Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

# Курсовой проект по курсу

# «Операционные системы»

Группа: М8О-215Б-23

Студент: Голосов Г.С.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 13.03.25

Москва, 2025

# Постановка задачи

**Вариант 18.**

Необходимо сравнить два алгоритма аллокации: блоки по 2 в степени n и алгоритм двойников по следующим характеристикам:

- Фактор использования
- Скорость выделения блоков
- Скорость освобождения блоков
- Простота использования аллокатора

Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям free и malloc (realloc, опционально). Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра. 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.

# Общий метод и алгоритм решения

Использованные системные вызовы:

- void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);– выделяет страницу памяти.
- int munmap(void *addr, size_t length); освобождения памяти, ранее отображенной с помощью mmap

Целью курсового проекта является сравнить две два аллокатора: аллокатор, использующий блоки по 2 в степени n, и аллокатор, использующий метод двойников. Рассмотрим подробнее каждый аллокатор, а потом сравним их.

Принцип работы алгоритма блоков по 2 в степени n:

- Все блоки памяти, которые могут быть выделены, имеют размеры, равные степеням двойки (например, 4, 8, 16, 32, … байт). При инициализации программы выделяется один большой непрерывный пул памяти, размер которого также является степенью двойки (например, 1024 или 4096 байт).
- При получении запроса на выделение памяти запрошенный размер округляется до ближайшей степени двойки, которая не меньше этого размера.
- Аллокатор проверяет свободные списки блоков соответствующего размера. Если в списке уже есть свободный блок нужного размера, он возвращается пользователю.
- Если свободного блока нужного размера нет, аллокатор находит свободный блок большего размера и делит его на два равных «близнеца». Один из полученных блоков может быть дополнительно разделён до нужного размера, а оставшиеся части помещаются в списки свободных блоков для соответствующего порядка.

При освобождении блока определяется его размер. Блок возвращается в свободный список своего размера.

Принцип работы метода двойников:

- все выделяемые блоки имеют размер в степень двойки, 4, 8, 16, 32, … байт;
- размер изначального единого свободного блока (при старте программы) тоже равен степени 2;
- когда требуется выделить блок размера S:
    - находим минимальную степень 2 больше S;
    - проверяем наличие свободных страниц такого размера, отдаём если есть;
    - если нет, находим минимальный блок большего размера и рекурсивно расщепляем его до нужного размера. Получившиеся “обрезки” пополняют списки свободных блоков.
- при освобождении блока:
    - находим размер блока;
    - проверяем, свободен ли второй блок (buddy) и если свободен, объединяем их и рекурсивно проверяем вышестоящий (уже объединённый) блок.

Размер соседнего блока (двойника, buddy) всегда равен размеру освобождаемого. Причем сосед только один (слева или справа, согласно алгоритму выделения), его адрес легко установить из значения указателя текущего блока и его размера.

