Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

**Курсовая работа по курсу**

**«Операционные системы»**

Группа: М8О-203Б-23

Студент: Снетков Н.С.

Преподаватель: Миронов Е.С.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 28.12.24

Москва, 2024

**Постановка задачи**

Необходимо реализовать два алгоритма аллокации памяти и сравнить их 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. Сравнить требуется алгоритм двойников и алгоритм свободных блоков.

**Описание каждого из исследуемых алгоритмов**

* **Блоки памяти размером 2n.** Алгоритм аллокации памяти, основанный на блоках размером 2n, работает следующим образом. Для запроса памяти вычисляется минимальный размер блока 2n, где nn — двоичный логарифм запрашиваемого размера, округленный вверх. Аллокатор ищет свободный блок нужного размера, и если находит, резервирует его и возвращает указатель. Если блок не найден, поиск продолжается на уровнях выше (блоки 2n+1, 2n+2 и т.д.), и найденный блок большего размера делится пополам до получения блока нужного размера, один блок резервируется, второй остается свободным. Если подходящий блок отсутствует, аллоцируется новая страница памяти, и процесс разделения повторяется. При освобождении блока он помечается как свободный, и если его "двойник" (соседний блок того же размера) также свободен, они объединяются в один блок большего размера. Процесс объединения продолжается вверх по уровням, пока не встретится занятый блок или не будет достигнут размер страницы.
* **Алгоритм свободных блоков.** При аллокации выбирается первый подходящий блок, размер которого больше или равен нужному размеру. Тогда начало свободного блока сдвигается и уменьшается его размер, если блок не найден, аллоцируется новая страница и выделение происходит от него. При деаллокации мы освобождаем блок, и пробуем объединить его с правым и левым соседом. Если объединения не произошло мы добавляем его в массив свободных блоков и сортируем массив.

**Процесс тестирования**

Тестируем два аллокатора (ListAllocator и BuddyAllocator). Сначала проверяем, как быстро они создаются. Потом тестируем, как они работают при нехватке памяти, выделении и освобождении блоков разного размера. Также проверяем, могут ли они повторно использовать освобожденные блоки. В конце сравниваем их производительность и количество ошибок. Все результаты выводятся в консоль.

**Код программы**

**Allocator.h**

#ifndef ALLOCATOR\_H

#define ALLOCATOR\_H

#include <cstddef>

// Базовая структура аллокатора

struct Allocator {

void\* memory; // Указатель на начало памяти

size\_t memory\_size; // Размер доступной памяти

size\_t used\_memory; // Используемая память

};

// Интерфейс аллокатора

Allocator\* createMemoryAllocator(void\* realMemory, size\_t memory\_size);

void\* alloc(Allocator\* allocator, size\_t block\_size);

void freeBlock(Allocator\* allocator, void\* block);

#endif // ALLOCATOR\_H

**BuddyAllocator.h**

#ifndef BUDDYALLOCATOR\_H

#define BUDDYALLOCATOR\_H

#include "Allocator.h"

namespace BuddyAllocator {

// Объявляем структуру BlockHeader

struct BlockHeader {

size\_t size; // Размер блока

bool is\_free; // Флаг занятости

BlockHeader\* next; // Указатель на следующий блок

};

// Наследуемся от основной структуры Allocator

struct Allocator : public ::Allocator {

BlockHeader\* free\_lists[32]; // Списки свободных блоков для каждого размера (2^5 ... 2^31)

};

Allocator\* createMemoryAllocator(void\* realMemory, size\_t memory\_size);

void\* alloc(Allocator\* allocator, size\_t block\_size);

void freeBlock(Allocator\* allocator, void\* block);

} // namespace BuddyAllocator

#endif // BUDDYALLOCATOR\_H

**ListAllocator.h**

#ifndef LISTALLOCATOR\_H

#define LISTALLOCATOR\_H

#include "Allocator.h"

namespace ListAllocator {

// Наследуемся от основной структуры Allocator

struct Allocator : public ::Allocator {};

