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-211Б-23

Студент: Бугренков В.П.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 22.12.24

Москва, 2024

Постановка задачи

Вариант 4.

Цель работы:

Приобретение практических навыков в: 1) Создании аллокаторов памяти и их анализу; 2) Создании динамических библиотек и программ, использующие динамические библиотеки.

Задание

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). Если аргумент не передан или по переданному пути библиотеки не оказалось, то указатели на функции, реализующие API аллокатора ниже, должны быть присвоены функциям, которые оборачивают системный аллокатор ОС (mmap / VirtualAlloc) в этот API. Эти аварийные оберточные функции должны быть реализованы внутри программы, которая загружает динамические библиотеки (см. пример на GitHub Gist). Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям malloc и free (realloc, опционально). Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра (mmap / VirtualAlloc). Необходимо самостоятельно разработать стратегию тестирования для определения ключевых характеристик аллокаторов памяти. При тестировании нужно свести к минимуму потери точности из-за накладных расходов при измерении ключевых характеристик, описанных выше. Каждый аллокатор должен обладать следующим интерфейсом (могут быть отличия в зависимости от особенностей алгоритма):

Allocator* allocator_create(void *const memory, const size_t size) (инициализация аллокатора на памяти memory размера size);

void allocator_destroy(Allocator *const allocator) (деинициализация структуры аллокатора);

void* allocator_alloc(Allocator *const allocator, const size_t size) (выделение памяти аллокатором памяти размера size);

void allocator_free(Allocator *const allocator, void *const memory) (возвращает выделенную память аллокатору);

Необходимо реализовать:

4. Алгоритм Мак-Кьюзика-Кэрелса и блоки по 2^n;

Общий метод и алгоритм решения

Использованные системные вызовы:

- mmap() – для выделения блока памяти (аналог malloc), используется для резервирования памяти для аллокатора.
- munmap() – для освобождения ранее выделенной памяти (аналог free), применяется для очистки ресурсов, выделенных через mmap().
- dlopen() – для загрузки динамической библиотеки во время выполнения программы, используется для подключения аллокатора из внешней библиотеки.
- dlsym() – для получения указателей на функции из динамически загруженной библиотеки, позволяет использовать функции аллокатора.
- dlclose() – для закрытия загруженной динамической библиотеки, освобождает ресурсы, связанные с загрузкой библиотеки.

Реализованы аллокаторы:

1. Аллокатор McKusick-Karels: аллокатор использует стратегию распределения памяти, где блоки памяти организуются в свободные списки, что позволяет эффективно управлять памятью и избежать фрагментации. Он использует списки свободных блоков и делит их на разные группы по размеру.
2. Аллокатор блоков 2^n использует стратегию, где память делится на блоки кратные степеням двойки (например, 16, 32, 64 байт). Для каждого размера блоков существует список свободных блоков (free list). При выделении памяти запрашиваемый размер округляется до ближайшего 2^n, и блок извлекается из соответствующего списка. Если подходящий блок отсутствует, используется блок большего размера, который расщепляется на два меньших. Освобожденные блоки возвращаются в списки, и если соседние блоки свободны, они объединяются, уменьшая фрагментацию. Эта стратегия обеспечивает быстрое выделение, эффективное управление памятью, но может страдать от внутренней фрагментации, если размер запрашиваемой памяти меньше выделенного блока.

Сравнение аллокаторов:

Вот сравнение двух подходов: **алгоритма Мак-Кьюзика-Кэрелса** и **блоков размером 2^n** по указанным критериям.

1. Фактор использования памяти

Фактор использования памяти оценивает, насколько эффективно используется доступная память, минимизируя фрагментацию.

Алгоритм Мак-Кьюзика-Кэрелса

- Этот алгоритм использует **"слоящиеся списки"** (slab allocation), где память разбивается на фиксированные блоки, соответствующие частым размерам запросов.
- Фрагментация минимальная, так как для каждого размера выделяется отдельный список блоков.
- Затраты на память зависят от того, насколько хорошо аллокатор предугадывает типичные запросы.
- Подходит для систем, где запросы памяти предсказуемы и небольшие.

Блоки по 2^n

- Память разбивается на блоки степеней двойки (2^n), а каждый запрос округляется до ближайшей степени 2.
- Этот подход страдает от **внутренней фрагментации**: если запрашивается 33 байта, выделяется блок размером 64 байта.
- При небольших запросах фрагментация может стать существенной.

Итого Алгоритм Мак-Кьюзика-Кэрелса лучше в факторе использование

2. Скорость выделения блоков

Скорость выделения зависит от алгоритма поиска свободных блоков и структуры данных, которая поддерживает аллокатор.

Алгоритм Мак-Кьюзика-Кэрелса

- Использует предварительно выделенные списки блоков. Если в списке для определенного размера есть свободные блоки, выделение происходит мгновенно.
- Если блоков нет, требуется выделение из пула или новой страницы памяти, что занимает больше времени.
- Эффективен при повторных запросах типичных размеров.

Блоки по 2^n

- Для округления размера до степени двойки достаточно простой арифметической операции, после чего выполняется выделение из соответствующего списка блоков.
- Простота структуры (битовая карта или массив списков) делает этот метод быстрым.
- Однако при необходимости выделения больших блоков могут быть накладные расходы на объединение блоков или выделение новой страницы.