# Код программы

## Main.cpp

```
#include "binary_allocator.h"   // Интерфейс Binary Allocator (блоки по 2^n)
#include "buddy_allocator.h"    // Интерфейс Buddy Allocator (метод двойников)
#include "shared.h"             // Общие inline-функции (closest_pow2, pow2 и т.д.)
#include <sys/mman.h>
#include <chrono>
#include <vector>
#include <thread>
#include <random>
#include <iostream>
#include <string>

// Получение текущего времени
#define NOW std::chrono::high_resolution_clock::now()

// Вспомогательные функции для вычисления суммарного объёма свободной памяти.
// Они проходят по всем спискам свободных блоков и суммируют их размеры.
size_t bin_alloc_get_free_memory(BinaryAllocator* ba) {
    size_t total = 0;
    for (uint64_t i = 0; i <= ba->max_order; i++) {
        ForwardMemory* fm = ba->free_blocks[i];
        while (fm) {
            total += pow2(i);
            fm = fm->next;
        }
    }
    return total;
}

size_t buddy_get_free_memory(BuddyAllocator* ba) {
    size_t total = 0;
    for (uint64_t i = 0; i <= ba->max_order; i++) {
        ForwardMemory* fm = ba->free_blocks[i];
        while (fm) {
            total += pow2(i);
            fm = fm->next;
        }
    }
    return total;
}

template <typename T>
void print(const T &st)
{
    std::cout << st << "\n";
}

int main()
{
    std::cout << "Creating allocators...\n";

    // Выделяем два пула памяти по 10 MB через mmap
```

```
size_t poolSize = 10 * 1024 * 1024; // 10 MB
void *mem1 = mmap(NULL, poolSize, PROT_READ | PROT_WRITE,
                  MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
void *mem2 = mmap(NULL, poolSize, PROT_READ | PROT_WRITE,
                  MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

// Вызовы bin_alloc_create() и buddy_create() выделяют свой пул памяти самостоятельно,
// поэтому здесь мы просто создаём аллокаторы с заданным размером пула.
BinaryAllocator* al_bin = bin_alloc_create(poolSize);
BuddyAllocator* al_buddy = buddy_create(poolSize);

std::cout << "Allocators created.\n\n";

// ================================
// TEST 1 – Allocate many blocks of a fixed small size
// ================================
int cycl_count = 5;
int blocks_count = 10000;
// Массив экспонент: размер блока = 1 << exp (например, exp=4 → 16 байт)
int t1_exponents[5] = {4, 5, 6, 7, 8};

for (int i = 0; i < cycl_count; i++)
{
   size_t blockSize = 1 << t1_exponents[i];
   std::cout << "TEST 1: Block size = " << blockSize
          << " bytes, Block count = " << blocks_count << "\n";
   print("Testing allocation...");

   // --- Binary Allocator ---
   std::vector<void*> ptrs_bin(blocks_count);
   auto start = NOW;
   for (int j = 0; j < blocks_count; j++)
   {
      ptrs_bin[j] = bin_alloc_allocate(al_bin, blockSize);
   }
   auto end = NOW;
   auto duration_alloc_bin = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

   // --- Buddy Allocator ---
   std::vector<void*> ptrs_buddy(blocks_count);
   start = NOW;
   for (int j = 0; j < blocks_count; j++)
   {
      ptrs_buddy[j] = buddy_allocate(al_buddy, blockSize);
   }
   end = NOW;
   auto duration_alloc_buddy = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

   // Free allocated blocks
   start = NOW;
   for (int j = 0; j < blocks_count; j++)
   {
      bin_alloc_deallocate(al_bin, ptrs_bin[j]);
   }
   end = NOW;
```

```
    auto duration_free_bin = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

    start = NOW;
    for (int j = 0; j < blocks_count; j++)
    {
       buddy_deallocate(al_buddy, ptrs_buddy[j], blockSize);
    }
    end = NOW;
    auto duration_free_buddy = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

    std::cout << "Result Allocate:\n  Binary Allocator: " << duration_alloc_bin.count() << " us\n"
              << "  Buddy Allocator:  " << duration_alloc_buddy.count() << " us\n"
              << "  Ratio (Binary/Buddy): " << float(duration_alloc_bin.count()) /
float(duration_alloc_buddy.count()) << "\n";
    std::cout << "Result Free:\n  Binary Allocator: " << duration_free_bin.count() << " us\n"
              << "  Buddy Allocator:  " << duration_free_buddy.count() << " us\n"
              << "  Ratio (Binary/Buddy): " << float(duration_free_bin.count()) /
float(duration_free_buddy.count()) << "\n\n";
  }

  // ==============================
  // TEST 2 – High Fragmentation Test
  // ==============================
  {
    print("\nTEST 2 – High Fragmentation Test");
    int frag_blocks = 10000;
    std::vector<void*> frag_bin(frag_blocks);
    for (int i = 0; i < frag_blocks; i++)
    {
       frag_bin[i] = bin_alloc_allocate(al_bin, 100);
    }
    print("Freeing even-indexed blocks in Binary Allocator...");
    for (int i = 0; i < frag_blocks; i++)
    {
       if (i % 2 == 0)
          bin_alloc_deallocate(al_bin, frag_bin[i]);
    }

    std::vector<void*> frag_buddy(frag_blocks);
    for (int i = 0; i < frag_blocks; i++)
    {
       frag_buddy[i] = buddy_allocate(al_buddy, 100);
    }
    print("Freeing even-indexed blocks in Buddy Allocator...");
    for (int i = 0; i < frag_blocks; i++)
    {
       if (i % 2 == 0)
          buddy_deallocate(al_buddy, frag_buddy[i], 100);
    }
    print("Fragmentation created.");

    std::cout << "Allocating new blocks after fragmentation...\n";
    std::vector<void*> new_bin(frag_blocks / 2);
    auto start = NOW;
    for (int j = 0; j < frag_blocks / 2; j++)
```

```
    {
        new_bin[j] = bin_alloc_allocate(al_bin, 70);
    }
    auto end = NOW;
    auto duration_alloc_bin = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

    std::vector<void*> new_buddy(frag_blocks / 2);
    start = NOW;
    for (int j = 0; j < frag_blocks / 2; j++)
    {
        new_buddy[j] = buddy_allocate(al_buddy, 80);
    }
    end = NOW;
    auto duration_alloc_buddy = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

    start = NOW;
    for (int j = 0; j < frag_blocks / 2; j++)
    {
        bin_alloc_deallocate(al_bin, new_bin[j]);
    }
    end = NOW;
    auto duration_free_bin = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

    start = NOW;
    for (int j = 0; j < frag_blocks / 2; j++)
    {
        buddy_deallocate(al_buddy, new_buddy[j], 80);
    }
    end = NOW;
    auto duration_free_buddy = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

