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-213Б-23

Студент: Колесник Д.С.

Преподаватель: Бахарев В.Д. (ФИИТ)

Оценка: _____

Дата: 24.12.24

Москва, 2024

Постановка задачи

Вариант 6.

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). Аллокаторы – метод двойников и алгоритм блоков степени двойки.

Общий метод и алгоритм решения

Использованные системные вызовы:

- int write(int fd, void* buf, size_t count); – записывает count байт из buf в fd.
- void *mmap(void addr, size_t length, int prot, int flags, int fd, off_t offset); – выполняет отображение файла или устройства на память.
- int munmap(void addr, size_t length); – удаляет отображение файла или устройства на память.
- void* dlopen(const char* filename, int flags); – открывает динамическую библиотеку.
- void* dlsym(void* handle, const char* symbol); – позволяет получить адрес символа из загруженной библиотеки.
- int dlclose(void* handle) – закрываем динамическую библиотеку.

- load_memory_manager(const char* library_path):

Эта функция отвечает за загрузку и подготовку к использованию выбранного менеджера памяти. Она принимает путь к динамической библиотеке (.so файлу) в качестве аргумента. Если путь не предоставлен или загрузка библиотеки не удалась, load_memory_manager использует “заглушки” - функции, которые используют системный mmap для выделения памяти, таким образом обеспечивая работоспособность программы в случае отсутствия кастомной библиотеки. Если же библиотека загружена успешно, то функция получает указатели на функции create_memory_manager, allocate_memory, release_memory и destroy_memory_manager из динамической библиотеки, используя dlsym. В конце функция возвращает структуру, содержащую указатели на эти функции. Это своего рода “интерфейс”, через который программа будет взаимодействовать с менеджером памяти.

- run_memory_test(const char* library_path):

Эта функция содержит основной тест для менеджера памяти. Она сначала вызывает load_memory_manager для получения нужных указателей на функции. Затем выделяет общую область памяти при помощи mmap, которая будет использоваться менеджером памяти. После этого она вызывает create_memory_manager для инициализации менеджера памяти, передавая ему выделенную область. Далее, она вызывает allocate_memory, чтобы выделить блок памяти для теста, затем копирует в

эту память строку и выводит её. Затем вызывает release_memory, чтобы освободить выделенную область и destroy_memory_manager, чтобы очистить память, выделенную менеджером. В конце теста, она освобождает общую область памяти, выделенную через mmap, и структуру, хранящую указатели на функции, полученную от load_memory_manager.

- main(int argc, char** argv):

Это главная функция программы, точка ее входа. Она проверяет наличие аргумента командной строки (пути к динамической библиотеке) и передает его в функцию run_memory_test, которая и выполняет всю основную работу. Если run_memory_test завершится с ошибкой, main вернет код ошибки, в противном случае вернёт код успеха.

Взаимодействие функций:

- main управляет общим потоком программы и запускает тест.
- load_memory_manager отвечает за загрузку нужного менеджера памяти (либо кастомного, либо дефолтного)
- run_memory_test проводит фактическое тестирование менеджера памяти, используя функции, полученные от load_memory_manager

Что происходит в AllocatorBuddy.c:

- create_memory_manager(void* mem_area, size_t total_size):

Эта функция отвечает за инициализацию менеджера памяти. Она принимает указатель на область памяти (mem_area), выделенную через mmap, и ее размер (total_size). Внутри этой функции: * Создается структура allocator (в которой содержится указатель на начало области памяти, ее размер и указатель на верхний уровень менеджера). * Инициализируются поля allocator. * Создается начальный блок mem_chunk в начале предоставленной области памяти, устанавливая его как “доступный” и настраивая его начальные свойства, а также устанавливая его как верхний уровень менеджера (allocator->top). * Возвращается указатель на созданный аллокатор.