Таким образом:
При частых малых запросах: **Блоки по 2^n** быстрее.
При разнообразных запросах: **Мак-Кьюзик-Кэрелс** эффективнее, так как он лучше подстраивается под размеры запросов.

3. Скорость освобождения блоков

Скорость освобождения зависит от структуры данных, в которую возвращаются блоки.

Алгоритм Мак-Кьюзика-Кэрелса

- Освобождение происходит путем возврата блока в соответствующий список. Это занимает константное время (O(1)).
- Если освобождается весь список, возможно возврат страницы памяти в систему, что увеличивает затраты.

Блоки по 2^n

- Освобождение также выполняется за константное время, так как блоки возвращаются в заранее определенные списки.
- Никаких дополнительных операций, кроме обновления списка, не требуется.

4. Простота использования аллокатора

Алгоритм Мак-Кьюзика-Кэрелса

- Требует ручной настройки размеров списков и анализа типичных запросов.
- Если размер блока не соответствует доступным в списках, выделение будет дорогим.

- Сложнее в реализации, но хорошо работает в системах с предсказуемыми нагрузками.

Блоки по 2^n

- Прост в реализации, так как каждый запрос автоматически округляется до ближайшей степени 2.
- Не требует сложной настройки, подходит для общего назначения. Однако из-за внутренней фрагментации может потребоваться дополнительный анализ использования памяти.
  Итого Блоки по 2^n проще в реализации и использовании.

Сравним производительность двух аллокаторов:

1. **Аллокатор Мак-Кьюзика-Кэрелса** (libmckusick_carels.so)
2. **Аллокатор блоков 2^n** (libblock_2n.so)

Методика:

Для каждого аллокатора были запущены несколько последовательных тестов с использованием утилиты hyperfine для измерения времени выполнения.

Результаты тестов:

1. **Аллокатор блоков 2^n** (libblock_2n.so):
   - Среднее время выполнения: **0.3 ms**
   - Стандартное отклонение: **0.1 ms**
   - Минимальное время: **0.2 ms**
   - Максимальное время: **0.8 ms**
   - Количество запусков: **4628**
2. **Аллокатор Мак-Кьюзика-Кэрелса** (libmckusick_carels.so):
   - Среднее время выполнения: **0.2 ms**
   - Стандартное отклонение: **0.2 ms**
   - Минимальное время: **0.2 ms**
   - Максимальное время: **9.2 ms**
   - Количество запусков: **4825**

Выводы:

- **Время выполнения**: Аллокатор Мак-Кьюзика-Кэрелса показал немного более высокую скорость выполнения по сравнению с аллокатором блоков 2^n (на 5% быстрее).

Сравнение скорости выделения и освобождения блоков

Тест	**Аллокатор блоков 2^n** (ns)	**Аллокатор Мак-Кьюзика-Кэрелса** (ns)	Более быстрый
Test 1	Allocate: 130	Allocate: 81	Аллокатор Мак-Кьюзика-Кэрелса
Test 1	Free: 40	Free: 40	Ничья
Test 2	Allocate: 61	Allocate: 60	Ничья

Test 2	Free: 131	Free: 40	Аллокатор Мак-Кьюзика-Кэрелса
Test 3	Allocate: 100	Allocate: 30	Аллокатор Мак-Кьюзика-Кэрелса
Test 3	Free: 130	Free: 30	Аллокатор Мак-Кьюзика-Кэрелса
Test 4	Allocate: 90	Allocate: 30	Аллокатор Мак-Кьюзика-Кэрелса
Test 4	Free: 81	Free: 30	Аллокатор Мак-Кьюзика-Кэрелса

Таким образом, аллокатор Мак-Кьюзика-Кэрелса демонстрирует лучшую производительность

Код программы

mckusick_carels.h

```
//
// Created by Qwental on 22.12.2024.
//

#ifndef MCKUSICK_CARELS_H
#define MCKUSICK_CARELS_H

#include <stddef.h>

// Макросы для выравнивания
#define ALIGN_SIZE(size, alignment) (((size) + (alignment - 1)) & ~(alignment - 1))
#define FREE_LIST_ALIGNMENT 8 // Выравнивание списка свободных блоков


typedef struct FreeBlock {
    struct FreeBlock *next_block;
} FreeBlock;

typedef struct MemoryAllocator {
    void *memory_start;
    size_t memory_size;
    FreeBlock *free_list_head;
} MemoryAllocator;


MemoryAllocator *allocator_create(void *memory_pool, size_t total_size);

void allocator_destroy(MemoryAllocator *allocator);

void *allocator_alloc(MemoryAllocator *allocator, size_t size);

void allocator_free(MemoryAllocator *allocator, void *memory_block);

#endif // MCKUSICK_CARELS_H
```