    std::cout << "Result Allocate:\n  Binary Allocator: " << duration_alloc_bin.count() << " us\n"
              << "  Buddy Allocator:  " << duration_alloc_buddy.count() << " us\n"
              << "  Ratio: " << float(duration_alloc_bin.count()) / float(duration_alloc_buddy.count()) << "\n";
    std::cout << "Result Free:\n  Binary Allocator: " << duration_free_bin.count() << " us\n"
              << "  Buddy Allocator:  " << duration_free_buddy.count() << " us\n"
              << "  Ratio: " << float(duration_free_bin.count()) / float(duration_free_buddy.count()) << "\n\n";

    // Освобождаем оставшиеся фрагментированные блоки (нечетные индексы)
    for (int i = 0; i < frag_blocks; i++)
    {
        if (i % 2 != 0)
        {
            bin_alloc_deallocate(al_bin, frag_bin[i]);
            buddy_deallocate(al_buddy, frag_buddy[i], 100);
        }
    }
}

// ==============================
// TEST 3 – Random Block Allocation Test
// ==============================
{
    print("\nTEST 3 – Random Block Allocation");
    int num_iterations = 5;
```

```
    int num_blocks = 10000; // большое число блоков
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> size_dist(4, 1024); // размеры блоков до 1KB

    for (int iter = 0; iter < num_iterations; iter++)
    {
        std::vector<size_t> sizes(num_blocks);
        std::vector<void*> ptrs_bin(num_blocks);
        std::vector<void*> ptrs_buddy(num_blocks);
        for (int i = 0; i < num_blocks; i++)
        {
            sizes[i] = size_dist(gen);
        }

        auto start = NOW;
        for (int i = 0; i < num_blocks; i++)
        {
            ptrs_bin[i] = bin_alloc_allocate(al_bin, sizes[i]);
        }
        auto end = NOW;
        auto duration_alloc_bin = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

        start = NOW;
        for (int i = 0; i < num_blocks; i++)
        {
            ptrs_buddy[i] = buddy_allocate(al_buddy, sizes[i]);
        }
        end = NOW;
        auto duration_alloc_buddy = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

        start = NOW;
        for (int i = 0; i < num_blocks; i++)
        {
            bin_alloc_deallocate(al_bin, ptrs_bin[i]);
        }
        end = NOW;
        auto duration_free_bin = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

        start = NOW;
        for (int i = 0; i < num_blocks; i++)
        {
            buddy_deallocate(al_buddy, ptrs_buddy[i], sizes[i]);
        }
        end = NOW;
        auto duration_free_buddy = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

        std::cout << "Iteration " << iter + 1 << ":\n";
        std::cout << "Result Allocate:\n  Binary Allocator: " << duration_alloc_bin.count() << " us\n"
                << "  Buddy Allocator:  " << duration_alloc_buddy.count() << " us\n"
                << "  Ratio: " << (duration_alloc_buddy.count() > 0 ? float(duration_alloc_bin.count()) /
float(duration_alloc_buddy.count()) : 0) << "\n";
        std::cout << "Result Free:\n  Binary Allocator: " << duration_free_bin.count() << " us\n"
                << "  Buddy Allocator:  " << duration_free_buddy.count() << " us\n"
```

```
                << "  Ratio: " << (duration_free_buddy.count() > 0 ? float(duration_free_bin.count()) /
float(duration_free_buddy.count()) : 0) << "\n\n";
        }
    }

    // ================================
    // TEST 4 – Usage Factor for Binary Allocator
    // ================================
    {
        print("\nTEST 4 – Usage Factor for Binary Allocator");
        const int num_requests = 10000;
        const size_t max_req = 512;
        std::uniform_int_distribution<size_t> req_dist(16, max_req);
        std::random_device rd;
        std::mt19937 gen(rd());

        size_t total_requested = 0;
        size_t total_memory = bin_alloc_get_free_memory(al_bin) + 1; // +1 чтобы избежать деления на 0
        std::cout << "Total memory (Binary): " << total_memory << " bytes\n";
        std::vector<void*> ptrs(num_requests);
        for (int i = 0; i < num_requests; i++)
        {
            size_t req = req_dist(gen);
            ptrs[i] = bin_alloc_allocate(al_bin, req);
            total_requested += req;
        }
        size_t free_after = bin_alloc_get_free_memory(al_bin);
        double utilization = (double)total_requested / (total_memory - free_after);
        std::cout << "Total requested: " << total_requested << " bytes\n";
        std::cout << "Free memory after allocations: " << free_after << " bytes\n";
        std::cout << "Usage factor (Binary): " << utilization << "\n";

        for (auto ptr : ptrs)
        {
            bin_alloc_deallocate(al_bin, ptr);
        }
    }

    // ================================
    // TEST 5 – Usage Factor for Buddy Allocator
    // ================================
    {
        print("\nTEST 5 – Usage Factor for Buddy Allocator");
        const int num_requests = 10000;
        const size_t max_req = 512;
        std::uniform_int_distribution<size_t> req_dist(16, max_req);
        std::random_device rd;
        std::mt19937 gen(rd());

        size_t total_requested = 0;
        size_t total_memory = buddy_get_free_memory(al_buddy) + 1;
        std::cout << "Total memory (Buddy): " << total_memory << " bytes\n";
        std::vector<void*> ptrs(num_requests);
        for (int i = 0; i < num_requests; i++)
        {
```

```
            size_t req = req_dist(gen);
            ptrs[i] = buddy_allocate(al_buddy, req);
            total_requested += req;
        }
        size_t free_after = buddy_get_free_memory(al_buddy);
        double utilization = (double)total_requested / (total_memory - free_after);
        std::cout << "Total requested: " << total_requested << " bytes\n";
        std::cout << "Free memory after allocations: " << free_after << " bytes\n";
        std::cout << "Usage factor (Buddy): " << utilization << "\n";

        for (auto ptr : ptrs)
        {
            // Если точный размер не отслеживается, передаем условное значение (например, 100)
            buddy_deallocate(al_buddy, ptr, 100);
        }
    }

    // Освобождаем аллокаторы и маппинги
    bin_alloc_destroy(al_bin);
    buddy_destroy(al_buddy);
    munmap(mem1, poolSize);
    munmap(mem2, poolSize);

    return 0;
}
```

