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: Амелина А.Е.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 05.01.25

Москва, 2024

Постановка задачи

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.

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);

Вариант 2. Списки свободных блоков (первое подходящее) и алгоритм Мак-Кьюзи-Кэрелса.

Общий метод и алгоритм решения

Использованные системные вызовы:

- ssize_t write(int __fd, const void *__buf, size_t __n); – записывает N байт из буфер(BUF) в файл (FD). Возвращает количество записанных байт или -1.
- void exit(int __status); – выполняет немедленное завершение программы. Все используемые программой потоки закрываются, и временные файлы удаляются, управление возвращается ОС или другой программе.
- void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset); – отражает length байтов, начиная со смещения offset файла (или другого объекта), определенного файловым дескриптором fd, в память, начиная с адреса start.
- int munmap(void *start, size_t length); – удаляет все отражения из заданной области памяти, после чего все ссылки на данную область будут вызывать ошибку "неправильное обращение к памяти".
- void *dlopen(const char *filename, int flag); – загружает динамическую библиотеку, имя которой указано в строке filename, и возвращает прямой указатель на начало динамической библиотеки.
- void *dlsym(void *handle, char *symbol); – использует указатель на динамическую библиотеку, возвращаемую dlopen, и оканчивающееся нулем символьное имя, а затем возвращает адрес, указывающий, откуда загружается этот символ. Если символ не найден, то возвращаемым значением dlsym является NULL;
- int dlclose(void *handle); – уменьшает на единицу счетчик ссылок на указатель динамической библиотеки.

Аллокатор памяти на основе списка свободных блоков

Этот аллокатор поддерживает список свободных блоков памяти. Каждый блок в этом списке представляет собой участок памяти, который в данный момент не используется программой. Когда программе требуется выделить память, аллокатор ищет подходящий блок в этом списке. После того как память больше не нужна, она возвращается в список свободных блоков.

Список свободных блоков представляет собой структуру данных, где каждый элемент описывает свободный участок памяти.

Структуры данных:

Каждый элемент (блок) представляется структурой Block и содержит:

- size - размер блока в байтах
- next – указатель на следующий свободный блок в списке

Структура аллокатора Allocator содержит:

- memory – указатель на выделенную память
- size – общий размер выделенной памяти
- free_list – указатель на голову списка свободных блоков

Функции:

1) Инициализация аллокатора – Allocator* allocator_create(void *memory, size_t size)
   Создает аллокатор и инициализирует список свободных блоков.
   1. Выделяет начальный блок памяти с помощью mmap.
   2. Инициализирует поля аллокатора:

   - size - размер блока в байтах
   - next – указатель на следующий свободный блок в списке
   - free_list – указывает на начало переданной памяти

   3. Возвращает указатель на созданный аллокатор.

2) Уничтожение аллокатора – void allocator_destroy(Allocator *allocator)
   Освобождает память, выделенную для структуры Allocator.

3) Выделение памяти – void* allocator_alloc(Allocator* allocator, size_t size)
   Выделяет блок памяти запрошенного размера.
   1. Проходит по списку свободных блоков, начиная с головы
   2. Ищет первый блок, размер которого больше или равен запрошенному(First-Fit).
   3. Если найден подходящий блок:
   - Если блок больше, чем нужно, он разделяется на два: один выделяется под запрос, второй(оставшаяся часть) добавляется в список свободных блоков.
   - Указатель на выделенный блок возвращается пользователю.
   4. Если подходящий блок не найден, возвращает NULL.

4) Освобождение памяти – void allocator_free(Allocator* allocator, void* memory)
   Возвращает выделенный блок памяти в список свободных элементов.
   1. Получает указатель на заголовок блока (Block), который находится перед выделенной памятью.
   2. Добавляет освобожденный блок в начало списка свободных блоков.
   3. Обновляет голову списка(free_list).