mckusick_carels.c

```


//
// Created by Qwental on 22.12.2024.
//
```

```c
#include "mckusick_carels.h"


MemoryAllocator *allocator_create(void *memory_pool, size_t total_size) {
    if (memory_pool == NULL || total_size < sizeof(MemoryAllocator)) {
        return NULL;
    }

    MemoryAllocator *allocator = (MemoryAllocator *)memory_pool;
    allocator->memory_start = (char *)memory_pool + sizeof(MemoryAllocator);
    allocator->memory_size = total_size - sizeof(MemoryAllocator);
    allocator->free_list_head = (FreeBlock *)allocator->memory_start;

    // Инициализация списка
    if (allocator->free_list_head != NULL) {
        allocator->free_list_head->next_block = NULL;
    }

    return allocator;
}

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

    allocator->memory_start = NULL;
    allocator->memory_size = 0;
    allocator->free_list_head = NULL;
}


void *allocator_alloc(MemoryAllocator *allocator, size_t size) {
    if (allocator == NULL || size == 0) {
        return NULL;
    }

    size_t aligned_size = ALIGN_SIZE(size, FREE_LIST_ALIGNMENT);
    FreeBlock *previous_block = NULL;
    FreeBlock *current_block = allocator->free_list_head;

    while (current_block != NULL) {
        if (aligned_size <= allocator->memory_size) {
            if (previous_block != NULL) {
                previous_block->next_block = current_block->next_block;
            } else {
                allocator->free_list_head = current_block->next_block;
            }
            return current_block;
        }

        previous_block = current_block;
        current_block = current_block->next_block;
    }

    return NULL;
}


void allocator_free(MemoryAllocator *allocator, void *memory_block) {
```

```
    if (allocator == NULL || memory_block == NULL) {
        return;
    }

    FreeBlock *block_to_free = (FreeBlock *)memory_block;
    block_to_free->next_block = allocator->free_list_head;
    allocator->free_list_head = block_to_free;
}
```

block_2n.h

```
#ifndef BLOCK_2N_H
#define BLOCK_2N_H

#include <math.h>
#include <stdint.h>
#include <sys/mman.h>
#include <unistd.h>

#define OFFSET_FREE_LIST(index, allocator) ((Block **) ((char *)(allocator) +
sizeof(Allocator)))[index]
#define BLOCK_MIN_SIZE 16
#define BLOCK_DEFAULT_SIZE 32
#define DIVISOR_LOG2 2
#define BLOCK_MAX_SIZE(size) (((size) < BLOCK_DEFAULT_SIZE) ? BLOCK_DEFAULT_SIZE :
(size))
#define SELECT_LAST_LIST(index, num_lists) (((index) >= (num_lists)) ? ((num_lists) - 1)
: (index))


typedef struct Block {
    struct Block *next_block;
} Block;


typedef struct Allocator {
    Block **free_lists;
    size_t num_lists;
    void *base_addr;
    size_t total_size;
} Allocator;


Allocator *allocator_create(void *memory_region, size_t region_size);
void *allocator_alloc(Allocator *allocator, size_t alloc_size);
void allocator_free(Allocator *allocator, void *memory_pointer);
void allocator_destroy(Allocator *allocator);

#endif // BLOCK_2N_H
```

block_2n.c

```c
#include "block_2n.h"


Allocator *allocator_create(void *memory_region, size_t region_size) {
    if (memory_region == NULL || region_size < sizeof(Allocator)) {
        return NULL;
    }

    Allocator *allocator = (Allocator *) memory_region;
    allocator->base_addr = memory_region;
    allocator->total_size = region_size;

    const size_t min_usable = sizeof(Block) + BLOCK_MIN_SIZE;
    size_t max_block = BLOCK_MAX_SIZE(region_size);

    allocator->num_lists = (size_t) floor(log2_calc(max_block) / DIVISOR_LOG2) + 3;
    allocator->free_lists = (Block **) ((char *) memory_region + sizeof(Allocator));

    for (size_t i = 0; i < allocator->num_lists; i++) {
        allocator->free_lists[i] = NULL;
    }

    void *current_block = (char *) memory_region + sizeof(Allocator) +
                          allocator->num_lists * sizeof(Block *);
    size_t remaining_size = region_size - sizeof(Allocator) - allocator->num_lists *
sizeof(Block *);

    size_t block_size = BLOCK_MIN_SIZE;
    while (remaining_size >= min_usable) {
        if (block_size > remaining_size || block_size > max_block) {
            break;
        }

        if (remaining_size >= (block_size + sizeof(Block)) * 2) {
            for (int i = 0; i < 2; i++) {
                Block *block_header = (Block *) current_block;
                size_t list_index = log2_calc(block_size);
                block_header->next_block = allocator->free_lists[list_index];
                allocator->free_lists[list_index] = block_header;

                current_block = (char *) current_block + block_size;
                remaining_size -= block_size;
            }
        } else {
            Block *block_header = (Block *) current_block;
            size_t list_index = log2_calc(block_size);
            block_header->next_block = allocator->free_lists[list_index];
            allocator->free_lists[list_index] = block_header;

            current_block = (char *) current_block + remaining_size;
            remaining_size = 0;
        }

        block_size <<= 1;
    }
    return allocator;
}


void *allocator_alloc(Allocator *allocator, size_t alloc_size) {
```

```
    if (allocator == NULL || alloc_size == 0) {
        return NULL;
    }

    size_t list_index = log2_calc(alloc_size) + 1;
    list_index = SELECT_LAST_LIST(list_index, allocator->num_lists);

    while (list_index < allocator->num_lists && allocator->free_lists[list_index] ==
NULL) {
        list_index++;
    }

    if (list_index >= allocator->num_lists) {
        return NULL;
    }

    Block *allocated_block = allocator->free_lists[list_index];
    allocator->free_lists[list_index] = allocated_block->next_block;

    return (void *) ((char *) allocated_block + sizeof(Block));
}


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

    Block *block_to_free = (Block *) ((char *) memory_pointer - sizeof(Block));
    size_t offset = (char *) block_to_free - (char *) allocator->base_addr;
    size_t temp_size = BLOCK_DEFAULT_SIZE;

    while (temp_size <= offset) {
        size_t next_size = temp_size << 1;
        if (next_size > offset) {
            break;
        }
        temp_size = next_size;
    }

    size_t list_index = log2_calc(temp_size);
    list_index = SELECT_LAST_LIST(list_index, allocator->num_lists);

    block_to_free->next_block = allocator->free_lists[list_index];
    allocator->free_lists[list_index] = block_to_free;
}


void allocator_destroy(Allocator *allocator) {
    if (allocator != NULL) {
        munmap(allocator->base_addr, allocator->total_size);
    }
}
```