## Binary_allocator.h

```
#ifndef BINARY_ALLOCATOR_H
#define BINARY_ALLOCATOR_H

#include "shared.h"
#include <cstdint>
#include <iostream>

// Структура, описывающая отдельный блок памяти
struct Block {
    void* memory;
    uint64_t taken; // если 0 – блок свободен, иначе хранится запрошенный объём
};

// Структура аллокатора, содержащая массив блоков и массив списков свободных блоков по
порядкам
struct BinaryAllocator {
    Block* blocks;
    ForwardMemory** free_blocks; // free_blocks[i] – список блоков с ёмкостью 2^i
    uint64_t max_order;
};

// Функции создания/уничтожения аллокатора и операций выделения/освобождения
BinaryAllocator* bin_alloc_create(uint64_t byte_count);
BinaryAllocator* bin_alloc_create_with_block_size(uint64_t block_count, uint64_t block_size);
void bin_alloc_destroy(BinaryAllocator* ba);

void* bin_alloc_allocate(BinaryAllocator* ba, uint64_t byte_count);
uint64_t bin_alloc_deallocate(BinaryAllocator* ba, void* memory);
```

```
void bin_alloc_print(const BinaryAllocator& ba);

#endif
```

## Binary_allocator.cpp

```
#include "binary_allocator.h"
#include <cassert>
#include <cstdlib>
#include <cstring>
#include <iostream>

// Рекурсивное освобождение элементов списка ForwardMemory
static void recursive_free_forward_memory(ForwardMemory* fm) {
    if (!fm) return;
    recursive_free_forward_memory(fm->next);
    free(fm);
}

BinaryAllocator* bin_alloc_create(uint64_t byte_count) {
    uint64_t max_order = closest_n_pow2(byte_count);
    byte_count = pow2(max_order);

    // Выделяем общий пул памяти
    void* memory = calloc(byte_count, 1);
    assert(memory != nullptr);

    // Создаём массив блоков – один блок на каждый байт пула
    Block* blocks = (Block*) calloc(byte_count, sizeof(Block));
    assert(blocks != nullptr);
    for (uint64_t i = 0; i < byte_count; ++i) {
        blocks[i].memory = static_cast<char*>(memory) + i;
        blocks[i].taken = 0;
    }

    // Выделяем массив указателей на списки свободных блоков
    ForwardMemory** free_blocks = (ForwardMemory**) calloc(max_order + 1,
sizeof(ForwardMemory*));
    assert(free_blocks != nullptr);
    // Инициализируем список для максимального порядка – весь пул как один свободный блок
    free_blocks[max_order] = (ForwardMemory*) calloc(1, sizeof(ForwardMemory));
    assert(free_blocks[max_order] != nullptr);
    free_blocks[max_order]->memory = memory;
    free_blocks[max_order]->next = nullptr;

    BinaryAllocator* ba = (BinaryAllocator*) calloc(1, sizeof(BinaryAllocator));
    ba->blocks = blocks;
    ba->free_blocks = free_blocks;
    ba->max_order = max_order;

    return ba;
}

BinaryAllocator* bin_alloc_create_with_block_size(uint64_t block_count, uint64_t block_size) {
    return bin_alloc_create(block_count * block_size);
```

```
}

void bin_alloc_destroy(BinaryAllocator* ba) {
    if (!ba) return;
    free(ba->blocks[0].memory);
    free(ba->blocks);
    for (uint64_t i = 0; i <= ba->max_order; ++i) {
        recursive_free_forward_memory(ba->free_blocks[i]);
    }
    free(ba->free_blocks);
    free(ba);
}

// Вспомогательная функция, которая делит блок указанного порядка до уровня,
// на котором его размер удовлетворяет требуемому числу байт.
void* bin_alloc_divide_block(BinaryAllocator* ba, uint64_t order, uint64_t bytes_needed) {
    if (order == 0 || order > ba->max_order)
        return nullptr;

    void* result = nullptr;
    // Извлекаем блок из списка free_blocks[order]
    void* memory1 = ba->free_blocks[order]->memory;
    void* memory2 = static_cast<char*>(ba->free_blocks[order]->memory) + pow2(order - 1);

    // Находим индекс блока в массиве blocks
    uint64_t block_index = 0;
    uint64_t total_blocks = pow2(ba->max_order);
    for (; block_index < total_blocks; ++block_index) {
        if (ba->blocks[block_index].memory == memory1)
            break;
    }

    ForwardMemory* next = ba->free_blocks[order]->next;
    free(ba->free_blocks[order]);
    ba->free_blocks[order] = next;

    // Формируем два новых блока для списка порядка order-1
    ForwardMemory* leftmost_free = ba->free_blocks[order - 1];

    ForwardMemory* new_block2 = (ForwardMemory*) calloc(1, sizeof(ForwardMemory));
    new_block2->memory = memory2;
    new_block2->next = leftmost_free;

    ForwardMemory* new_block1 = (ForwardMemory*) calloc(1, sizeof(ForwardMemory));
    new_block1->memory = memory1;
    new_block1->next = new_block2;

    ba->free_blocks[order - 1] = new_block1;

    if (pow2(order - 1) > bytes_needed) {
        result = bin_alloc_divide_block(ba, order - 1, bytes_needed);
    } else {
        result = memory1;
        ba->blocks[block_index].taken = bytes_needed;
        // Удаляем первый элемент списка free_blocks[order - 1]
```

```
        ForwardMemory* temp = ba->free_blocks[order - 1];
        ba->free_blocks[order - 1] = new_block2;
        free(temp);
    }

    return result;
}

void* bin_alloc_allocate(BinaryAllocator* ba, uint64_t bytes_needed) {
    // Округляем запрошенный объём до ближайшей степени двойки
    bytes_needed = closest_pow2(bytes_needed);
    uint64_t order = closest_n_pow2(bytes_needed);

    ForwardMemory* current_fm = ba->free_blocks[order];
    if (current_fm != nullptr) {
        uint64_t block_index = 0;
        uint64_t total_blocks = pow2(ba->max_order);
        for (; block_index < total_blocks; ++block_index) {
            if (ba->blocks[block_index].memory == current_fm->memory)
                break;
        }
        ba->blocks[block_index].taken = bytes_needed;

        ForwardMemory* next = current_fm->next;
        void* memory = current_fm->memory;
        ba->free_blocks[order] = next;
        free(current_fm);
        return memory;
    }

    while (order <= ba->max_order && ba->free_blocks[order] == nullptr) {
        ++order;
    }
    if (order > ba->max_order)
        return nullptr;

    return bin_alloc_divide_block(ba, order, bytes_needed);
}

uint64_t bin_alloc_deallocate(BinaryAllocator* ba, void* memory) {
    uint64_t block_index = 0;
    uint64_t total_blocks = pow2(ba->max_order);
    for (; block_index < total_blocks; ++block_index) {
        if (ba->blocks[block_index].memory == memory)
            break;
    }
    uint64_t order = closest_n_pow2(ba->blocks[block_index].taken);
    uint64_t result = ba->blocks[block_index].taken;
    ba->blocks[block_index].taken = 0;

    ForwardMemory* leftmost_free = ba->free_blocks[order];
    ForwardMemory* new_block = (ForwardMemory*) calloc(1, sizeof(ForwardMemory));
    new_block->memory = memory;
    new_block->next = leftmost_free;
    ba->free_blocks[order] = new_block;
```

```
    return result;
}

void bin_alloc_print(const BinaryAllocator& ba) {
    std::cout << "BinaryAllocator = {\n";
    if (ba.max_order < 6) {
        std::cout << "\tblocks = {\n\t\t";
        uint64_t total_blocks = pow2(ba.max_order);
        for (uint64_t i = 0; i < total_blocks; ++i) {
            std::cout << ba.blocks[i].memory << ", " << ba.blocks[i].taken << "; ";
        }
        std::cout << "\n\t}\n";
    }
    std::cout << "\tfree_blocks = {\n";
    for (uint64_t i = 0; i <= ba.max_order; ++i) {
        ForwardMemory* curr = ba.free_blocks[i];
        std::cout << "\t\t";
        if (!curr) {
            std::cout << "(nil); ";
        } else {
            while (curr != nullptr) {
                std::cout << curr->memory << "; ";
                curr = curr->next;
            }
        }
        std::cout << "\n";
    }
    std::cout << "\t}\n";
    std::cout << "\tmax_order = " << ba.max_order << "\n";
    std::cout << "}\n";
}
```

## Buddy_allocator.h

```
#ifndef BUDDY_ALLOCATOR_H
#define BUDDY_ALLOCATOR_H

#include "shared.h"
#include <cstdint>
#include <cstdio>

// Структура buddy-аллокатора: содержит массив списков свободных блоков по порядкам и начальный адрес пула памяти
struct BuddyAllocator {
    ForwardMemory** free_blocks; // free_blocks[i] – список блоков с размером 2^i
    uint64_t max_order;
    void* mem_start;
};

// Функции создания/уничтожения аллокатора и операций выделения/освобождения
BuddyAllocator* buddy_create(uint64_t byte_count);
BuddyAllocator* buddy_create_with_block_size(uint64_t block_count, uint64_t block_size);
void buddy_destroy(BuddyAllocator* ba);

void* buddy_allocate(BuddyAllocator* ba, uint64_t bytes_needed);
```

```
uint64_t buddy_deallocate(BuddyAllocator* ba, void* memory, uint64_t byte_count);

void buddy_print(const BuddyAllocator& ba);

// Вспомогательная функция для рекурсивного освобождения односвязных списков
void recursive_free_forward_memory(ForwardMemory* fm);

#endif // BUDDY_ALLOCATOR_H
```