- allocate_memory(void* manager, size_t size):

Эта функция отвечает за выделение памяти заданного размера (size) из менеджера. Она принимает указатель на аллокатор (manager). Внутри этой функции происходит следующее: * Выполняется проверка на то, что менеджер существует и что размер выделяемой области больше 0. * Размер выделяемой области округляется до ближайшей степени двойки. * Вызывается allocate_memory_chunk, которая рекурсивно ищет подходящий блок памяти (если блок слишком большой, то происходит деление) и помечает его как “недоступный”. * Возвращается указатель на начало области, выделенной для пользователя, то есть указатель на область памяти сразу после структуры mem_chunk.

- 3. allocate_memory_chunk(allocator *allocator, mem_chunk* current, int capacity):

Эта рекурсивная функция является сердцем логики выделения памяти. Она принимает указатель на аллокатор, текущий блок (current) и требуемый размер (capacity). Она работает следующим образом: * Проверяется, является ли текущий блок доступным и имеет ли он достаточную емкость. * Если блок слишком большой, он делится на два блока меньшего размера. * Рекурсивно вызывается сама для

поиска подходящего блока. * Если подходящий блок найден, он помечается как недоступный и возвращается.

- release_memory(allocator* allocator, void* chunk_ptr):

Эта функция отвечает за освобождение ранее выделенной памяти. Она принимает указатель на аллокатор и указатель на начало области, выделенной для пользователя. Внутри этой функции происходит следующее: * Выполняется проверка на то, что менеджер существует и что chunk_ptr не равен нулю. * Вычисляется указатель на структуру mem_chunk на основе chunk_ptr. * Блок помечается как “доступный”. * Если соседние блоки также “доступны”, то они сливаются в один более крупный блок, при этом вызывается рекурсивно release_memory.

- destroy_memory_manager(allocator *allocator):

Эта функция отвечает за уничтожение менеджера памяти. Она принимает указатель на аллокатор. Внутри этой функции: * Выполняется проверка на то, что менеджер существует. * Освобождается память, выделенная через allocate_memory, а также весь массив памяти, выделенный через mmap. * Удаляется сам аллокатор.

Про AllocatorBiRange.c:

Представьте, что у вас есть структура данных BiRange, которая используется для организации свободных блоков памяти в аллокаторе. Вместо сложных структур данных, она просто описывает один блок свободной памяти.

- Функция allocate_memory(void* manager, size_t size):

Эта функция получает указатель на аллокатор (manager) и запрашиваемый размер size. Она ищет в списке BiRange первый BiRange, размер которого (size) достаточен для удовлетворения запроса. Если такой BiRange найден, он выделяется и возвращает указатель на область памяти в нем. Если такой BiRange не найден или нет свободных BiRange вообще, она возвращает NULL. Если BiRange больше, чем нужно, то он немедленно делится, и возвращается указатель на затребованный кусок памяти. Этот кусок памяти немедленно выводится из списка свободных BiRange.

- Функция release_memory(void* manager, void* memory_address):

Эта функция получает указатель на аллокатор (manager) и адрес освобождаемого блока памяти (memory_address). Она находит BiRange, к которому принадлежит memory_address. Этот BiRange помечается как свободный. Затем проверяются соседние BiRange — если они тоже свободны и имеют одинаковый размер, то эти BiRange объединяются в один больший. Если это произойдет, то функция рекурсивно проверяет снова соседние BiRange. Этот процесс продолжается до тех пор, пока не будет найдена ситуация, когда ни один из соседей не может быть объединён. Если функция не находит BiRange, она возвращается без изменений.

- Функция create_memory_manager(void* mem_area, size_t total_size):

Эта функция инициализирует менеджер памяти и создаёт один BiRange, который охватывает всю предоставленную область памяти. start указывает на начало mem_area, size на total_size, left и right равны NULL поскольку это единственный BiRange. Функция возвращает указатель на созданный аллокатор.

- Функция destroy_memory_manager(void* manager):

Функция освобождает память, выделенную для аллокатора, включая все BiRange. Она возвращает NULL, чтобы показать, что все ресурсы освобождены.

Важно: Эта модель неэффективна для больших объёмов памяти и может быть подвержена внутренней фрагментации. Реальные реализации buddy system используют более сложные структуры для более эффективного управления памятью, в отличие от этой простой концепции BiRange без явного связывания.