main.c

```
// Created by Qwental on 22.12.2024.

/*
gcc -shared -fPIC -o libmckusick_carels.so mckusick_carels.c
gcc -shared -fPIC -o libblock_2n.so block_2n.c
```

```
gcc -o main main.c -ldl

./main ./libblock_2n.so
./main ./libmckusick_carels.so
*/

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>
#include <time.h>
#define SNPRINTF_BUF 256

typedef void *(*allocator_create_t)(void *const memory, const size_t size);

typedef void *(*allocator_alloc_t)(void *const allocator, const size_t size);

typedef void (*allocator_free_t)(void *const allocator, void *const memory);

typedef void (*allocator_destroy_t)(void *const allocator);

typedef struct {
    char first_name[52];
    char last_name[52];
    char group[15];
} Student;

void write_message(const char *message) {
    write(STDOUT_FILENO, message, strlen(message));
}

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

long long calculate_time_diff_ns(const struct timespec start, const struct timespec end)
{
    return (end.tv_sec - start.tv_sec) * 1000000000LL + (end.tv_nsec - start.tv_nsec);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        write_error("Usage: <program> <path_to_allocator_library>\n");
        return 52;
    }

    void *allocator_lib = dlopen(argv[1], RTLD_LAZY);
    if (allocator_lib == NULL) {
        write_error("Error loading library: ");
        write_error(dlerror());
        write_error("\n");
        return 1;
    }

    allocator_create_t allocator_create = (allocator_create_t) dlsym(allocator_lib,
"allocator_create");
    allocator_alloc_t allocator_alloc = (allocator_alloc_t) dlsym(allocator_lib,
"allocator_alloc");
```

```
    allocator_free_t allocator_free = (allocator_free_t) dlsym(allocator_lib,
"allocator_free");
    allocator_destroy_t allocator_destroy = (allocator_destroy_t) dlsym(allocator_lib,
"allocator_destroy");

    if (!allocator_create || !allocator_alloc || !allocator_free || !allocator_destroy)
{
        write_error("Error locating functions: ");
        write_error(dlerror());
        write_error("\n");
        dlclose(allocator_lib);
        return 1;
    }

    size_t pool_size = 4 * 1024 * 1024; // 4 MB
    void *memory_pool = mmap(NULL, pool_size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
    if (memory_pool == MAP_FAILED) {
        write_error("Memory allocation for pool failed (mmap)\n");
        dlclose(allocator_lib);
        return 1;
    }
    write_message("POOL SIZE is 4 MB = 4 * 1024 * 1024\n");

    void *allocator = allocator_create(memory_pool, pool_size);
    if (!allocator) {
        write_error("Allocator creation failed\n");
        munmap(memory_pool, pool_size);
        dlclose(allocator_lib);
        return 1;
    }

    struct timespec start, end;

    write_message("\n===============TEST1==============\n\n");
    clock_gettime(CLOCK_MONOTONIC, &start);
    Student *stud_array = (Student *) allocator_alloc(allocator, 5 * sizeof(Student));
    clock_gettime(CLOCK_MONOTONIC, &end);

    if (stud_array) {
        char buffer[SNPRINTF_BUF];
        snprintf(buffer, sizeof(buffer), "[[LOG]]Time to allocate: %lld ns\n",
calculate_time_diff_ns(start, end));
        write_message(buffer);

        for (int i = 0; i < 5; i++) {
            snprintf(stud_array[i].first_name, sizeof(stud_array[i].first_name),
"Vladimir%d", i + 1);
            snprintf(stud_array[i].last_name, sizeof(stud_array[i].last_name),
"Bugrenkov%d", i + 1);
            snprintf(stud_array[i].group, sizeof(stud_array[i].group), "M8O-21%d-23",
i);
        }

        write_message("Students before shuffle:\n");
        for (int i = 0; i < 5; i++) {
            snprintf(buffer, sizeof(buffer), "Name: %s, Last Name: %s, Group: %s\n",
                     stud_array[i].first_name, stud_array[i].last_name,
stud_array[i].group);
            write_message(buffer);
```

```
        }

        Student temp = stud_array[0];
        stud_array[0] = stud_array[4];
        stud_array[4] = temp;

        write_message("Students after shuffle:\n");
        for (int i = 0; i < 5; i++) {
            snprintf(buffer, sizeof(buffer), "Name: %s, Last Name: %s, Group: %s\n",
                     stud_array[i].first_name, stud_array[i].last_name,