## Buddy_allocator.cpp

```
#include "buddy_allocator.h"
#include <cstdlib>
#include <cassert>
#include <iostream>
#include <cinttypes>

// Рекурсивное освобождение элементов списка ForwardMemory
void recursive_free_forward_memory(ForwardMemory* fm) {
    if (!fm) return;
    recursive_free_forward_memory(fm->next);
    free(fm);
}

BuddyAllocator* buddy_create(uint64_t byte_count) {
    // Округляем общий размер до ближайшей степени двойки и вычисляем max_order
    byte_count = closest_pow2(byte_count);
    uint64_t max_order = closest_n_pow2(byte_count);

    // Выделяем пул памяти через calloc (инициализирован нулями)
    void* memory = calloc(byte_count, 1);
    assert(memory != nullptr);

    // Выделяем массив списков свободных блоков (по порядкам от 0 до max_order)
    ForwardMemory** free_blocks = (ForwardMemory**) calloc(max_order + 1,
sizeof(ForwardMemory*));
    assert(free_blocks != nullptr);

    // Инициализируем список для максимального порядка – весь пул как один свободный блок
    free_blocks[max_order] = (ForwardMemory*) calloc(1, sizeof(ForwardMemory));
    assert(free_blocks[max_order] != nullptr);
    free_blocks[max_order]->memory = memory;
    free_blocks[max_order]->next = nullptr;

    // Создаём сам аллокатор
    BuddyAllocator* ba = (BuddyAllocator*) calloc(1, sizeof(BuddyAllocator));
    assert(ba != nullptr);

    ba->free_blocks = free_blocks;
    ba->max_order = max_order;
    ba->mem_start = memory;

    return ba;
}

BuddyAllocator* buddy_create_with_block_size(uint64_t block_count, uint64_t block_size) {
```

```
    return buddy_create(block_count * block_size);
}

void buddy_destroy(BuddyAllocator* ba) {
    for (uint64_t i = 0; i <= ba->max_order; ++i) {
        recursive_free_forward_memory(ba->free_blocks[i]);
    }
    free(ba->free_blocks);
    free(ba->mem_start);
    free(ba);
}

// Вспомогательная функция для деления блока до требуемого порядка (order_needed)
void* buddy_divide_block(BuddyAllocator* ba, uint64_t order, uint64_t order_needed) {
    void* result = nullptr;
    uint64_t shift = pow2(order - 1);

    // Удаляем блок из списка free_blocks[order]
    void* memory = ba->free_blocks[order]->memory;
    ForwardMemory* next = ba->free_blocks[order]->next;
    free(ba->free_blocks[order]);
    ba->free_blocks[order] = next;

    // Формируем правый "близнец" – смещён на shift байт
    ForwardMemory* right_buddy = (ForwardMemory*) calloc(1, sizeof(ForwardMemory));
    assert(right_buddy != nullptr);
    right_buddy->memory = static_cast<char*>(memory) + shift;
    right_buddy->next = ba->free_blocks[order - 1];
    ba->free_blocks[order - 1] = right_buddy;

    // Если достигнут нужный порядок, возвращаем левый блок; иначе – делим левый блок дальше
    if (order - 1 != order_needed) {
        ForwardMemory* left_buddy = (ForwardMemory*) calloc(1, sizeof(ForwardMemory));
        assert(left_buddy != nullptr);
        left_buddy->memory = memory;
        left_buddy->next = ba->free_blocks[order - 1];
        ba->free_blocks[order - 1] = left_buddy;

        result = buddy_divide_block(ba, order - 1, order_needed);
    } else {
        result = memory;
    }

    return result;
}

void* buddy_allocate(BuddyAllocator* ba, uint64_t bytes_needed) {
    // Вычисляем требуемый порядок для запрошенного количества байт
    uint64_t order_needed = closest_n_pow2(bytes_needed);
    uint64_t order = order_needed;
    if (order > ba->max_order) return nullptr;

    // Если в списке свободных блоков нужного порядка есть блок – возвращаем его
    ForwardMemory* current_fm = ba->free_blocks[order];
    if (current_fm != nullptr) {
```

```
        ForwardMemory* next = current_fm->next;
        void* memory = current_fm->memory;
        ba->free_blocks[order] = next;
        free(current_fm);
        return memory;
    }

    // Если блока нужного порядка нет, ищем блок большего порядка
    while (order <= ba->max_order && ba->free_blocks[order] == nullptr) {
        ++order;
    }
    if (order > ba->max_order) return nullptr;

    // Делим блок до нужного порядка
    return buddy_divide_block(ba, order, order_needed);
}

void buddy_merge(BuddyAllocator* ba, uint64_t order) {
    // Функция объединяет свободные блоки одинакового порядка в блоки более высокого порядка
    void* memory = ba->free_blocks[order]->memory;
    uint64_t byte_count = pow2(order);
    int is_right = ((static_cast<char*>(memory) - static_cast<char*>(ba->mem_start)) / byte_count) % 2;

    ForwardMemory* buddy_left = nullptr;
    ForwardMemory* buddy_right = nullptr;
    void* buddy_memory = nullptr;

    if (is_right) {
        buddy_right = ba->free_blocks[order];
        buddy_memory = static_cast<char*>(memory) - byte_count;
    } else {
        buddy_left = ba->free_blocks[order];
        buddy_memory = static_cast<char*>(memory) + byte_count;
    }

    ForwardMemory* previous = nullptr;
    ForwardMemory* current = ba->free_blocks[order]->next;

    // Поиск блока-близнеца в списке
    while (current != nullptr) {
        if (current->memory == buddy_memory) {
            if (is_right)
                buddy_left = current;
            else
                buddy_right = current;
            break;
        }
        previous = current;
        current = current->next;
    }

    if (buddy_left && buddy_right) {
        // Объединяем блоки в один блок более высокого порядка
        ForwardMemory* merged = (ForwardMemory*) calloc(1, sizeof(ForwardMemory));
        assert(merged != nullptr);
```

```
        merged->memory = buddy_left->memory;

        // Удаляем найденные блоки из списка
        if (is_right)
            ba->free_blocks[order] = buddy_right->next;
        else
            ba->free_blocks[order] = buddy_left->next;

        if (previous != nullptr) {
            if (is_right)
                previous->next = buddy_left->next;
            else
                previous->next = buddy_right->next;
        } else {
            if (is_right)
                ba->free_blocks[order] = buddy_left->next;
            else
                ba->free_blocks[order] = buddy_right->next;
        }

        free(buddy_right);
        free(buddy_left);

        // Добавляем объединённый блок в список более высокого порядка
        merged->next = ba->free_blocks[order + 1];
        ba->free_blocks[order + 1] = merged;

        buddy_merge(ba, order + 1);
    }
}

uint64_t buddy_deallocate(BuddyAllocator* ba, void* memory, uint64_t byte_count) {
    // Приводим переданное количество байт к ближайшей степени двойки и вычисляем порядок
    byte_count = closest_pow2(byte_count);
    uint64_t order = closest_n_pow2(byte_count);
    if (order >= ba->max_order) return 0;

    // Определяем, является ли данный блок правым близнецом
    int is_right = ((static_cast<char*>(memory) - static_cast<char*>(ba->mem_start)) / byte_count) % 2;

    ForwardMemory* buddy_left = nullptr;
    ForwardMemory* buddy_right = nullptr;
    void* buddy_memory = nullptr;

    if (is_right) {
        buddy_right = (ForwardMemory*) calloc(1, sizeof(ForwardMemory));
        assert(buddy_right != nullptr);
        buddy_right->memory = memory;
        buddy_right->next = nullptr;
        buddy_memory = static_cast<char*>(memory) - byte_count;
    } else {
        buddy_left = (ForwardMemory*) calloc(1, sizeof(ForwardMemory));
        assert(buddy_left != nullptr);
        buddy_left->memory = memory;
        buddy_left->next = nullptr;
```

```
        buddy_memory = static_cast<char*>(memory) + byte_count;
    }

    ForwardMemory* previous = nullptr;
    ForwardMemory* current = ba->free_blocks[order];

    // Поиск блока-близнеца в списке
    while (current != nullptr) {
        if (current->memory == buddy_memory) {
            if (is_right)
                buddy_left = current;
            else
                buddy_right = current;
            break;
        }
        previous = current;
        current = current->next;
    }

    if (buddy_left && buddy_right) {
        ForwardMemory* merged = (ForwardMemory*) calloc(1, sizeof(ForwardMemory));
        assert(merged != nullptr);
        merged->memory = buddy_left->memory;

        if (previous != nullptr) {
            if (is_right)
                previous->next = buddy_left->next;
            else
                previous->next = buddy_right->next;
        } else {
            if (is_right)
                ba->free_blocks[order] = buddy_left->next;
            else
                ba->free_blocks[order] = buddy_right->next;
        }

        free(buddy_right);
        free(buddy_left);

        merged->next = ba->free_blocks[order + 1];
        ba->free_blocks[order + 1] = merged;

        buddy_merge(ba, order + 1);
    } else {
        // Если объединение невозможно, просто добавляем блок в соответствующий список
        ForwardMemory* leftmost_fm = (is_right ? buddy_right : buddy_left);
        leftmost_fm->next = ba->free_blocks[order];
        ba->free_blocks[order] = leftmost_fm;
    }

    return byte_count;
}

void buddy_print(const BuddyAllocator& ba) {
    std::printf("BuddyAllocator = {\n");
```

```
    std::printf("\tfree_blocks = {\n");
    for (uint64_t i = 0; i <= ba.max_order; ++i) {
        ForwardMemory* fm = ba.free_blocks[i];
        std::printf("\t\t");
        if (!fm) {
            std::printf("(nil); ");
        } else {
            while (fm != nullptr) {
                std::printf("%p; ", fm->memory);
                fm = fm->next;
            }
        }
        std::printf("\n");
    }
    std::printf("\t}\n");
    std::printf("\tmax_order = %" PRIu64 "\n", ba.max_order);
    std::printf("\tmem_start = %p\n", ba.mem_start);
    std::printf("}\n");
}
```

