного блока нужного размера нет:
    - Алгоритм выделяет новую страницу и инициализирует её блоками нужного размера.

3. Освобождение блока

- Освобождённый блок добавляется обратно в список свободных блоков страницы, из которой он был взят.

Алгоритм работы функций

1. allocator_create

- Делит память на страницы фиксированного размера (PAGE_SIZE).
- Каждая страница добавляется в список free_pages.

2. allocator_alloc

- Определяет минимальный размер блока (степень двойки), который может удовлетворить запрос.
- Ищет подходящую страницу.
- Если свободного блока нет, инициализирует новую страницу.

3. allocator_free

- Помечает освобождённый блок как свободный.
- Добавляет блок в список свободных блоков на странице.

Преимущества и недостатки

Преимущества:

1. Эффективное управление блоками памяти фиксированных размеров.
2. Минимальная внутренняя фрагментация благодаря степеням двойки.
3. Быстрое выделение и освобождение памяти.

Недостатки:

1. Потенциальная внешняя фрагментация, если требуется много блоков разных размеров.
2. Сложность обработки больших запросов на память.

Анализ эффективности двух алгоритмов

Алгоритм Мак-Кьюзика-Кэрелса (McCusIcarel's Algorithm)

1. **Фактор использования памяти**:
    - **Преимущества**:
        - Меньшая внутренняя фрагментация, поскольку память делится на блоки фиксированного размера (степени двойки), что подходит для многих типичных запросов.
        - Блоки небольшого размера (например, 16, 32, 64 байта) эффективно заполняются, минимизируя избыточное использование.
    - **Недостатки**:
        - **Внешняя фрагментация** возникает, если многие страницы заполнены блоками определённого размера, а запросы на память других размеров не могут быть удовлетворены.
        - Запрос памяти, превышающий размер страницы, может быть отклонён, даже если в других страницах достаточно памяти.

Итог: коэффициент использования умеренно высокий, но может страдать от внешней фрагментации.

2. **Скорость выделения блоков**:
    - **Преимущества**:
        - Быстрое выделение благодаря связному списку свободных блоков внутри каждой страницы.
        - Разбиение блоков степеней двойки позволяет быстро находить подходящий размер.
    - **Недостатки**:
        - Если подходящего свободного блока нет, требуется инициализация новой страницы, что увеличивает накладные расходы.

Итог: высокая скорость выделения, особенно для запросов, соответствующих размеру существующих блоков.

3. **Скорость освобождения блоков**:
    - **Преимущества**:
        - Освобождение блока быстрое: блок возвращается в список свободных на своей странице.
    - **Недостатки**:
        - Для восстановления оптимальной структуры (например, при объединении страниц) могут потребоваться дополнительные операции.

Итог: освобождение блоков эффективно, особенно если не нужно объединять страницы.

4. **Простота использования**:
    - **Преимущества**:
        - Пользователь работает с интерфейсом, аналогичным стандартным malloc и free, что упрощает использование.
    - **Недостатки**:
        - Сложная внутренняя реализация, что затрудняет модификацию и отладку.

Итог: простота использования для разработчика высокая, сложность реализации компенсируется универсальностью.

Алгоритм "Списки свободных блоков"

1. **Фактор использования памяти**:
    - **Преимущества**:
        - Возможность выделять блоки точного размера, минимизируя внутреннюю фрагментацию.
        - Освобождённые блоки можно объединять, чтобы уменьшить внешнюю фрагментацию.
    - **Недостатки**:
        - При большом количестве мелких запросов или сложной структуре списка может возникать **внешняя фрагментация** (разрыв между свободными блоками).

Итог: высокий коэффициент использования, но может снизиться из-за внешней фрагментации при большом количестве запросов.

2. **Скорость выделения блоков**:
    - **Преимущества**:

        - Быстрое выделение при использовании стратегии "первый подходящий блок".
    - **Недостатки**:
        - Если список свободных блоков становится длинным, поиск подходящего блока может замедляться.
        - Разделение блоков на части увеличивает накладные расходы.

Итог: Скорость выделения средняя, особенно при длинных списках свободных блоков.

3. **Скорость освобождения блоков**:
    - **Преимущества**:
        - Простое добавление освобождённого блока в список свободных.
        - Возможность объединения соседних блоков, что снижает фрагментацию.
    - **Недостатки**:
        - Для объединения блоков требуется проверка соседей, что увеличивает накладные расходы.