Allocator\* createMemoryAllocator(void\* realMemory, size\_t memory\_size);

void\* alloc(Allocator\* allocator, size\_t block\_size);

void freeBlock(Allocator\* allocator, void\* block);

} // namespace ListAllocator

#endif // LISTALLOCATOR\_H

**BuddyAllocator.cpp**

#include "../include/BuddyAllocator.h"

#include <iostream>

#include <cmath>

namespace BuddyAllocator {

Allocator\* createMemoryAllocator(void\* realMemory, size\_t memory\_size) {

// Приводим realMemory к типу Allocator\*

Allocator\* allocator = static\_cast<Allocator\*>(realMemory);

// Вычисляем начало области памяти для блоков

allocator->memory = static\_cast<char\*>(realMemory) + sizeof(Allocator);

allocator->memory\_size = memory\_size - sizeof(Allocator);

allocator->used\_memory = 0;

// Инициализация списков свободных блоков

for (size\_t i = 0; i < 32; ++i) {

allocator->free\_lists[i] = nullptr;

}

// Инициализация первого блока

BlockHeader\* initial\_block = static\_cast<BlockHeader\*>(allocator->memory);

initial\_block->size = allocator->memory\_size;

initial\_block->is\_free = true;

initial\_block->next = nullptr;

// Добавляем первый блок в соответствующий список

size\_t index = static\_cast<size\_t>(std::log2(initial\_block->size));

initial\_block->next = allocator->free\_lists[index];

allocator->free\_lists[index] = initial\_block;

return allocator;

}

void\* alloc(Allocator\* allocator, size\_t block\_size) {

if (block\_size == 0) return nullptr;

// Выравнивание размера запроса до степени двойки

size\_t aligned\_size = 1;

while (aligned\_size < block\_size) {

aligned\_size <<= 1;

}

size\_t index = static\_cast<size\_t>(std::log2(aligned\_size));

// Ищем свободный блок в соответствующем списке

if (allocator->free\_lists[index]) {

BlockHeader\* block = allocator->free\_lists[index];

block->is\_free = false;

allocator->free\_lists[index] = block->next; // Убираем блок из списка

allocator->used\_memory += aligned\_size;

return reinterpret\_cast<void\*>(block + 1);

}

// Если подходящий блок не найден, ищем блок большего размера и разбиваем его

for (size\_t i = index + 1; i < 32; ++i) {

if (allocator->free\_lists[i]) {

BlockHeader\* block = allocator->free\_lists[i];

allocator->free\_lists[i] = block->next; // Убираем блок из списка

// Разбиваем блок на два

size\_t current\_size = 1 << i;

while (current\_size > aligned\_size) {

current\_size >>= 1;

BlockHeader\* buddy = reinterpret\_cast<BlockHeader\*>(

reinterpret\_cast<char\*>(block) + current\_size

);

buddy->size = current\_size;

buddy->is\_free = true;

size\_t buddy\_index = static\_cast<size\_t>(std::log2(current\_size));

buddy->next = allocator->free\_lists[buddy\_index];

allocator->free\_lists[buddy\_index] = buddy;

}

block->size = aligned\_size;

block->is\_free = false;

allocator->used\_memory += aligned\_size;

return reinterpret\_cast<void\*>(block + 1);

}

}

return nullptr; // Нет подходящих блоков

}

void freeBlock(Allocator\* allocator, void\* block) {

if (!block) return;

BlockHeader\* header = reinterpret\_cast<BlockHeader\*>(block) - 1;

header->is\_free = true;

allocator->used\_memory -= header->size;

size\_t index = static\_cast<size\_t>(std::log2(header->size));

// Слияние с соседними свободными блоками (бадди)

while (true) {

size\_t buddy\_address = reinterpret\_cast<size\_t>(header) ^ (1 << index);

BlockHeader\* buddy = reinterpret\_cast<BlockHeader\*>(buddy\_address);

bool is\_buddy\_free = false;