stud_array[i].group);
            write_message(buffer);
        }

        clock_gettime(CLOCK_MONOTONIC, &start);
        allocator_free(allocator, stud_array);
        clock_gettime(CLOCK_MONOTONIC, &end);
        snprintf(buffer, sizeof(buffer), "[[LOG]]Time to free block: %lld ns\n",
calculate_time_diff_ns(start, end));
        write_message(buffer);
    } else {
        write_error("Memory allocation for students failed\n");
    }


    write_message("\n===============TEST2===============\n\n");
    clock_gettime(CLOCK_MONOTONIC, &start);
    long double *long_double_array = (long double *) allocator_alloc(allocator, 20 *
sizeof(long double));
    clock_gettime(CLOCK_MONOTONIC, &end);

    if (long_double_array) {
        char buffer[SNPRINTF_BUF];
        snprintf(buffer, sizeof(buffer), "[[LOG]]Time to allocate: %lld ns\n",
calculate_time_diff_ns(start, end));
        write_message(buffer);

        for (int i = 0; i < 20; i++) {
            long_double_array[i] = i + 1;
        }
        write_message("array before mulp52:\n");
        for (int i = 0; i < 20; i++) {
            char buffer[SNPRINTF_BUF];
            snprintf(buffer, sizeof(buffer), "long_double_array[%d] = %Lf\n", i,
long_double_array[i]);
            write_message(buffer);
        }
        for (int i = 0; i < 20; i++) {
            long_double_array[i] *= 52;
        }
        write_message("array after mulp52:\n");
        for (int i = 0; i < 20; i++) {
            char buffer[SNPRINTF_BUF];
            snprintf(buffer, sizeof(buffer), "long_double_array[%d] = %Lf\n", i,
long_double_array[i]);
            write_message(buffer);
        }
        clock_gettime(CLOCK_MONOTONIC, &start);
        allocator_free(allocator, long_double_array);
        clock_gettime(CLOCK_MONOTONIC, &end);
```

```
        snprintf(buffer, sizeof(buffer), "[[LOG]]Time to free block: %lld ns\n",
calculate_time_diff_ns(start, end));
        write_message(buffer);
    } else {
        write_error("Memory allocation for long double array failed\n");
    }
    write_message("\n===============TEST3==============\n\n");
    clock_gettime(CLOCK_MONOTONIC, &start);
    long double *large_array = (long double *) allocator_alloc(allocator, 5001 *
sizeof(long double));
    clock_gettime(CLOCK_MONOTONIC, &end);


    if (large_array) {
        char buffer[SNPRINTF_BUF];
        snprintf(buffer, sizeof(buffer), "[[LOG]]Time to allocate: %lld ns\n",
calculate_time_diff_ns(start, end));
        write_message(buffer);
        large_array[5000] = 52;
        snprintf(buffer, sizeof(buffer), "large_array[5000] = %Lf\n",
large_array[5000]);
        write_message(buffer);
        write_message("Large array (5001 long double) allocated and freed
successfully\n");

        clock_gettime(CLOCK_MONOTONIC, &start);
        allocator_free(allocator, large_array);
        clock_gettime(CLOCK_MONOTONIC, &end);
        snprintf(buffer, sizeof(buffer), "[[LOG]]Time to free block: %lld ns\n",
calculate_time_diff_ns(start, end));
        write_message(buffer);
    } else {
        write_error("Memory allocation for large array failed\n");
    }

    write_message("\n===============TEST4==============\n\n");
    clock_gettime(CLOCK_MONOTONIC, &start);

    void *block1 = allocator_alloc(allocator, 3 * 1024 * 1024);

    clock_gettime(CLOCK_MONOTONIC, &end);

    if (block1) {
        char buffer[SNPRINTF_BUF];
        snprintf(buffer, sizeof(buffer), "[[LOG]]Time to allocate: %lld ns\n",
calculate_time_diff_ns(start, end));
        write_message(buffer);

        write_message("Memory allocated (3 * 1024 * 1024 bytes)\n");
        clock_gettime(CLOCK_MONOTONIC, &start);
        allocator_free(allocator, block1);
        clock_gettime(CLOCK_MONOTONIC, &end);
        snprintf(buffer, sizeof(buffer), "[[LOG]]Time to free block: %lld ns\n",
calculate_time_diff_ns(start, end));
        write_message(buffer);
        write_message("Memory freed (3 * 1024 * 1024 bytes)\n");
    } else {
        write_error("Memory allocation failed\n");
    }
```

```
    allocator_destroy(allocator);
    write_message("Allocator destroyed\n");
    munmap(memory_pool, pool_size);
    dlclose(allocator_lib);
    return 0;
}
```