Код программы

main.c

```
#include <stdlib.h>
#include <stdint.h>
#include <stddef.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <dlfcn.h>



#ifndef MAP_ANON
  #ifdef MAP_ANONYMOUS
    #define MAP_ANON MAP_ANONYMOUS
  #else
    #define MAP_ANON 0x20
  #endif
#endif



typedef struct memory_manager {
    void *(*create)(void *mem_area, size_t total_size);
    void *(*allocate)(void *manager, size_t size);
    void (*release)(void *manager, void* chunk);
```

```
    void (*destroy)(void *manager);
} memory_manager;



void* stub_create_manager(void* memory_area, size_t total_size) {
    (void)memory_area;
    (void)total_size;
    return memory_area;
}



void* stub_allocate_memory(void* manager, size_t size) {
    manager = manager;
    uint32_t* memory = mmap(NULL, size + sizeof(uint32_t), PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANON, -1, 0);
    if (memory == MAP_FAILED) {
        return NULL;
    }
    *memory = (uint32_t)(size + sizeof(uint32_t));
    return memory + 1;
}


void stub_release_memory(void* manager, void* chunk){
        (void)manager;
        (void)chunk;
}


void stub_destroy_manager(void* manager){
    (void)manager;
}
```

```
memory_manager* load_memory_manager(const char* library_path) {
    if (!library_path || !library_path[0]) {
        const char msg[] = "ERROR: Do`s not set library path\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        memory_manager *manager = malloc(sizeof(memory_manager));
        if (!manager) {
            const char msg[] = "ERROR: failed to allocate memory manager\n";
            write(STDERR_FILENO, msg, sizeof(msg) - 1);
            return NULL;
        }
        manager->create = stub_create_manager;
        manager->allocate = stub_allocate_memory;
        manager->release = stub_release_memory;
        manager->destroy = stub_destroy_manager;
        return manager;
    }

    void* library = dlopen(library_path, RTLD_LOCAL | RTLD_NOW);
    if (!library) {
        const char msg[] = "ERROR: I can`t open library\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        memory_manager *manager = malloc(sizeof(memory_manager));
        if (!manager) {
            const char msg[] = "ERROR: failed to allocate memory manager\n";
            write(STDERR_FILENO, msg, sizeof(msg) - 1);
            return NULL;
        }
        manager->create = stub_create_manager;
        manager->allocate = stub_allocate_memory;
```

```
        manager->release = stub_release_memory;
        manager->destroy = stub_destroy_manager;
        return manager;
    }

    const char msg[] = "SUCCESS: I load the library\n";
    write(STDOUT_FILENO, msg, sizeof(msg) - 1);

    memory_manager *manager = malloc(sizeof(memory_manager));
    if (!manager) {
        const char msg[] = "ERROR: failed to allocate memory manager\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return NULL;
    }
    manager->create = dlsym(library, "create_memory_manager");
    manager->allocate = dlsym(library, "malloc_my_realize");
    manager->release = dlsym(library, "release_memory");
    manager->destroy = dlsym(library, "destroy_memory_manager");


    if (!manager->create || !manager->allocate || !manager->release || !manager->destroy) {
        const char msg[] = "ERROR: failed to load symbols\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        free(manager);
        dlclose(library);
        return NULL;
    }

    return manager;
}
```

```
int run_memory_test(const char* library_path) {
    memory_manager* manager_api = load_memory_manager(library_path);

    if (!manager_api) {
        return EXIT_FAILURE;
    }
    size_t test_area_size = 4096;
    void *test_area_address = mmap(NULL, test_area_size, PROT_READ | PROT_WRITE,
MAP_PRIVATE | MAP_ANON, -1, 0);
    if (test_area_address == MAP_FAILED) {
        const char msg[] = "ERROR: failed to create test area\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        free(manager_api);
        return EXIT_FAILURE;
    }

    void *manager = manager_api->create(test_area_address, test_area_size);
    if (!manager) {
        const char msg[] = "ERROR: failed to create memory manager\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        munmap(test_area_address, test_area_size);
        free(manager_api);
        return EXIT_FAILURE;
    }

    char msg[] = "SUCCESS: I created memory manager :)\n";
    write(STDOUT_FILENO, msg, sizeof(msg) - 1);
```

```
    void *allocated_chunk = manager_api->allocate(manager, 64);
    if (allocated_chunk == NULL) {
        const char msg[] = "ERROR: failed to allocate memory\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
    } else {
        const char msg[] = "SUCCESS: memory allocated\n";
        write(STDOUT_FILENO, msg, sizeof(msg) - 1);
    }


    if (allocated_chunk) {
        // const char* text = "meow\n";
        // memcpy(allocated_chunk, text, strlen(text) + 1);

    }

    strcpy(allocated_chunk, "VERY VERY LONG WORD!\n");


    write(STDOUT_FILENO, allocated_chunk, strlen(allocated_chunk));
    char address_buffer[64];
    snprintf(address_buffer, sizeof(address_buffer), "SUCCESS: Allocated memory address: %p\n",
allocated_chunk);
    write(STDOUT_FILENO, address_buffer, strlen(address_buffer));

    manager_api->release(manager, allocated_chunk);
    char free_message[] = "SUCCESS: I released memory\n";
    write(STDOUT_FILENO, free_message, sizeof(free_message) - 1);
```

```
    manager_api->destroy(manager);



    const char msg2[] = "SUCCESS: I destroyed memory manager :)\n";
    write(STDOUT_FILENO, msg2, sizeof(msg2) - 1);

    free(manager_api);
    munmap(test_area_address, test_area_size);







    return EXIT_SUCCESS;
}



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

    if (run_memory_test(library_path) != EXIT_SUCCESS) {
        return EXIT_FAILURE;
    }
      return EXIT_SUCCESS;
}
```