Аллокатор памяти на основе алгоритма Мак-Кьюзика-Кэрелса

Программа реализует аллокатор памяти на основе алгоритма Мак-Кьюзика-Кэрелса. Этот алгоритм использует несколько списков свободных блоков для управления памятью. Каждый список соответствует определенному размеру блоков. Когда требуется выделить память, аллокатор ищет подходящий свободный блок в соответствующем списке. После использования блоки возвращаются обратно в список.

Основные компоненты:

1. Списки свободных блоков:

   - В данном аллокаторе используется один список свободных блоков (free_list), который хранит указатели на свободные участки памяти.
   - Каждый элемент списка представляет собой структуру Block, которая содержит информацию о размере блока и указатель на следующий свободный блок.

2. Блоки памяти:

- Блоки памяти — это участки памяти, которые аллокатор выделяет или освобождает.
- Каждый блок содержит информацию о своем размере (size) и указатель на следующий свободный блок (next).

3. Функции выделения и освобождения памяти:
    - Выделение: Аллокатор ищет подходящий свободный блок в списке свободных блоков.
    - Освобождение: Блок возвращается в список свободных блоков.

Операции:

1. Инициализация свободных списков:
    - Аллокатор инициализируется функцией allocator_create, которая принимает указатель на память и ее размер.
    - Внутри этой функции создается структура Allocator, которая управляет памятью.
    - Весь доступный объем памяти, за исключением размера структуры Allocator, используется для создания первого свободного блока.
    - Этот блок добавляется в список свободных блоков (free_list).
2. Выделение памяти:
    - Функция allocator_alloc выполняет выделение памяти.
    - Размер запроса округляется до ближайшего кратного FREE_LIST_ALIGNMENT (в данном случае 8 байт).
    - Аллокатор проходит по списку свободных блоков в поисках блока, размер которого достаточен для удовлетворения запроса.
    - Если подходящий блок найден, он удаляется из списка свободных блоков, и указатель на выделенную память возвращается.
3. Освобождение памяти:
    - Функция allocator_free выполняет освобождение памяти.
    - Блок, который нужно освободить, добавляется в начало списка свободных блоков.
    - Указатель на следующий блок в списке обновляется, чтобы указать на текущий освобожденный блок.

Тестирование

В ходе тестирования были проведены следующие шаги для проверки работы двух типов аллокаторов памяти: аллокатора с обычным списком свободных блоков (liballocator_firstfit.so) и аллокатора с оптимизированным алгоритмом (liballocator_mkk_alg.so). Каждый из аллокаторов был проверен на выделение и освобождение блоков памяти, а также измерено время, необходимое для этих операций.

В качестве объектов были выбраны структуры данных, так как они являются хорошим представлением для моделирования реальных данных, которые могут быть динамически выделены и освобождены в процессе работы программы. Каждый объект содержит несколько полей (целое число, строку и вещественное число), что позволяет более полно протестировать выделение памяти, включая необходимость корректной работы с различными типами данных.

1. Инициализация аллокаторов: для каждого тестируемого аллокатора была выделена память, соответствующая заявленному размеру (1 МБ). Это базируется на функции mmap, которая

выделяет память для аллокатора. Успешная инициализация проверялась путем создания аллокатора и проверки возвращаемого значения.

2. Выделение памяти: для каждого аллокатора было выделено 3 объекта типа Object с помощью соответствующих функций выделения памяти. Для каждого выделенного блока измерялось время выполнения операции выделения памяти. Результаты сохранялись и выводились на экран в формате: адрес блока, размер и время выделения.
3. Заполнение данных в объектах: после выделения памяти для каждого объекта присваивались значения его полям (id, name, value). Это позволило проверить, что выделенная память корректно используется для хранения данных.
4. Освобождение памяти: после заполнения данных каждый объект был освобожден с помощью соответствующих функций. Также выводилось соответствующее сообщение о том, что блок освобожден, и выводился его id и данные внутри блока.
5. Завершение тестирования: по завершению всех операций тестирования аллокаторы были уничтожены, и все ресурсы были освобождены.