## Shared.h

```
#ifndef SHARED_H
#define SHARED_H

#include <cstdint>

// Возвращает наименьшую степень двойки, не меньшую n
inline uint64_t closest_pow2(uint64_t n) {
    if (n == 0) return 1;
    n -= 1;
    uint64_t result = 1;
    while (n != 0) {
        result *= 2;
        n /= 2;
    }
    return result;
}

// Возвращает порядок (количество делений на 2) для n (если n – степень двойки)
inline uint64_t closest_n_pow2(uint64_t n) {
    if (n == 0) return 0;
    n -= 1;
    uint64_t result = 0;
    while (n != 0) {
        result += 1;
        n /= 2;
    }
    return result;
}

// Возвращает 2^n
inline uint64_t pow2(uint64_t n) {
    return 1ULL << n;
```

```
}

struct ForwardMemory {
    void* memory;
    ForwardMemory* next;
};

#endif // SHARED_H
```

# Протокол работы программы

**Тестирование:**

tobiklosj@LAPTOP-C3C2PI9E:~/labs_OS/CP/build$ ./main

Creating allocators...

Allocators created.

TEST 1: Block size = 4 bytes, Block count = 10000

Testing allocation...

Result Allocate:

  Binary Allocator: 677789 us

  Buddy Allocator:  616 us

  Ratio (Binary/Buddy): 1100.31

Result Free:

  Binary Allocator: 391903 us

  Buddy Allocator:  990 us

  Ratio (Binary/Buddy): 395.862

TEST 1: Block size = 8 bytes, Block count = 10000

Testing allocation...

Result Allocate:

  Binary Allocator: 2849624 us

  Buddy Allocator:  2037 us

  Ratio (Binary/Buddy): 1398.93

Result Free:

  Binary Allocator: 1872078 us

Buddy Allocator:  1626 us

Ratio (Binary/Buddy): 1151.34

TEST 1: Block size = 16 bytes, Block count = 10000

Testing allocation...

Result Allocate:

Binary Allocator: 8518461 us

Buddy Allocator:  1379 us

Ratio (Binary/Buddy): 6177.27

Result Free:

Binary Allocator: 4488543 us

Buddy Allocator:  1009 us

Ratio (Binary/Buddy): 4448.51

TEST 1: Block size = 32 bytes, Block count = 10000

Testing allocation...

Result Allocate:

Binary Allocator: 17212046 us

Buddy Allocator:  630 us

Ratio (Binary/Buddy): 27320.7

Result Free:

Binary Allocator: 10968420 us

Buddy Allocator:  1726 us

Ratio (Binary/Buddy): 6354.82

TEST 1: Block size = 64 bytes, Block count = 10000

Testing allocation...

Result Allocate:

Binary Allocator: 37426894 us

Buddy Allocator:  1141 us

Ratio (Binary/Buddy): 32801.8

Result Free:

Binary Allocator: 23136785 us

Buddy Allocator: 1848 us

Ratio (Binary/Buddy): 12519.9

TEST 2 – High Fragmentation Test

Freeing even-indexed blocks in Binary Allocator...

Freeing even-indexed blocks in Buddy Allocator...

Fragmentation created.

Allocating new blocks after fragmentation...

Result Allocate:

Binary Allocator: 30070388 us

Buddy Allocator: 155 us

Ratio: 194002

Result Free:

Binary Allocator: 23544106 us

Buddy Allocator: 47507 us

Ratio: 495.592

TEST 3A – Random Block Allocation (100 blocks)

TEST 3A Results:

Allocate - Binary Allocator: 1001563 us

Buddy Allocator: 13 us

Free - Binary Allocator: 581609 us

Buddy Allocator: 20 us

TEST 3B – Random Block Allocation (10000 blocks)

TEST 3B Results:

Allocate - Binary Allocator: 219423702 us

Buddy Allocator: 1160 us

Free - Binary Allocator: 137340681 us

Buddy Allocator: 1796 us

TEST 4 – Usage Factor for Binary Allocator

Total memory (Binary): 16777217 bytes

Total requested: 265461 bytes

Free memory after allocations: 16425376 bytes

Usage factor (Binary): 0.754491

TEST 5 – Usage Factor for Buddy Allocator

Total memory (Buddy): 16777217 bytes

Total requested: 265697 bytes

Free memory after allocations: 16424144 bytes

Usage factor (Buddy): 0.752527

## Strace:

800 execve("./main", ["./main"], 0x7ffd1060a7a8 /* 26 vars */) = 0

800 brk(NULL) = 0x55e338244000

800 arch_prctl(0x3001 /* ARCH_??? */, 0x7ffcaa9d6520) = -1 EINVAL (Invalid argument)

800 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f72c43b6000

800 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)

800 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

800 newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=37419, ...}, AT_EMPTY_PATH) = 0

800 mmap(NULL, 37419, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f72c43ac000

800 close(3) = 0

800 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libstdc++.so.6", O_RDONLY|O_CLOEXEC) = 3

800 read(3, "\177ELF\2\1\1\3\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

800 newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=2260296, ...}, AT_EMPTY_PATH) = 0

800 mmap(NULL, 2275520, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f72c4180000

800 mprotect(0x7f72c421a000, 1576960, PROT_NONE) = 0

800 mmap(0x7f72c421a000, 1118208, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x9a000) = 0x7f72c421a000

800 mmap(0x7f72c432b000, 454656, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ab000) = 0x7f72c432b000

800 mmap(0x7f72c439b000, 57344, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x21a000) = 0x7f72c439b000