AllocatorBuddy.c

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/mman.h>
```

```
//для блока памяти
typedef struct mem_chunk {
    int is_available;
    int capacity;
    struct mem_chunk *left_brother;
    struct mem_chunk *right_brother;
} mem_chunk;

//для аллокатора
typedef struct allocator {
    void *area_memory;
    mem_chunk *top;
    int current_offset;
    int current_size;
} allocator;

//тут создаем новый блок памяти
mem_chunk* allocate_memory(allocator *alloc, int capacity) {

    if ((sizeof(mem_chunk) + alloc->current_offset) > alloc->current_size) {
        const char msg[] = "ERROR BUDDY: not enough memory for new chunk\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return NULL;
    }

    mem_chunk *new_chunk = (mem_chunk *)((char *)alloc->area_memory +
alloc->current_offset);
    alloc->current_offset += sizeof(mem_chunk);

    new_chunk->capacity = capacity;
```

```
    new_chunk->is_available = 1;

    new_chunk->left_brother = NULL;
    new_chunk->right_brother = NULL;

    return new_chunk;

}

int is_power_two(unsigned int n) {
    if (n <= 0) {
        return 0;
    }
    int count = 0;
    while (n > 0) {
        if (n % 2 == 1) {
            count++;
        }
        n = n / 2;
    }
    return count == 1;
}

//тут создаем наш аллокатор
allocator* create_memory_manager(void* mem_area, size_t total_size) {

    if (!is_power_two(total_size)) {
        const char msg[] = "ERROR BUDDY: total_size is not a power of 2\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return NULL;
```

```
    }

    allocator *new_allocator = (allocator *)mem_area;

    new_allocator->area_memory = (char*)mem_area + sizeof(allocator);
    new_allocator->current_offset = 0;
    new_allocator->current_size = total_size - sizeof(allocator);

    new_allocator->top = allocate_memory(new_allocator, total_size);
    if (!new_allocator->top) {
        const char msg[] = "ERROR BUDDY: failed to create top chunk\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return NULL;
    }

    return new_allocator;
}

void split(allocator *alloc, mem_chunk *chunk)
{
    int newSize = chunk->capacity / 2;

    chunk->right_brother = allocate_memory(alloc, newSize);
    chunk->left_brother = allocate_memory(alloc, newSize);