Результаты тестирования

Тестирование показало следующие результаты для двух типов аллокаторов:

Аллокатор с обычным списком свободных блоков (liballocator_firstfit.so):

Allocated block: 0x7f547f0ce010, size: 1024 bytes, time: 0.000000110 seconds

Freed block: 0x7f547f0ce010 (id=1, name=Object 1, value=123.45), time: 0.000000110 seconds

Allocated block: 0x7f547f0ce420, size: 2048 bytes, time: 0.000000090 seconds

Freed block: 0x7f547f0ce420 (id=2, name=Object 2, value=246.90), time: 0.000000020 seconds

Allocated block: 0x7f547f0cec30, size: 3072 bytes, time: 0.000003556 seconds

Freed block: 0x7f547f0cec30 (id=3, name=Object 3, value=370.35), time: 0.000000040 seconds

Аллокатор на основе алгоритма Мак-Кьюзика-Кэрелса (liballocator_mkk_alg.so):

Allocated block: 0x7f01bd663028, size: 1024 bytes, time: 0.000000100 seconds

Freed block: 0x7f01bd663028 (id=1, name=Object 1, value=123.45), time: 0.000000060 seconds

Allocated block: 0x7f01bd663028, size: 2048 bytes, time: 0.000000060 seconds

Freed block: 0x7f01bd663028 (id=2, name=Object 2, value=246.90), time: 0.000000020 seconds

Allocated block: 0x7f01bd663028, size: 3072 bytes, time: 0.000000030 seconds

Freed block: 0x7f01bd663028 (id=3, name=Object 3, value=370.35), time: 0.000000060 seconds

Оба аллокатора эффективно справляются с выделением и освобождением памяти. Однако наблюдаются следующие различия:

Аллокатор с обычным списком свободных блоков: медленнее работает с большими блоками из-за необходимости поиска подходящего блока в свободном списке. Не использует повторно освобожденные блоки памяти, что может способствовать фрагментации

Аллокатор на основе алгоритма Мак-Кьюзика-Кэрелса: работает быстрее и стабильнее, особенно для больших блоков. Эффективно повторно использует освобожденные блоки, что уменьшает накладные расходы на выделение новой памяти.

Код программы

main.c

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

typedef struct AllocatorAPI {
    void* (*allocator_create)(void*, size_t);
    void (*allocator_destroy)(void*);
    void* (*allocator_alloc)(void*, size_t);
    void (*allocator_free)(void*, void*);
} AllocatorAPI;

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

void default_allocator_destroy(void* allocator) {
}

void* default_allocator_alloc(void* allocator, size_t size) {
    if (allocator) {
        return (void*)((char*)allocator + sizeof(size_t));
    }
    return NULL;
}

void default_allocator_free(void* allocator, void* memory) {
}

int main(int argc, char** argv) {
    AllocatorAPI api;
    void* library_handle = NULL;

    if (argc > 1) {
        library_handle = dlopen(argv[1], RTLD_LAZY);
        if (library_handle) {
            api.allocator_create = dlsym(library_handle, "allocator_create");
            api.allocator_destroy = dlsym(library_handle, "allocator_destroy");
            api.allocator_alloc = dlsym(library_handle, "allocator_alloc");
            api.allocator_free = dlsym(library_handle, "allocator_free");
        } else {
            fprintf(stderr, "Failed to load library: %s\n", dlerror());
        }
    } else {
        fprintf(stderr, "No library path provided. Using default allocator.\n");
    }
```