Протокол работы программы

qwental@DESKTOP-NKF1EUK:~/workspace/OS_LABS/lab4/src$ ./compare_allocators.sh

Компиляция libmckusick_carels.so...

libmckusick_carels.so успешно скомпилирована.

Компиляция libblock_2n.so...

libblock_2n.so успешно скомпилирована.

Компиляция main.c...

main успешно скомпилирован.

=== Сравнение времени выполнения аллокаторов ===

Benchmark 1: ./main ./libblock_2n.so

Time (mean ± σ): 0.4 ms ± 0.3 ms [User: 0.4 ms, System: 0.0 ms]

Range (min … max): 0.3 ms … 13.8 ms 3990 runs

Warning: Command took less than 5 ms to complete. Results might be inaccurate.

Warning: Statistical outliers were detected. Consider re-running this benchmark on a quiet PC without any interferences from other programs. It might help to use the '--warmup' or '--prepare' options.

Benchmark 2: ./main ./libmckusick_carels.so

Time (mean ± σ): 0.3 ms ± 0.1 ms [User: 0.3 ms, System: 0.0 ms]

Range (min … max): 0.2 ms … 1.7 ms 4701 runs

Warning: Command took less than 5 ms to complete. Results might be inaccurate.

Warning: Statistical outliers were detected. Consider re-running this benchmark on a quiet PC without any interferences from other programs. It might help to use the '--warmup' or '--prepare' options.

Summary

'./main ./libmckusick_carels.so' ran

1.26 ± 0.93 times faster than './main ./libblock_2n.so'

Сравнение завершено.

Компиляция libmckusick_carels.so...

libmckusick_carels.so успешно скомпилирована.

Компиляция libblock_2n.so...

libblock_2n.so успешно скомпилирована.

Компиляция main.c...

main успешно скомпилирован.

Тестирование с libblock_2n.so...

POOL SIZE is 4 MB = 4 * 1024 * 1024

==========TEST1==========

[[LOG]]Time to allocate: 151 ns

Students before shuffle:

Name: Vladimir1, Last Name: Bugrenkov1, Group: M8O-210-23

Name: Vladimir2, Last Name: Bugrenkov2, Group: M8O-211-23

Name: Vladimir3, Last Name: Bugrenkov3, Group: M8O-212-23

Name: Vladimir4, Last Name: Bugrenkov4, Group: M8O-213-23

Name: Vladimir5, Last Name: Bugrenkov5, Group: M8O-214-23

Students after shuffle:

Name: Vladimir5, Last Name: Bugrenkov5, Group: M8O-214-23

Name: Vladimir2, Last Name: Bugrenkov2, Group: M8O-211-23

Name: Vladimir3, Last Name: Bugrenkov3, Group: M8O-212-23

Name: Vladimir4, Last Name: Bugrenkov4, Group: M8O-213-23

Name: Vladimir1, Last Name: Bugrenkov1, Group: M8O-210-23

[[LOG]]Time to free block: 181 ns

==========TEST2==========

[[LOG]]Time to allocate: 70 ns

array before mulp52:

long_double_array[0] = 1.000000

long_double_array[1] = 2.000000

long_double_array[2] = 3.000000

long_double_array[3] = 4.000000

long_double_array[4] = 5.000000

long_double_array[5] = 6.000000

long_double_array[6] = 7.000000

long_double_array[7] = 8.000000

long_double_array[8] = 9.000000

long_double_array[9] = 10.000000

long_double_array[10] = 11.000000

long_double_array[11] = 12.000000

long_double_array[12] = 13.000000

long_double_array[13] = 14.000000

long_double_array[14] = 15.000000

long_double_array[15] = 16.000000

long_double_array[16] = 17.000000

long_double_array[17] = 18.000000

long_double_array[18] = 19.000000

long_double_array[19] = 20.000000

array after mulp52:

long_double_array[0] = 52.000000

long_double_array[1] = 104.000000

long_double_array[2] = 156.000000

long_double_array[3] = 208.000000

long_double_array[4] = 260.000000

long_double_array[5] = 312.000000

long_double_array[6] = 364.000000

long_double_array[7] = 416.000000

long_double_array[8] = 468.000000

long_double_array[9] = 520.000000

long_double_array[10] = 572.000000

long_double_array[11] = 624.000000

long_double_array[12] = 676.000000

long_double_array[13] = 728.000000

long_double_array[14] = 780.000000

long_double_array[15] = 832.000000

long_double_array[16] = 884.000000

long_double_array[17] = 936.000000

long_double_array[18] = 988.000000

long_double_array[19] = 1040.000000

[[LOG]]Time to free block: 80 ns

==========TEST3==========

[[LOG]]Time to allocate: 120 ns

large_array[5000] = 52.000000

Large array (5001 long double) allocated and freed successfully

[[LOG]]Time to free block: 100 ns

==========TEST4==========

[[LOG]]Time to allocate: 60 ns

Memory allocated (3 * 1024 * 1024 bytes)

[[LOG]]Time to free block: 80 ns

Memory freed (3 * 1024 * 1024 bytes)

Allocator destroyed

Тестирование с libblock_2n.so завершено.

Тестирование с libmckusick_carels.so...