}

//тут ищем блок нужного размера
```

```
mem_chunk *allocate_memory_chunk(allocator *alloc, mem_chunk *current_chunk, int capacity)
{


   if (current_chunk == NULL || current_chunk->capacity < capacity ||
!current_chunk->is_available) {

      const char msg[] = "ERROR BUDDY: failed to allocate memory chunk\n";

      write(STDERR_FILENO, msg, sizeof(msg) - 1);

      return NULL;

   }


   if (current_chunk->capacity == capacity) {

      current_chunk->is_available = 0;

      return (void *)current_chunk;

   }


   if (current_chunk->left_brother == NULL) {

     split(alloc, current_chunk);

   }


   void* allocated = allocate_memory_chunk(alloc, current_chunk->left_brother, capacity);

   if (allocated == NULL) {

      allocated = allocate_memory_chunk(alloc, current_chunk->right_brother, capacity);

   }


   current_chunk->is_available = (current_chunk->left_brother &&
current_chunk->left_brother->is_available) ||

                              (current_chunk->right_brother &&
current_chunk->right_brother->is_available);

   return allocated;

}
```

```
void* malloc_my_realize(allocator *alloc, int capacity) {
    if ((alloc == NULL) || (capacity <= 0)) {
        const char msg[] = "ERROR BUDDY: failed to allocate memory\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return NULL;
    }

    while (!is_power_two(capacity)) {
        capacity++;
    }

    void* allocated = allocate_memory_chunk(alloc, alloc->top, capacity);
    if (!allocated) {
        const char msg[] = "ERROR BUDDY: failed to allocate memory\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return NULL;
    }
    return (char*)allocated + sizeof(mem_chunk);
}

void release_memory(allocator *alloc, void* chunk_ptr) {

    if (alloc == NULL || chunk_ptr == NULL) {
        const char msg[] = "ERROR BUDDY: failed to free memory\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return;
    }

    mem_chunk* freed_chunk = (mem_chunk*)((char*)chunk_ptr - sizeof(mem_chunk));
    if (freed_chunk == NULL) {
```

```
        const char msg[] = "ERROR BUDDY: failed to free memory\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return;
    }

    freed_chunk->is_available = 1;

    if (freed_chunk->left_brother != NULL && freed_chunk->left_brother->is_available
        && freed_chunk->right_brother->is_available) {
        release_memory(alloc, freed_chunk->left_brother);
        release_memory(alloc, freed_chunk->right_brother);
        freed_chunk->left_brother = freed_chunk->right_brother = NULL;
    }

}

void destroy_memory_manager(allocator *alloc) {
    if (!alloc) {
        const char msg[] = "ERROR: failed to destroy allocator\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return;
    }

    release_memory(alloc, alloc->top + sizeof(mem_chunk));


    if (munmap((void *)alloc, alloc->current_size + sizeof(allocator)) == 1) {
        const char msg[] = "ERROR: failed to destroy allocator\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return;
```