```
    if (!library_handle) {
        api.allocator_create = default_allocator_create;
        api.allocator_destroy = default_allocator_destroy;
        api.allocator_alloc = default_allocator_alloc;
        api.allocator_free = default_allocator_free;
    }

    size_t pool_size = 1024 * 1024;
    void* memory = mmap(NULL, pool_size, PROT_READ | PROT_WRITE, MAP_ANONYMOUS |
MAP_PRIVATE, -1, 0);
    if (memory == MAP_FAILED) {
        perror("mmap failed");
        return 1;
    }

    void* allocator = api.allocator_create(memory, pool_size);

    struct timespec start, end;
    double time_taken;

    // тестирование выделения памяти
    for (int i = 0; i < 3; i++) {
        size_t block_size = 1024 * (i + 1);  // размер блока увеличивается с каждой
итерацией

        clock_gettime(CLOCK_MONOTONIC, &start);
        void* ptr = api.allocator_alloc(allocator, block_size);
        clock_gettime(CLOCK_MONOTONIC, &end);

        time_taken = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
        printf("Allocated block: %p, size: %zu bytes, time: %.9f seconds\n", ptr,
block_size, time_taken);

        clock_gettime(CLOCK_MONOTONIC, &start);
        api.allocator_free(allocator, ptr);
        clock_gettime(CLOCK_MONOTONIC, &end);

        time_taken = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
        printf("Freed block: %p (id=%d, name=Object %d, value=%.2f), time: %.9f
seconds\n",
               ptr, i + 1, i + 1, (double)(i + 1) * 123.45, time_taken);
    }

    api.allocator_destroy(allocator);
    munmap(memory, pool_size);

    if (library_handle) {
        dlclose(library_handle);
    }

    return 0;
}
```

allocator_firstfit.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>

typedef struct Block {
    size_t size;
    struct Block* next;
} Block;
```

```
typedef struct Allocator{
    void* memory;
    size_t size;
    Block* free_list;
} Allocator;

Allocator* allocator_create(void* memory, size_t size) {
    Allocator* allocator = (Allocator*)mmap(NULL, sizeof(Allocator), PROT_READ |
PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
    allocator->memory = memory;
    allocator->size = size;
    allocator->free_list = (Block*)memory;
    allocator->free_list->size = size;
    allocator->free_list->next = NULL;
    return allocator;
}

void allocator_destroy(Allocator* allocator) {
    munmap(allocator, sizeof(Allocator));
}

void* allocator_alloc(Allocator* allocator, size_t size) {
    Block* prev = NULL;
    Block* curr = allocator->free_list;

    while (curr != NULL) {
        if (curr->size >= size) {
            if (curr->size > size + sizeof(Block)) {
                Block* new_block = (Block*)((char*)curr + sizeof(Block) + size);
                new_block->size = curr->size - size - sizeof(Block);
                new_block->next = curr->next;
                curr->size = size;
                curr->next = new_block;
            }
            if (prev == NULL) {
                allocator->free_list = curr->next;
            } else {
                prev->next = curr->next;
            }
            return (void*)((char*)curr + sizeof(Block));
        }
        prev = curr;
        curr = curr->next;
    }
    return NULL;
}

void allocator_free(Allocator* allocator, void* memory) {
    Block* block = (Block*)((char*)memory - sizeof(Block));
    block->next = allocator->free_list;
    allocator->free_list = block;
}
```

allocator_mkk_alg.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>

#define ALIGN_SIZE(size, alignment) (((size) + (alignment - 1)) & ~(alignment - 1))
#define FREE_LIST_ALIGNMENT 8

typedef struct Block {
    size_t size;
    struct Block* next;
```

```c
} Block;

typedef struct Allocator{
    void* memory;
    size_t size;
    Block* free_list;
} Allocator;

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

    Allocator* allocator = (Allocator*)memory;
    allocator->memory = (char*)memory + sizeof(Allocator);
    allocator->size = size - sizeof(Allocator);
    allocator->free_list = (Block*)allocator->memory;

    if (allocator->free_list != NULL) {
        allocator->free_list->size = allocator->size;
        allocator->free_list->next = NULL;
    }

    return allocator;
}

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

    allocator->memory = NULL;
    allocator->size = 0;
    allocator->free_list = NULL;
}