800 mmap(0x7f72c43a9000, 10432, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f72c43a9000

800 close(3) = 0

800 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libgcc_s.so.1", O_RDONLY|O_CLOEXEC) = 3

800 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

800 newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=125488, ...}, AT_EMPTY_PATH) = 0

800 mmap(NULL, 127720, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f72c4160000

800 mmap(0x7f72c4163000, 94208, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x3000) = 0x7f72c4163000

800 mmap(0x7f72c417a000, 16384, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1a000) = 0x7f72c417a000

800 mmap(0x7f72c417e000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1d000) = 0x7f72c417e000

800 close(3) = 0

800 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

800 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

800 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

800 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

800 pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\315A\vq\17\17\tLh2\355\331Y1\0m"..., 68, 896) = 68

800 newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2220400, ...}, AT_EMPTY_PATH) = 0

800 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

800 mmap(NULL, 2264656, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f72c3f37000

800 mprotect(0x7f72c3f5f000, 2023424, PROT_NONE) = 0

800 mmap(0x7f72c3f5f000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f72c3f5f000

800 mmap(0x7f72c40f4000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7f72c40f4000

800 mmap(0x7f72c414d000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x7f72c414d000

800 mmap(0x7f72c4153000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f72c4153000

800 close(3) = 0

800 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libm.so.6", O_RDONLY|O_CLOEXEC) = 3

800 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0"..., 832) = 832

800 newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=940560, ...}, AT_EMPTY_PATH) = 0

800 mmap(NULL, 942344, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f72c3e50000

800 mmap(0x7f72c3e5e000, 507904, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xe000) = 0x7f72c3e5e000

800 mmap(0x7f72c3eda000, 372736, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x8a000) = 0x7f72c3eda000

800 mmap(0x7f72c3f35000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xe4000) = 0x7f72c3f35000

800 close(3) = 0

800 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f72c3e4e000

800 arch_prctl(ARCH_SET_FS, 0x7f72c3e4f3c0) = 0

800 set_tid_address(0x7f72c3e4f690) = 800

800 set_robust_list(0x7f72c3e4f6a0, 24) = 0

800 rseq(0x7f72c3e4fd60, 0x20, 0, 0x53053053) = 0

800 mprotect(0x7f72c414d000, 16384, PROT_READ) = 0

800 mprotect(0x7f72c3f35000, 4096, PROT_READ) = 0

800 mprotect(0x7f72c417e000, 4096, PROT_READ) = 0

800 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f72c3e4c000

800 mprotect(0x7f72c439b000, 45056, PROT_READ) = 0

800 mprotect(0x55e31dc3e000, 4096, PROT_READ) = 0

800 mprotect(0x7f72c43f0000, 8192, PROT_READ) = 0

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

800 munmap(0x7f72c43ac000, 37419) = 0

800 getrandom("\x25\x36\xd3\xe5\xb4\xc6\x17\x4e", 8, GRND_NONBLOCK) = 8

800 brk(NULL) = 0x55e338244000

800 brk(0x55e338265000) = 0x55e338265000

800 futex(0x7f72c43a977c, FUTEX_WAKE_PRIVATE, 2147483647) = 0

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

800 write(1, "Creating allocators...\n", 23) = 23

800 mmap(NULL, 10485760, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f72c344c000

800 mmap(NULL, 10485760, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f72c2a4c000

800 mmap(NULL, 16781312, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f72c1a4b000

800 mmap(NULL, 268439552, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f72b1a4a000

800 mmap(NULL, 16781312, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f72b0a49000

800 write(1, "Allocators created.\n\n", 21) = 21

800 write(1, "TEST 1: Block size = 4 bytes, Bl"..., 50) = 50

800 write(1, "Testing allocation...\n", 22) = 22

800 brk(0x55e33828a000) = 0x55e33828a000

800 brk(0x55e3382ab000) = 0x55e3382ab000

800 brk(0x55e3382cc000) = 0x55e3382cc000

800 write(1, "Result Allocate:\n", 17) = 17

800 write(1, " Binary Allocator: 654782 us\n", 30) = 30

800 write(1, " Buddy Allocator: 1025 us\n", 28) = 28

800 write(1, " Ratio (Binary/Buddy): 638.812\n", 32) = 32

800 write(1, "Result Free:\n", 13) = 13

800 write(1, " Binary Allocator: 364961 us\n", 30) = 30

800 write(1, " Buddy Allocator: 1149 us\n", 28) = 28

800 write(1, " Ratio (Binary/Buddy): 317.634\n"..., 33) = 33

800 write(1, "TEST 1: Block size = 8 bytes, Bl"..., 50) = 50

800 write(1, "Testing allocation...\n", 22) = 22

800 brk(0x55e3382ed000) = 0x55e3382ed000

800 brk(0x55e33830e000) = 0x55e33830e000

800 brk(0x55e33832f000) = 0x55e33832f000

800 write(1, "Result Allocate:\n", 17) = 17

800 write(1, " Binary Allocator: 2486145 us\n", 31) = 31

800 write(1, " Buddy Allocator: 618 us\n", 27) = 27

```
800   write(1, "  Ratio (Binary/Buddy): 4022.89\n", 32) = 32
800   write(1, "Result Free:\n", 13)    = 13
800   write(1, "  Binary Allocator: 1847172 us\n", 31) = 31
800   write(1, "  Buddy Allocator:  951 us\n", 27) = 27
800   write(1, "  Ratio (Binary/Buddy): 1942.35\n"..., 33) = 33
800   write(1, "TEST 1: Block size = 16 bytes, B"..., 51) = 51
800   write(1, "Testing allocation...\n", 22) = 22
800   brk(0x55e338350000)               = 0x55e338350000
800   brk(0x55e338371000)               = 0x55e338371000
800   write(1, "Result Allocate:\n", 17) = 17
800   write(1, "  Binary Allocator: 9550572 us\n", 31) = 31
800   write(1, "  Buddy Allocator:  1406 us\n", 28) = 28
800   write(1, "  Ratio (Binary/Buddy): 6792.73\n", 32) = 32
800   write(1, "Result Free:\n", 13)    = 13
800   write(1, "  Binary Allocator: 5057525 us\n", 31) = 31
800   write(1, "  Buddy Allocator:  979 us\n", 27) = 27
800   write(1, "  Ratio (Binary/Buddy): 5166.01\n"..., 33) = 33
800   write(1, "TEST 1: Block size = 32 bytes, B"..., 51) = 51
800   write(1, "Testing allocation...\n", 22) = 22
800   brk(0x55e338392000)               = 0x55e338392000
800   brk(0x55e3383b3000)               = 0x55e3383b3000
800   brk(0x55e3383d4000)               = 0x55e3383d4000
800   write(1, "Result Allocate:\n", 17) = 17
800   write(1, "  Binary Allocator: 19135324 us\n", 32) = 32
800   write(1, "  Buddy Allocator:  1379 us\n", 28) = 28
800   write(1, "  Ratio (Binary/Buddy): 13876.2\n", 32) = 32
800   write(1, "Result Free:\n", 13)    = 13
800   write(1, "  Binary Allocator: 12550143 us\n", 32) = 32
800   write(1, "  Buddy Allocator:  1816 us\n", 28) = 28
800   write(1, "  Ratio (Binary/Buddy): 6910.87\n"..., 33) = 33
800   write(1, "TEST 1: Block size = 64 bytes, B"..., 51) = 51
800   write(1, "Testing allocation...\n", 22) = 22
```

```
800   brk(0x55e3383f5000)               = 0x55e3383f5000
800   brk(0x55e338416000)               = 0x55e338416000
800   write(1, "Result Allocate:\n", 17) = 17
800   write(1, "  Binary Allocator: 39302536 us\n", 32) = 32
800   write(1, "  Buddy Allocator:  653 us\n", 27) = 27
800   write(1, "  Ratio (Binary/Buddy): 60187.7\n", 32) = 32
800   write(1, "Result Free:\n", 13)    = 13
800   write(1, "  Binary Allocator: 21540186 us\n", 32) = 32
800   write(1, "  Buddy Allocator:  1054 us\n", 28) = 28
800   write(1, "  Ratio (Binary/Buddy): 20436.6\n"..., 33) = 33
800   munmap(0x7f72c1a4b000, 16781312)  = 0
800   munmap(0x7f72b1a4a000, 268439552) = 0
800   munmap(0x7f72b0a49000, 16781312)  = 0
800   munmap(0x7f72c344c000, 10485760)  = 0
800   munmap(0x7f72c2a4c000, 10485760)  = 0
800   exit_group(0)                     = ?
800   +++ exited with 0 +++
```