POOL SIZE is 4 MB = 4 * 1024 * 1024

==========TEST1==========

[[LOG]]Time to allocate: 80 ns

Students before shuffle:

Name: Vladimir1, Last Name: Bugrenkov1, Group: M8O-210-23

Name: Vladimir2, Last Name: Bugrenkov2, Group: M8O-211-23

Name: Vladimir3, Last Name: Bugrenkov3, Group: M8O-212-23

Name: Vladimir4, Last Name: Bugrenkov4, Group: M8O-213-23

Name: Vladimir5, Last Name: Bugrenkov5, Group: M8O-214-23

Students after shuffle:

Name: Vladimir5, Last Name: Bugrenkov5, Group: M8O-214-23

Name: Vladimir2, Last Name: Bugrenkov2, Group: M8O-211-23

Name: Vladimir3, Last Name: Bugrenkov3, Group: M8O-212-23

Name: Vladimir4, Last Name: Bugrenkov4, Group: M8O-213-23

Name: Vladimir1, Last Name: Bugrenkov1, Group: M8O-210-23

[[LOG]]Time to free block: 40 ns

==========TEST2==========

[[LOG]]Time to allocate: 40 ns

array before mulp52:

long_double_array[0] = 1.000000

long_double_array[1] = 2.000000

long_double_array[2] = 3.000000

long_double_array[3] = 4.000000

long_double_array[4] = 5.000000

long_double_array[5] = 6.000000

long_double_array[6] = 7.000000

long_double_array[7] = 8.000000

long_double_array[8] = 9.000000

long_double_array[9] = 10.000000

long_double_array[10] = 11.000000

long_double_array[11] = 12.000000

long_double_array[12] = 13.000000

long_double_array[13] = 14.000000

long_double_array[14] = 15.000000

long_double_array[15] = 16.000000

long_double_array[16] = 17.000000

long_double_array[17] = 18.000000

long_double_array[18] = 19.000000

long_double_array[19] = 20.000000

array after mulp52:

long_double_array[0] = 52.000000

long_double_array[1] = 104.000000

long_double_array[2] = 156.000000

long_double_array[3] = 208.000000

long_double_array[4] = 260.000000

long_double_array[5] = 312.000000

long_double_array[6] = 364.000000

long_double_array[7] = 416.000000

long_double_array[8] = 468.000000

long_double_array[9] = 520.000000

long_double_array[10] = 572.000000

long_double_array[11] = 624.000000

long_double_array[12] = 676.000000

long_double_array[13] = 728.000000

long_double_array[14] = 780.000000

long_double_array[15] = 832.000000

long_double_array[16] = 884.000000

long_double_array[17] = 936.000000

long_double_array[18] = 988.000000

long_double_array[19] = 1040.000000

[[LOG]]Time to free block: 30 ns

==========TEST3==========

[[LOG]]Time to allocate: 40 ns

large_array[5000] = 52.000000

Large array (5001 long double) allocated and freed successfully

[[LOG]]Time to free block: 30 ns

==========TEST4==========

[[LOG]]Time to allocate: 30 ns

Memory allocated (3 * 1024 * 1024 bytes)

[[LOG]]Time to free block: 30 ns

Memory freed (3 * 1024 * 1024 bytes)

Allocator destroyed

Тестирование с libmckusick_carels.so завершено.

Все библиотеки успешно скомпилированы и протестированы.

Strace:

```
execve("./main", ["./main", "./libmckusick_carels.so"], 0x7fff95e2d378 /* 20 vars */) = 0
brk(NULL)                               = 0x564e2ec8b000
```

```
arch_prctl(0x3001 /* ARCH_??? */, 0x7fff579a0570) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f741389c000
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=31847, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 31847, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f7413894000
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) = 0x7f741366b000
mprotect(0x7f7413693000, 2023424, PROT_NONE) = 0
mmap(0x7f7413693000, 1658880, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f7413693000
mmap(0x7f7413828000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x1bd000) = 0x7f7413828000
mmap(0x7f7413881000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x7f7413881000
mmap(0x7f7413887000, 52816, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f7413887000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f7413668000
arch_prctl(ARCH_SET_FS, 0x7f7413668740) = 0
set_tid_address(0x7f7413668a10)         = 17729
set_robust_list(0x7f7413668a20, 24)     = 0
rseq(0x7f74136690e0, 0x20, 0, 0x53053053) = 0
mprotect(0x7f7413881000, 16384, PROT_READ) = 0
mprotect(0x564e2de4b000, 4096, PROT_READ) = 0
mprotect(0x7f74138d6000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f7413894000, 31847)           = 0
getrandom("\x7d\x76\x0a\x95\x39\x42\x92\x0d", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x564e2ec8b000
brk(0x564e2ecac000)                     = 0x564e2ecac000
openat(AT_FDCWD, "./libmckusick_carels.so", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=15272, ...}, AT_EMPTY_PATH) = 0
getcwd("/home/qwental/workspace/OS_LABS/lab4/src", 128) = 41
mmap(NULL, 16424, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7413897000
mmap(0x7f7413898000, 4096, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7f7413898000
mmap(0x7f7413899000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x2000) = 0x7f7413899000
mmap(0x7f741389a000, 8192, PROT_READ|PROT_WRITE,
```

MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f741389a000
close(3)   = 0
mprotect(0x7f741389a000, 4096, PROT_READ) = 0
mmap(NULL, 4194304, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7413268000
write(1, "POOL SIZE is 4 MB = 4 * 1024 * 1"..., 36) = 36
write(1, "\n==============TEST1==========="..., 38) = 38
write(1, "[[LOG]]Time to allocate: 190 ns\n", 32) = 32
write(1, "Students before shuffle:\n", 25) = 25
write(1, "Name: Vladimir1, Last Name: Bugr"..., 60) = 60
write(1, "Name: Vladimir2, Last Name: Bugr"..., 60) = 60
write(1, "Name: Vladimir3, Last Name: Bugr"..., 60) = 60
write(1, "Name: Vladimir4, Last Name: Bugr"..., 60) = 60
write(1, "Name: Vladimir5, Last Name: Bugr"..., 60) = 60
write(1, "Students after shuffle:\n", 24) = 24
write(1, "Name: Vladimir5, Last Name: Bugr"..., 60) = 60
write(1, "Name: Vladimir2, Last Name: Bugr"..., 60) = 60
write(1, "Name: Vladimir3, Last Name: Bugr"..., 60) = 60
write(1, "Name: Vladimir4, Last Name: Bugr"..., 60) = 60
write(1, "Name: Vladimir1, Last Name: Bugr"..., 60) = 60
write(1, "[[LOG]]Time to free block: 160 n"..., 34) = 34
write(1, "\n==============TEST2==========="..., 38) = 38
write(1, "[[LOG]]Time to allocate: 270 ns\n", 32) = 32
write(1, "array before mulp52:\n", 21) = 21
write(1, "long_double_array[0] = 1.000000\n", 32) = 32
write(1, "long_double_array[1] = 2.000000\n", 32) = 32
write(1, "long_double_array[2] = 3.000000\n", 32) = 32
write(1, "long_double_array[3] = 4.000000\n", 32) = 32
write(1, "long_double_array[4] = 5.000000\n", 32) = 32
write(1, "long_double_array[5] = 6.000000\n", 32) = 32
write(1, "long_double_array[6] = 7.000000\n", 32) = 32
write(1, "long_double_array[7] = 8.000000\n", 32) = 32
write(1, "long_double_array[8] = 9.000000\n", 32) = 32
write(1, "long_double_array[9] = 10.000000"..., 33) = 33
write(1, "long_double_array[10] = 11.00000"..., 34) = 34
write(1, "long_double_array[11] = 12.00000"..., 34) = 34
write(1, "long_double_array[12] = 13.00000"..., 34) = 34
write(1, "long_double_array[13] = 14.00000"..., 34) = 34
write(1, "long_double_array[14] = 15.00000"..., 34) = 34
write(1, "long_double_array[15] = 16.00000"..., 34) = 34
write(1, "long_double_array[16] = 17.00000"..., 34) = 34
write(1, "long_double_array[17] = 18.00000"..., 34) = 34
write(1, "long_double_array[18] = 19.00000"..., 34) = 34
write(1, "long_double_array[19] = 20.00000"..., 34) = 34
write(1, "array after mulp52:\n", 20)   = 20
write(1, "long_double_array[0] = 52.000000"..., 33) = 33
write(1, "long_double_array[1] = 104.00000"..., 34) = 34
write(1, "long_double_array[2] = 156.00000"..., 34) = 34
write(1, "long_double_array[3] = 208.00000"..., 34) = 34
write(1, "long_double_array[4] = 260.00000"..., 34) = 34

```
write(1, "long_double_array[5] = 312.00000"..., 34) = 34
write(1, "long_double_array[6] = 364.00000"..., 34) = 34
write(1, "long_double_array[7] = 416.00000"..., 34) = 34
write(1, "long_double_array[8] = 468.00000"..., 34) = 34
write(1, "long_double_array[9] = 520.00000"..., 34) = 34
write(1, "long_double_array[10] = 572.0000"..., 35) = 35
write(1, "long_double_array[11] = 624.0000"..., 35) = 35
write(1, "long_double_array[12] = 676.0000"..., 35) = 35
write(1, "long_double_array[13] = 728.0000"..., 35) = 35
write(1, "long_double_array[14] = 780.0000"..., 35) = 35
write(1, "long_double_array[15] = 832.0000"..., 35) = 35
write(1, "long_double_array[16] = 884.0000"..., 35) = 35
write(1, "long_double_array[17] = 936.0000"..., 35) = 35
write(1, "long_double_array[18] = 988.0000"..., 35) = 35
write(1, "long_double_array[19] = 1040.000"..., 36) = 36
write(1, "[[LOG]]Time to free block: 170 n"..., 34) = 34
write(1, "\n==============TEST3==========="..., 38) = 38
write(1, "[[LOG]]Time to allocate: 200 ns\n", 32) = 32
write(1, "large_array[5000] = 52.000000\n", 30) = 30
write(1, "Large array (5001 long double) a"..., 64) = 64
write(1, "[[LOG]]Time to free block: 91 ns"..., 33) = 33
write(1, "\n==============TEST4==========="..., 38) = 38
write(1, "[[LOG]]Time to allocate: 180 ns\n", 32) = 32
write(1, "Memory allocated (3 * 1024 * 102"..., 41) = 41
write(1, "[[LOG]]Time to free block: 170 n"..., 34) = 34
write(1, "Memory freed (3 * 1024 * 1024 by"..., 37) = 37
write(1, "Allocator destroyed\n", 20)   = 20
munmap(0x7f7413268000, 4194304)        = 0
munmap(0x7f7413897000, 16424)          = 0
exit_group(0)                          = ?
+++ exited with 0 +++
```

Вывод

В ходе работы были реализованы и протестированы два алгоритма аллокации памяти, которые сравнивались по фактору использования, скорости выделения и освобождения блоков, а также простоте использования. Оба аллокатора продемонстрировали эффективное использование пула памяти и интуитивность в применении, что подтвердило практическую ценность их реализации.