```
    }
}
```

AllocatorBiRange.c

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

#define COUNT 100
#define MAX 100
#define MIN 0

typedef struct mem_chunk {
    struct mem_chunk* next;
    struct mem_chunk* prev;
    int capacity;
} mem_chunk;

typedef struct {
    mem_chunk* lists_chunks[COUNT];
    void* memory;
    size_t size;
} allocator;

long long calculate_power(int base, int exponent) {
    long long result = 1;
    if (exponent < 0) {
        return -1; // Возвращаем -1, если экспонента отрицательная (не обрабатываем дробные степени)
    }
```

```
    for (int i = 0; i < exponent; ++i) {
        result *= base;
    }
    return result;
}

//создаем аллокатор
allocator* create_memory_manager(void* mem, size_t total_size) {
    allocator* manager = (allocator*)mem;
    manager->size = total_size - sizeof(allocator);
    manager->memory = (char*)mem + sizeof(allocator);

    size_t current_offset = 0;
    int list_index = 0;


    for (int i = 0; i < COUNT; ++i) {
        manager->lists_chunks[i] = NULL;
    }


    while (current_offset < manager->size) {
      int chunk_size = calculate_power(2, list_index/5);
        if (current_offset + chunk_size > manager->size) {
            break;
        }

        mem_chunk* new_chunk = (mem_chunk*)((char*)manager->memory + current_offset);
        new_chunk->capacity = chunk_size;
```

```
        if (manager->lists_chunks[list_index] == NULL) {
          new_chunk->next = NULL;
           new_chunk->prev = NULL;
        }
         else{
            new_chunk->next = manager->lists_chunks[list_index];
             manager->lists_chunks[list_index]->prev = new_chunk;
        }

        manager->lists_chunks[list_index] = new_chunk;



        current_offset += chunk_size;
        list_index++;
    }

    return manager;
}

//делим блоки
void split_memory_chunk(allocator* manager, mem_chunk* chunk) {
    int list_index = 0;
    int new_capacity = chunk->capacity / 2;



    while (calculate_power(2, list_index) < new_capacity) {
        list_index++;
    }
```

```
    mem_chunk* new_chunk = (mem_chunk*)((char*)manager->memory + new_capacity);
    new_chunk->capacity = new_capacity;



    chunk->capacity = new_capacity;



    if(manager->lists_chunks[list_index] == NULL){
        new_chunk->next = NULL;
          new_chunk->prev = NULL;
    } else{
        new_chunk->next = manager->lists_chunks[list_index];
        manager->lists_chunks[list_index]->prev = new_chunk;
    }
    manager->lists_chunks[list_index] = new_chunk;
}

void* malloc_my_realize(allocator* manager, size_t requested_size) {
   int list_index = 0;

    while (calculate_power(2, list_index) < requested_size) {
        list_index++;
    }

    if (list_index >= COUNT) {
        const char msg[] = "ERROR: failed to allocate memory\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return NULL;
    }
```

```
if (manager->lists_chunks[list_index] != NULL) {
    mem_chunk* found_chunk = manager->lists_chunks[list_index];
     manager->lists_chunks[list_index] = found_chunk->next;

    return found_chunk;
}




for (int i = list_index + 1; i < COUNT; ++i) {
    if (manager->lists_chunks[i] != NULL) {
       mem_chunk* current_chunk = manager->lists_chunks[i];

        while (i > list_index)
        {
            manager->lists_chunks[i] = current_chunk->next;
          split_memory_chunk(manager, current_chunk);
          i--;
            current_chunk = manager->lists_chunks[i];
        }

        if(manager->lists_chunks[list_index] != NULL){
         mem_chunk *block = manager->lists_chunks[list_index];
            manager->lists_chunks[list_index] = block->next;
            return block;
        }
    }
}
```

```
    return NULL;
}

void release_memory(allocator* manager, void* chunk_ptr) {
    if (!manager || !chunk_ptr) {
        return;
    }

    mem_chunk* chunk = (mem_chunk*)chunk_ptr;
     int list_index = 0;

    while (calculate_power(2, list_index) < chunk->capacity) {
        list_index++;
    }
    if(list_index >= COUNT){
        const char msg[] = "ERROR: failed to release memory\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return;
    }


     if(manager->lists_chunks[list_index] != NULL){
            chunk->next = manager->lists_chunks[list_index];
            manager->lists_chunks[list_index]->prev = chunk;
       } else {
        chunk->next = NULL;
       }
```

```
    manager->lists_chunks[list_index] = chunk;
}

void destroy_memory_manager(allocator* manager) {
     if (!manager) {
        return;
    }

    if (munmap((void *)manager, manager->size + sizeof(allocator)) != 0) {
        const char msg[] = "ERROR: failed to destroy allocator\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        exit(EXIT_FAILURE);
    }
}
```