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

    size_t aligned_size = ALIGN_SIZE(size, FREE_LIST_ALIGNMENT);
    Block* prev = NULL;
    Block* curr = allocator->free_list;

    while (curr != NULL) {
        if (curr->size >= aligned_size) {
            if (prev != NULL) {
                prev->next = curr->next;
            } else {
                allocator->free_list = curr->next;
            }
            return (void*)((char*)curr + sizeof(Block));
        }

        prev = curr;
        curr = curr->next;
    }

    return NULL;
}

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

```
    Block* block = (Block*)((char*)memory - sizeof(Block));
    block->next = allocator->free_list;
    allocator->free_list = block;
}
```

Протокол работы программы

anegamelina@LAPTOP-0ED9K3JN:/mnt/c/Users/Anega/CLionProjects/osi_labs/build$ ./main ./liballocator_firstfit.so

Allocated block: 0x7fb9ce6a2010, size: 1024 bytes, time: 0.000000191 seconds

Freed block: 0x7fb9ce6a2010 (id=1, name=Object 1, value=123.45), time: 0.000000120 seconds

Allocated block: 0x7fb9ce6a2420, size: 2048 bytes, time: 0.000000190 seconds

Freed block: 0x7fb9ce6a2420 (id=2, name=Object 2, value=246.90), time: 0.000000050 seconds

Allocated block: 0x7fb9ce6a2c30, size: 3072 bytes, time: 0.000006633 seconds

Freed block: 0x7fb9ce6a2c30 (id=3, name=Object 3, value=370.35), time: 0.000000050 seconds

anegamelina@LAPTOP-0ED9K3JN:/mnt/c/Users/Anega/CLionProjects/osi_labs/build$ ./main ./liballocator_mkk_alg.so

Allocated block: 0x7fb6e7d85028, size: 1024 bytes, time: 0.000000210 seconds

Freed block: 0x7fb6e7d85028 (id=1, name=Object 1, value=123.45), time: 0.000000211 seconds

Allocated block: 0x7fb6e7d85028, size: 2048 bytes, time: 0.000000131 seconds

Freed block: 0x7fb6e7d85028 (id=2, name=Object 2, value=246.90), time: 0.000000050 seconds

Allocated block: 0x7fb6e7d85028, size: 3072 bytes, time: 0.000000040 seconds

Freed block: 0x7fb6e7d85028 (id=3, name=Object 3, value=370.35), time: 0.000000050 seconds

anegamelina@LAPTOP-0ED9K3JN:/mnt/c/Users/Anega/CLionProjects/osi_labs/build$ ./main

No library path provided. Using default allocator.

Allocated block: 0x7f172e63e008, size: 1024 bytes, time: 0.000000181 seconds

Freed block: 0x7f172e63e008 (id=1, name=Object 1, value=123.45), time: 0.000000090 seconds

Allocated block: 0x7f172e63e008, size: 2048 bytes, time: 0.000000080 seconds

Freed block: 0x7f172e63e008 (id=2, name=Object 2, value=246.90), time: 0.000000041 seconds

Allocated block: 0x7f172e63e008, size: 3072 bytes, time: 0.000000040 seconds

Freed block: 0x7f172e63e008 (id=3, name=Object 3, value=370.35), time: 0.000000050 seconds

anegamelina@LAPTOP-0ED9K3JN:/mnt/c/Users/Anega/CLionProjects/osi_labs/build$ strace ./main ./liballocator_mkk_alg.so

execve("./main", ["./main", "./liballocator_mkk_alg.so"], 0x7ffd27c18538 /* 27 vars */) = 0

brk(NULL) = 0x556864ba3000

arch_prctl(0x3001 /* ARCH_??? */, 0x7ffe755dd0a0) = -1 EINVAL (Invalid argument)