BlockHeader\* prev = nullptr;

BlockHeader\* curr = allocator->free\_lists[index];

while (curr) {

if (curr == buddy) {

is\_buddy\_free = true;

if (prev) {

prev->next = curr->next;

} else {

allocator->free\_lists[index] = curr->next;

}

break;

}

prev = curr;

curr = curr->next;

}

if (!is\_buddy\_free) break;

// Сливаем блоки

if (header > buddy) {

std::swap(header, buddy);

}

header->size \*= 2;

index++;

}

// Добавляем блок в соответствующий список

header->next = allocator->free\_lists[index];

allocator->free\_lists[index] = header;

}

} // namespace BuddyAllocator

**ListAllocator.cpp**

#include "../include/ListAllocator.h"

#include <iostream>

namespace ListAllocator {

struct BlockHeader {

size\_t size; // Размер блока

bool is\_free; // Флаг занятости

BlockHeader\* next; // Указатель на следующий блок

};

Allocator\* createMemoryAllocator(void\* realMemory, size\_t memory\_size) {

Allocator\* allocator = static\_cast<Allocator\*>(realMemory);

allocator->memory = static\_cast<char\*>(realMemory) + sizeof(Allocator);

allocator->memory\_size = memory\_size - sizeof(Allocator);

allocator->used\_memory = 0; // Инициализация used\_memory

// Инициализация первого блока

BlockHeader\* initial\_block = static\_cast<BlockHeader\*>(allocator->memory);

initial\_block->size = allocator->memory\_size;

initial\_block->is\_free = true;

initial\_block->next = nullptr;

return allocator;

}

void\* alloc(Allocator\* allocator, size\_t block\_size) {

BlockHeader\* current = static\_cast<BlockHeader\*>(allocator->memory);

while (current) {

if (current->is\_free && current->size >= block\_size) {

// Если блок свободен и его размер достаточен

if (current->size >= block\_size + sizeof(BlockHeader) + 1) {

// Разбиваем блок, если он слишком большой

BlockHeader\* next\_block = reinterpret\_cast<BlockHeader\*>(

reinterpret\_cast<char\*>(current) + sizeof(BlockHeader) + block\_size

);

next\_block->size = current->size - block\_size - sizeof(BlockHeader);

next\_block->is\_free = true;

next\_block->next = current->next;

// Обновляем текущий блок

current->size = block\_size;

current->is\_free = false;

current->next = next\_block;

} else {

// Если блок идеально подходит, просто выделяем его

current->is\_free = false;

}

// Обновляем used\_memory только на размер выделенного блока

allocator->used\_memory += current->size;

return reinterpret\_cast<void\*>(current + 1); // Указатель на начало данных

}

current = current->next;

}

// Если не нашли подходящий блок

return nullptr;

}

void freeBlock(Allocator\* allocator, void\* block) {

if (!block) return;

BlockHeader\* header = reinterpret\_cast<BlockHeader\*>(block) - 1;

header->is\_free = true;

// Обновляем used\_memory только на размер освобожденного блока

allocator->used\_memory -= header->size;

// Слияние соседних свободных блоков

BlockHeader\* current = static\_cast<BlockHeader\*>(allocator->memory);

while (current) {

if (current->is\_free && current->next && current->next->is\_free) {

// Сливаем блоки

current->size += sizeof(BlockHeader) + current->next->size;

current->next = current->next->next;

} else {

current = current->next;

}

}

}

} // namespace ListAllocator

**CP\_tests.cpp**

#include "../CP/include/Allocator.h"

#include "../CP/include/BuddyAllocator.h"

#include "../CP/include/ListAllocator.h"

#include <gtest/gtest.h>

#include <cstdlib>

#include <vector>

#include <chrono>

#include <iostream>

#include <cstring>

// Размер тестовой памяти

const size\_t TEST\_MEMORY\_SIZE = 3 \* 1024 \* 1024; // 1 MB

// Вспомогательная функция для измерения времени

template<typename Func>

double measureTime(Func func) {

auto start = std::chrono::high\_resolution\_clock::now();

func();