# Результат сравнения

Для оценки эффективности аллокаторов были выбраны тесты, которые моделируют реальные сценарии использования памяти. Они включают в себя различные паттерны выделения и освобождения памяти, такие как частое выделение и освобождение небольших блоков, выделение и освобождение больших блоков, а также смешанные сценарии. Такой подход позволяет получить представление о производительности аллокаторов в различных условиях и выявить их сильные и слабые стороны.

### Фактор использования

Фактор использования является критически важным параметром, так как он определяет, насколько эффективно используется память. Высокий фактор использования означает, что большая часть выделенной памяти действительно используется, а не простаивает. Проверку будем проводить путем замера количества памяти до выделения блоков и после. Для того чтобы тест был максимально приближен к реальному размеры блоков будем выбирать произвольный из диапазона.

| Allocator | Binary allocator<br><br>bytes | Buddy allocator<br><br>bytes |
|---|---|---|

| | | |
|---|---|---|
| Total memory | 16777217 | 16777217 |
| Total requested | 265461 | 265697 |
| Free memory after allocations | 16425376 | 16424144 |
| Usage factor | 0.754491 | 0.752527 |

У обоих аллокаторов получился одинаковый фактор использования, равный 75%

**Скорость выделения и освобождения блоков**

Скорость выделения и освобождния блоков памяти является также ключевым показателем производительности аллокатора. Эффективный аллокатор должен минимизировать время, необходимое для выделения памяти, чтобы обеспечить быстрое выполнение программ.

1. Выделение большого числа блоков одинакового размера, идущих подряд. Число блоков 10000

| Block size/Allocator | Binary allocator<br>us | Buddy allocator<br>us |
|---|---|---|
| 4 | 677789 | 616 |
| 8 | 2849624 | 2037 |
| 16 | 8518461 | 1379 |
| 32 | 17212046 | 630 |
| 64 | 37426894 | 1141 |

2. Освобождение большого числа блоков одинакового размера, идущих подряд.

| Block size/Allocator | Binary allocator<br>us | Buddy allocator<br>us |
|---|---|---|
| 4 | 391903 | 990 |
| 8 | 1872078 | 1626 |
| 16 | 4488543 | 1009 |
| 32 | 17212046 | 630 |
| 64 | 23136785 | 1848 |

Buddy-allocator справляется с выделением большого числа блоков и их освобождением на 3-4 порядка быстрее аллокатора, использующего блоки по 2 в степени n.

3. Аллокация большого числа блоков в условиях высокой фрагментации памяти

Очень важно, чтобы аллокатор быстро работал в условиях высокой фрагментации памяти. Для имитации такой ситуации в начале выделим блоки подряд, затем удалим половину блоков.

Выделяется 10 000 блоков, каждый размером 100 байт. Затем запрашиваются блоки размером 70 байт.

| Allocator | Binary allocator<br>us | Buddy allocator<br>us |
|---|---|---|
| Time allocation | 30070388 | 155 |
| Time free | 23544106 | 47507 |

При высокой фрагментации, когда половина блоков освобождается, а оставшиеся остаются занятыми, бинарный аллокатор явно сталкивается с проблемами поиска подходящего свободного блока и управления фрагментированной памятью, что приводит к огромным накладным расходам. В свою очередь, buddy-аллокатор, благодаря механизму деления и последующего слияния блоков, эффективно справляется даже в условиях фрагментации, обеспечивая быструю работу как при выделении, так и при освобождении памяти.

4. Аллокация блоков случайного размера.

| Block`s count/Allocator | Binary allocator<br>us | Buddy allocator<br>us |
|---|---|---|
| 100 | 1001563 | 13 |
| 10000 | 219423702 | 1160 |

Очень важно, чтобы аллокатор мог быстро выделять память случайного размера, так как при использовании аллокатора в операционной системе, данное действие будет выполняться постоянно.

5. Освобождение блоков случайного размера.

| Block`s count/Allocator | First-fit allocator<br>us | Buddy allocator<br>us |
|---|---|---|
| 100 | 581609 | 20 |
| 10000 | 137340681 | 1796 |

Теперь рассмотрим освобождения блоков случайного размера. Buddy-аллокатор снова намного быстрее.

**Простота использования аллокатора**

Аллокатор с использованием блоков по 2 в степени n является одним из самых простых аллокаторов. Работает достаточно медленно. Buddy-аллокатор является улучшением предыдущего за счет того, что после освобождения блока происходит склейка блоков равных размеров. Это ускоряет его работы, но и усложняет сам алгоритм.

# Вывод

В курсовом проекте мной были рассмотрены два метода аллокации памяти: с использованием блоков размеров 2 в степени n и алгоритм двойников. Были написаны программы, реализующие оба алгоритма и проведено всестороннее тестирование для определения слабых и сильных сторон алгоритмов. По результатам тестов лучшим выбором для большинства задач является аллокатор, использующий метод двойников. Благодаря своей архитектуре, метод близнецов обеспечивает очень высокую скорость динамического выделения и освобождения памяти, и используется в составе многих современных операционных систем для динамического распределения памяти в ядре системы, драйверах или в других ответственных компонентах системы, критичных к скорости работы.