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7efdc8321000

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=19779, ...}, AT_EMPTY_PATH) = 0

mmap(NULL, 19779, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7efdc831c000

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"..., 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\0\302\211\332Pq\2439\235\350\223\322\257\201\326\243\f"..., 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) = 0x7efdc80f3000

mprotect(0x7efdc811b000, 2023424, PROT_NONE) = 0

mmap(0x7efdc811b000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7efdc811b000

mmap(0x7efdc82b0000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7efdc82b0000

mmap(0x7efdc8309000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x7efdc8309000

mmap(0x7efdc830f000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7efdc830f000

close(3)   = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7efdc80f0000

arch_prctl(ARCH_SET_FS, 0x7efdc80f0740) = 0

set_tid_address(0x7efdc80f0a10)   = 206887

set_robust_list(0x7efdc80f0a20, 24)   = 0

rseq(0x7efdc80f10e0, 0x20, 0, 0x53053053) = 0

mprotect(0x7efdc8309000, 16384, PROT_READ) = 0

mprotect(0x5568533b5000, 4096, PROT_READ) = 0

mprotect(0x7efdc835b000, 8192, PROT_READ) = 0

prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0

munmap(0x7efdc831c000, 19779)   = 0

getrandom("\x38\x84\xda\xb3\x90\x81\xcc\x27", 8, GRND_NONBLOCK) = 8

brk(NULL)   = 0x556864ba3000

brk(0x556864bc4000)   = 0x556864bc4000

openat(AT_FDCWD, "./liballocator_mkk_alg.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"..., 832) = 832

newfstatat(3, "", {st_mode=S_IFREG|0777, st_size=15280, ...}, AT_EMPTY_PATH) = 0

getcwd("/mnt/c/Users/Anega/CLionProjects/osi_labs/build", 128) = 48

mmap(NULL, 16424, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7efdc831c000

mmap(0x7efdc831d000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7efdc831d000

mmap(0x7efdc831e000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7efdc831e000

mmap(0x7efdc831f000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7efdc831f000

close(3)   = 0

mprotect(0x7efdc831f000, 4096, PROT_READ) = 0

mmap(NULL, 1048576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7efdc7ff0000

newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x2), ...}, AT_EMPTY_PATH) = 0

write(1, "Allocated block: 0x7efdc7ff0028,"..., 77Allocated block: 0x7efdc7ff0028, size: 1024 bytes, time: 0.000000300 seconds

) = 77

write(1, "Freed block: 0x7efdc7ff0028 (id="..., 91Freed block: 0x7efdc7ff0028 (id=1, name=Object 1, value=123.45), time: 0.000001092 seconds

) = 91

write(1, "Allocated block: 0x7efdc7ff0028,"..., 77Allocated block: 0x7efdc7ff0028, size: 2048 bytes, time: 0.000001763 seconds

) = 77

write(1, "Freed block: 0x7efdc7ff0028 (id="..., 91Freed block: 0x7efdc7ff0028 (id=2, name=Object 2, value=246.90), time: 0.000000932 seconds

) = 91

write(1, "Allocated block: 0x7efdc7ff0028,"..., 77Allocated block: 0x7efdc7ff0028, size: 3072 bytes, time: 0.000000962 seconds

) = 77

write(1, "Freed block: 0x7efdc7ff0028 (id="..., 91Freed block: 0x7efdc7ff0028 (id=3, name=Object 3, value=370.35), time: 0.000000942 seconds

) = 91

munmap(0x7efdc7ff0000, 1048576) = 0

munmap(0x7efdc831c000, 16424) = 0

exit_group(0) = ?

+++ exited with 0 +++

Вывод

В ходе написания данной лабораторной работы я узнала об устройстве аллокаторов. Научилась создавать, подключать и использовать динамические библиотеки. Были реализованы два алгоритма аллокации памяти, работающие через один API и подключаемые через динамические библиотеки.