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.

1. **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.

1. **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.

1. **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.

1. **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.

1. **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.

1. **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:

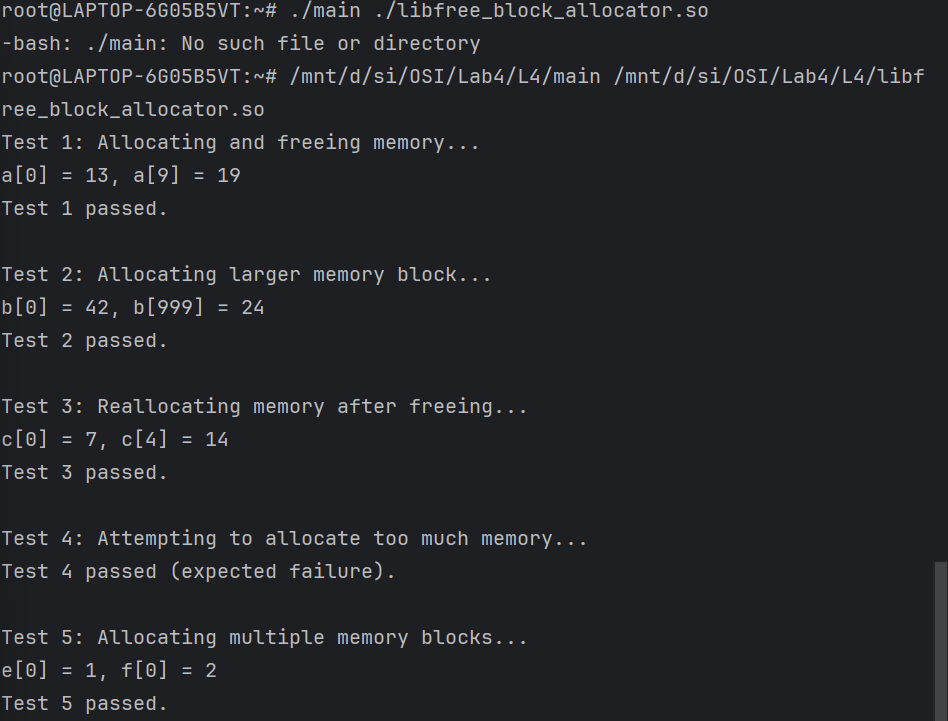
#include "free-block-allocator.h"  
  
  
Allocator \*allocator\_create(void \*const memory, const size\_t size) {  
 if (memory == NULL) {  
 return NULL;  
 }  
  
  
 Allocator\* allocator = (Allocator\*)memory;  
 allocator->memory = (void \*) ((char \*) memory + sizeof(Allocator));  
 allocator->size = 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");  
 }  
}

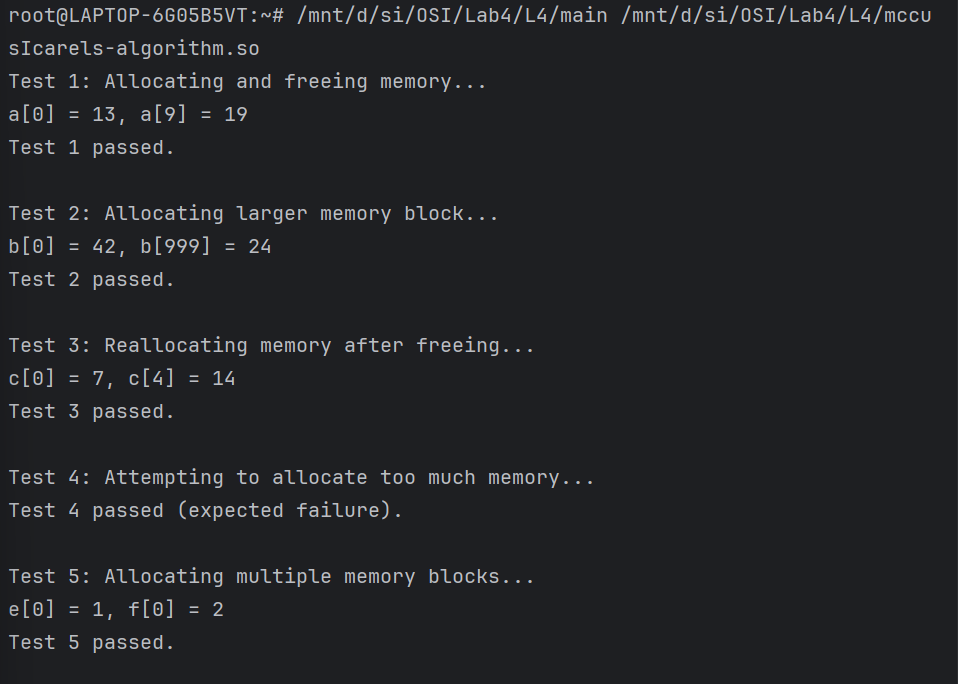
3) main.c

#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:**

****

****

**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.