Протокол работы программы

Тестирование:

```
~/MAI_OS/lab04/src on Kolesnik-lab04 ?2 ./main
ERROR: Do`s not set library path
SUCCESS: I created memory manager :)
SUCCESS: memory allocated
VERY VERY LONG WORD!
SUCCESS: Allocated memory address: 0x7e8e1d65f004
SUCCESS: I released memory
SUCCESS: I destroyed memory manager :)
```

```
~/MAI_OS/lab04/src on Kolesnik-lab04 ?2 ./main ./biRange.so
SUCCESS: I load the library
SUCCESS: I created memory manager :)
SUCCESS: memory allocated
VERY VERY LONG WORD!
SUCCESS: Allocated memory address: 0x7cc16a762337
SUCCESS: I released memory
SUCCESS: I destroyed memory manager :)
```

```
~/MAI_OS/lab04/src on Kolesnik-lab04 ?2 ./main ./buddy.so
SUCCESS: I load the library
SUCCESS: I created memory manager :)
SUCCESS: memory allocated
VERY VERY LONG WORD!
SUCCESS: Allocated memory address: 0x7fdfea57a150
SUCCESS: I released memory
SUCCESS: I destroyed memory manager :)
```

Strace:

execve("./main", ["./main", "./buddy.so"], 0x7fff37071990 /* 64 vars */) = 0
brk(NULL) = 0x586e91838000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffe3d328490) = -1 EINVAL (Invalid argument)
map(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7626f8b3d000
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=117967, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 117967, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7626f8b20000
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) =
0x7626f8800000
mprotect(0x7626f8828000, 2023424, PROT_NONE) = 0
mmap(0x7626f8828000, 1658880, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7626f8828000
mmap(0x7626f89bd000, 360448, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7626f89bd000
mmap(0x7626f8a16000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x7626f8a16000
mmap(0x7626f8a1c000, 52816, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7626f8a1c000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7626f8b1d000
arch_prctl(ARCH_SET_FS, 0x7626f8b1d740) = 0
set_tid_address(0x7626f8b1da10)         = 344603
set_robust_list(0x7626f8b1da20, 24)     = 0
rseq(0x7626f8b1e0e0, 0x20, 0, 0x53053053) = 0
mprotect(0x7626f8a16000, 16384, PROT_READ) = 0
mprotect(0x586e914c7000, 4096, PROT_READ) = 0
mprotect(0x7626f8b77000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) =
0
munmap(0x7626f8b20000, 117967)          = 0
getrandom("\x8f\x2b\x78\x5e\xdf\x94\x1f\x0a", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x586e91838000
brk(0x586e91859000)                     = 0x586e91859000
openat(AT_FDCWD, "./buddy.so", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832
newfstatat(3, "", {st_mode=S_IFREG|0775, st_size=16064, ...}, AT_EMPTY_PATH) = 0
getcwd("/home/ares/MAI_OS/lab04/src", 128) = 28
mmap(NULL, 16488, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7626f8b38000
mmap(0x7626f8b39000, 4096, PROT_READ|PROT_EXEC,
```

```
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7626f8b39000
mmap(0x7626f8b3a000, 4096, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7626f8b3a000
mmap(0x7626f8b3b000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7626f8b3b000
close(3)                                = 0
mprotect(0x7626f8b3b000, 4096, PROT_READ) = 0
write(1, "SUCCESS: I load the library\n", 28SUCCESS: I load the library
) = 28
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7626f8b76000
write(1, "SUCCESS: I created memory manage"..., 37SUCCESS: I created memory manager :)
) = 37
write(1, "SUCCESS: memory allocated\n", 26SUCCESS: memory allocated
) = 26
write(1, "VERY VERY LONG WORD!\n", 21VERY VERY LONG WORD!
)  = 21
write(1, "SUCCESS: Allocated memory addres"..., 50SUCCESS: Allocated memory address:
0x7626f8b76150
) = 50
write(1, "SUCCESS: I released memory\n", 27SUCCESS: I released memory
) = 27
munmap(0x7626f8b76000, 4096)            = 0
write(1, "SUCCESS: I destroyed memory mana"..., 39SUCCESS: I destroyed memory manager :)
) = 39
munmap(0x7626f8b76000, 4096)            = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

Вывод

Язык Си – невероятно мощный инструмент, с помощью которого можно создавать полноценные библиотеки и писать свои аллокаторы на любой вкус и цвет. Это просто фантастика.