Итог: Средняя скорость освобождения, особенно при большом количестве соседних блоков.

4. **Простота использования**:
    - **Преимущества**:
        - Простая структура списка блоков делает алгоритм интуитивно понятным.
    - **Недостатки**:
        - Требуется оптимизация для эффективной работы с большим количеством запросов.

Итог: Высокая простота как в использовании, так и в реализации.

Вывод: Выбор подходящего алгоритма зависит от характера нагрузки: если требуется высокая скорость, лучше использовать алгоритм Мак-Кьюзика-Кэрелса, а для гибкости — списки свободных блоков.

Код программы

(алгоритм Мак-Кьюзи Кэрелса) mccusIcarels-algorithm.c:

```
        #include "mccusIcarels-algorithm.h"

Allocator *allocator_create(void *const memory, const size_t size) {
    if (memory == NULL || size < PAGE_SIZE) {
        return NULL;
    }

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

```
    allocator->memory = (uint8_t *) memory + sizeof(Allocator);
    allocator->size = size - sizeof(Allocator);
    allocator->free_pages = (Page *) allocator->memory;

    size_t num_pages = allocator->size / PAGE_SIZE;
    for (size_t i = 0; i < num_pages; i++) {
        Page *page = (Page *) ((uint8_t *) allocator->memory + i *
PAGE_SIZE);
        page->page_size = PAGE_SIZE;
        page->next_free = (i == num_pages - 1) ? NULL : (Page *) ((uint8_t *)
allocator->memory + (i + 1) * PAGE_SIZE);
        page->free_blocks = NULL;
    }

    return allocator;
}


void allocator_destroy(Allocator *const allocator) {
    if (allocator == NULL) {
        return;
    }

    munmap(allocator, allocator->size + sizeof(Allocator));
}

void *allocator_alloc(Allocator *const allocator, const size_t size) {
    if (allocator == NULL || size == 0 || size > allocator->size) {
        return NULL;
    }

    size_t block_size = 1;
    while (block_size < size + sizeof(Block) && block_size < PAGE_SIZE) {
        block_size *= 2;
    }

    Page *page = allocator->free_pages;
    while (page != NULL) {
        Block *block = page->free_blocks;
        Block *prev = NULL;

        while (block != NULL) {
            if (block->block_size >= block_size && block->is_free) {
                if (prev == NULL) {
                    page->free_blocks = block->next_free;
                } else {
                    prev->next_free = block->next_free;
                }
                block->is_free = false;
                return (void *) ((uint8_t *) block + sizeof(Block));
            }
            prev = block;
            block = block->next_free;
        }
        page = page->next_free;
    }

    page = allocator->free_pages;
    if (page == NULL) {
        return NULL;
    }
    allocator->free_pages = page->next_free;

    size_t num_blocks = PAGE_SIZE / block_size;
```

```
    for (size_t i = 0; i < num_blocks; i++) {
        Block *block = (Block *) ((uint8_t *) page + i * block_size);
        block->block_size = block_size;
        block->next_free = page->free_blocks;
        block->is_free = true;
        page->free_blocks = block;
    }

    Block *block = page->free_blocks;
    page->free_blocks = block->next_free;
    block->is_free = false;
    return (void *) ((uint8_t *) block + sizeof(Block));
}

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

    Block *block = (Block *) ((uint8_t *) memory - sizeof(Block));

    Page *page = allocator->free_pages;
    while (page != NULL) {
        if ((uint8_t *) block >= (uint8_t *) page && (uint8_t *) block <
(uint8_t *) page + PAGE_SIZE) {
            block->next_free = page->free_blocks;
            block->is_free = true;
            page->free_blocks = block;
            return;
        }
        page = page->next_free;
    }
}
```

(списки свободных блоков) free-block-allocator.c:

```
#include "free-block-allocator.h"


Allocator *allocator_create(void *const memory, const size_t size) {
    if (memory == NULL) {
        return NULL;
    }


    Allocator* allocator = (Allocator*)memory;
    allocator->memory = (void *) ((char *) memory + sizeof(Allocator));
    allocator->size = size - sizeof(Allocator);
    allocator->free_list = (Block*)allocator->memory;
    allocator->free_list->size = allocator->size - sizeof(Block);
    allocator->free_list->next = NULL;
    allocator->free_list->is_free = 1;

    return allocator;
}

void *allocator_alloc(Allocator *allocator, size_t size) {
    if(allocator == NULL){
        return NULL;
    }
    if(size > allocator->size){
        return NULL;
    }
```

```
    Block *curr = allocator->free_list;
    while (curr != NULL) {
        if (curr->is_free && curr->size >= size) {
            if (curr->size >= size + sizeof(Block) + 1) {
                Block *new_block = (Block *) ((char *) curr + sizeof(Block) +
size);
                new_block->size = curr->size - size - sizeof(Block);
                new_block->next = curr->next;
                new_block->is_free = 1;

                curr->size = size;
                curr->next = new_block;
            }

            curr->is_free = 0;
            return (void *) (curr + 1);
        }
        curr = curr->next;
    }

    return NULL;
}

void allocator_free(Allocator *allocator, void *ptr) {
    if (ptr == NULL) return;

    Block *block = (Block *) ptr - 1;
    block->is_free = 1;


    Block *curr = allocator->free_list;
    while (curr != NULL && curr->next != NULL) {
        if (curr->is_free && curr->next->is_free) {
            curr->size += sizeof(Block) + curr->next->size;
            curr->next = curr->next->next;
        } else {
            curr = curr->next;
        }
    }
}

void allocator_destroy(Allocator* allocator) {
    if (munmap(allocator, allocator->size + sizeof(Allocator)) == -1) {
        perror("munmap failed");
    }
}
```

3) main.c

```
#include <stdio.h>
#include <dlfcn.h>
#include <sys/mman.h>

#include "errors.h"

#define MEMORY_SIZE 1024 * 1024

typedef struct Allocator Allocator;

typedef Allocator *create_allocator_func(void *memory, size_t size);

typedef void *allocator_alloc_func(Allocator *const allocator, const size_t
size);
```

```
typedef void allocator_free_func(Allocator *const allocator, void *const
memory);

typedef void allocator_destroy_func(Allocator *const allocator);

static create_allocator_func *create_allocator;
static allocator_alloc_func *allocator_alloc;
static allocator_free_func *allocator_free;
static allocator_destroy_func *allocator_destroy;

int print_error(error_msg error) {
    char buffer[100];
    if (error.type) {
        snprintf(buffer, 100, "Error - %s: %s\n", error.func, error.msg);
        write(STDERR_FILENO, buffer, strlen(buffer));
        return error.type;
    }
    return 0;
}

error_msg init_library(void *library) {
    create_allocator = dlsym(library, "allocator_create");
    if (create_allocator == NULL) {
        dlclose(library);
        return (error_msg) {INCORRECT_OPTIONS_ERROR, "main", "failed to find
create function"};
    }

    allocator_alloc = dlsym(library, "allocator_alloc");
    if (allocator_alloc == NULL) {
        dlclose(library);
        return (error_msg) {INCORRECT_OPTIONS_ERROR, "main", "failed to find
alloc function"};
    }

    allocator_free = dlsym(library, "allocator_free");
    if (allocator_free == NULL) {
        dlclose(library);
        return (error_msg) {INCORRECT_OPTIONS_ERROR, "main", "failed to find
free function"};
    }

    allocator_destroy = dlsym(library, "allocator_destroy");
    if (allocator_destroy == NULL) {
        dlclose(library);
        return (error_msg) {INCORRECT_OPTIONS_ERROR, "main", "failed to find
destroy function"};
    }
    return (error_msg) {SUCCESS, "", ""};
}


int main(int argc, char **argv) {
    void *library = NULL;

    if (argc == 2) {
        library = dlopen(argv[1], RTLD_LOCAL | RTLD_NOW);
    }
    if (argc != 2 || library == NULL) {
        library = dlopen("./libfree-block-allocator.so", RTLD_GLOBAL |
RTLD_LAZY);
    }
    if (library == NULL) {
```

```
        return print_error((error_msg) {INCORRECT_OPTIONS_ERROR, "main",
"incorrect count args"});
    }
    void *memory = mmap(
            NULL, MEMORY_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0
    );

    if (memory == MAP_FAILED) {
        return print_error((error_msg) {MEMORY_ALLOCATED_ERROR, "main",
"map"});
    }

    error_msg errorMsg = init_library(library);
    if(errorMsg.type){
        dlclose(library);
        return print_error(errorMsg);
    }

    Allocator * allocator = create_allocator(memory, MEMORY_SIZE);
    if(allocator == NULL){
        return print_error((error_msg) {MEMORY_ALLOCATED_ERROR, "main",
"allocator didn't create"});
    }

    // Тест 1: Выделение и освобождение памяти
    printf("Test 1: Allocating and freeing memory...\n");
    int *a = allocator_alloc(allocator, sizeof(int) * 10);
    if (a == NULL) {
        return print_error((error_msg) {MEMORY_ALLOCATED_ERROR, "main",
"memory allocated"});
    }
    a[0] = 13;
    a[9] = 19;
    printf("a[0] = %d, a[9] = %d\n", a[0], a[9]);
    allocator_free(allocator, a);
    printf("Test 1 passed.\n\n");

    // Тест 2: Выделение памяти большего размера
    printf("Test 2: Allocating larger memory block...\n");
    int *b = allocator_alloc(allocator, sizeof(int) * 1000);
    if (b == NULL) {
        return print_error((error_msg) {MEMORY_ALLOCATED_ERROR, "main",
"memory allocated"});
    }
    b[0] = 42;
    b[999] = 24;
    printf("b[0] = %d, b[999] = %d\n", b[0], b[999]);
    allocator_free(allocator, b);
    printf("Test 2 passed.\n\n");

    // Тест 3: Повторное выделение памяти после освобождения
    printf("Test 3: Reallocating memory after freeing...\n");
    int *c = allocator_alloc(allocator, sizeof(int) * 5);
    if (c == NULL) {
        return print_error((error_msg) {MEMORY_ALLOCATED_ERROR, "main",
"memory allocated"});
    }
    c[0] = 7;
    c[4] = 14;
    printf("c[0] = %d, c[4] = %d\n", c[0], c[4]);
    allocator_free(allocator, c);
    printf("Test 3 passed.\n\n");
```

```
    // Тест 4: Попытка выделения слишком большого блока памяти
    printf("Test 4: Attempting to allocate too much memory...\n");
    int *d = allocator_alloc(allocator, MEMORY_SIZE + 1); // Попытка выделить
больше, чем доступно
    if (d != NULL) {
        return print_error((error_msg) {MEMORY_ALLOCATED_ERROR, "main",
"memory allocated unexpectedly"});
    }
    printf("Test 4 passed (expected failure).\n\n");

    // Тест 5: Выделение нескольких блоков памяти
    printf("Test 5: Allocating multiple memory blocks...\n");
    int *e = allocator_alloc(allocator, sizeof(int) * 10);
    int *f = allocator_alloc(allocator, sizeof(int) * 20);
    if (e == NULL || f == NULL) {
        return print_error((error_msg) {MEMORY_ALLOCATED_ERROR, "main",
"memory allocated"});
    }
    e[0] = 1;
    f[0] = 2;
    printf("e[0] = %d, f[0] = %d\n", e[0], f[0]);
    allocator_free(allocator, e);
    allocator_free(allocator, f);
    printf("Test 5 passed.\n\n");

    allocator_destroy(allocator);

    dlclose(library);
    return 0;
}
```

Протокол работы программы

Тестирование:

```
root@LAPTOP-6G05B5VT:~# ./main ./libfree_block_allocator.so
-bash: ./main: No such file or directory
root@LAPTOP-6G05B5VT:~# /mnt/d/si/OSI/Lab4/L4/main /mnt/d/si/OSI/Lab4/L4/libf
ree_block_allocator.so
Test 1: Allocating and freeing memory...
a[0] = 13, a[9] = 19
Test 1 passed.

Test 2: Allocating larger memory block...
b[0] = 42, b[999] = 24
Test 2 passed.

Test 3: Reallocating memory after freeing...
c[0] = 7, c[4] = 14
Test 3 passed.

Test 4: Attempting to allocate too much memory...
Test 4 passed (expected failure).

Test 5: Allocating multiple memory blocks...
e[0] = 1, f[0] = 2
Test 5 passed.
```

```
root@LAPTOP-6G05B5VT:~# /mnt/d/si/OSI/Lab4/L4/main /mnt/d/si/OSI/Lab4/L4/mccu
sIcarels-algorithm.so
Test 1: Allocating and freeing memory...
a[0] = 13, a[9] = 19
Test 1 passed.

Test 2: Allocating larger memory block...
b[0] = 42, b[999] = 24
Test 2 passed.

Test 3: Reallocating memory after freeing...
c[0] = 7, c[4] = 14
Test 3 passed.

Test 4: Attempting to allocate too much memory...
Test 4 passed (expected failure).

Test 5: Allocating multiple memory blocks...
e[0] = 1, f[0] = 2
Test 5 passed.
```

Strace:

execve("/mnt/d/si/OSI/Lab4/L4/main", ["/mnt/d/si/OSI/Lab4/L4/main", "/mnt/d/si/OSI/Lab4/L4/mccusIcare"...], 0x7ffcbeeb8018 /* 26 vars */) = 0
brk(NULL) = 0x5625902d4000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdd29597000
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=20115, ...}) = 0
mmap(NULL, 20115, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fdd29592000
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) = 0x7fdd29380000
mmap(0x7fdd293a8000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7fdd293a8000
mmap(0x7fdd29530000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x7fdd29530000
mmap(0x7fdd2957f000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x7fdd2957f000
mmap(0x7fdd29585000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fdd29585000
close(3) = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fdd2937d000

```
arch_prctl(ARCH_SET_FS, 0x7fdd2937d740) = 0
set_tid_address(0x7fdd2937da10)         = 93374
set_robust_list(0x7fdd2937da20, 24)     = 0
rseq(0x7fdd2937e060, 0x20, 0, 0x53053053) = 0
mprotect(0x7fdd2957f000, 16384, PROT_READ) = 0
mprotect(0x562579c73000, 4096, PROT_READ) = 0
mprotect(0x7fdd295cf000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY})
= 0
munmap(0x7fdd29592000, 20115)           = 0
getrandom("\x62\x88\x7d\xc2\xdd\x94\x85\x56", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x5625902d4000
brk(0x5625902f5000)                     = 0x5625902f5000
openat(AT_FDCWD, "/mnt/d/si/OSI/Lab4/L4/mccusIcarels-algorithm.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=15624, ...}) = 0
mmap(NULL, 16408, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7fdd29592000
mmap(0x7fdd29593000, 4096, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7fdd29593000
mmap(0x7fdd29594000, 4096, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7fdd29594000
mmap(0x7fdd29595000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7fdd29595000
close(3)                                = 0
mprotect(0x7fdd29595000, 4096, PROT_READ) = 0
mmap(NULL, 1048576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x7fdd2927d000
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0x88, 0x2), ...}) = 0
write(1, "Test 1: Allocating and freeing m"..., 41) = 41
write(1, "a[0] = 13, a[9] = 19\n", 21)  = 21
write(1, "Test 1 passed.\n", 15)        = 15
write(1, "\n", 1)                       = 1
write(1, "Test 2: Allocating larger memory"..., 42) = 42
write(1, "b[0] = 42, b[999] = 24\n", 23) = 23
write(1, "Test 2 passed.\n", 15)        = 15
write(1, "\n", 1)                       = 1
write(1, "Test 3: Reallocating memory afte"..., 45) = 45
write(1, "c[0] = 7, c[4] = 14\n", 20)   = 20
write(1, "Test 3 passed.\n", 15)        = 15
write(1, "\n", 1)                       = 1
write(1, "Test 4: Attempting to allocate t"..., 50) = 50
write(1, "Test 4 passed (expected failure)"..., 34) = 34
write(1, "\n", 1)                       = 1
write(1, "Test 5: Allocating multiple memo"..., 45) = 45
write(1, "e[0] = 1, f[0] = 2\n", 19)    = 19
write(1, "Test 5 passed.\n", 15)        = 15
```

write(1, "\n", 1)   = 1
munmap(0x7fdd2927d000, 1048576)   = 0
munmap(0x7fdd29592000, 16408)   = 0
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 списки свободных блоков, научилась реализовывать их на языке С, а также анализировать их эффективность по таким параметрам, как фактор использования памяти, скорость выделения и освобождения блоков, и простота использования.