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> duration = end - start;

return duration.count();

}

// Вспомогательная функция для измерения использования памяти (в байтах)

size\_t getHeapUsage(const Allocator\* allocator) {

return allocator->used\_memory;

}

// Тест создания аллокаторов

TEST(AllocatorComparisonTest, CreationPerformance) {

char listMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для ListAllocator

char buddyMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для BuddyAllocator

double listTime = measureTime([&]() {

ListAllocator::Allocator\* listAllocator = ListAllocator::createMemoryAllocator(listMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(listAllocator, nullptr) << "ListAllocator creation failed.";

std::cout << "Memory used by ListAllocator: " << getHeapUsage(listAllocator) << " bytes." << std::endl;

});

double buddyTime = measureTime([&]() {

BuddyAllocator::Allocator\* buddyAllocator = BuddyAllocator::createMemoryAllocator(buddyMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(buddyAllocator, nullptr) << "BuddyAllocator creation failed.";

std::cout << "Memory used by BuddyAllocator: " << getHeapUsage(buddyAllocator) << " bytes." << std::endl;

});

std::cout << "ListAllocator creation time: " << listTime << " seconds." << std::endl;

std::cout << "BuddyAllocator creation time: " << buddyTime << " seconds." << std::endl;

}

// Тест для проверки поведения при нехватке памяти

TEST(AllocatorComparisonTest, OutOfMemory) {

char listMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для ListAllocator

char buddyMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для BuddyAllocator

ListAllocator::Allocator\* listAllocator = ListAllocator::createMemoryAllocator(listMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(listAllocator, nullptr) << "ListAllocator creation failed.";

BuddyAllocator::Allocator\* buddyAllocator = BuddyAllocator::createMemoryAllocator(buddyMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(buddyAllocator, nullptr) << "BuddyAllocator creation failed.";

size\_t blockSize = 64;

size\_t numBlocks = TEST\_MEMORY\_SIZE / blockSize;

// Заполняем всю доступную память

for (size\_t i = 0; i < numBlocks; ++i) {

void\* listBlock = ListAllocator::alloc(listAllocator, blockSize);

ASSERT\_NE(listBlock, nullptr) << "ListAllocator allocation failed at block " << i;

void\* buddyBlock = BuddyAllocator::alloc(buddyAllocator, blockSize);

if (buddyBlock == nullptr) {

std::cout << "BuddyAllocator out of memory at block " << i << std::endl;

break;

}

ASSERT\_NE(buddyBlock, nullptr) << "BuddyAllocator allocation failed at block " << i;

}

}

// Тест выделения и освобождения

TEST(AllocatorComparisonTest, AllocationAndFreePerformance) {

char listMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для ListAllocator

char buddyMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для BuddyAllocator

ListAllocator::Allocator\* listAllocator = ListAllocator::createMemoryAllocator(listMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(listAllocator, nullptr) << "ListAllocator creation failed.";

BuddyAllocator::Allocator\* buddyAllocator = BuddyAllocator::createMemoryAllocator(buddyMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(buddyAllocator, nullptr) << "BuddyAllocator creation failed.";

const size\_t numBlocks = 1000;

const size\_t blockSize = 64;

double listAllocTime = measureTime([&]() {

for (size\_t i = 0; i < numBlocks; ++i) {

void\* block = ListAllocator::alloc(listAllocator, blockSize);

ASSERT\_NE(block, nullptr) << "ListAllocator allocation failed at block " << i;

ListAllocator::freeBlock(listAllocator, block);

}

});

double buddyAllocTime = measureTime([&]() {

for (size\_t i = 0; i < numBlocks; ++i) {

void\* block = BuddyAllocator::alloc(buddyAllocator, blockSize);

ASSERT\_NE(block, nullptr) << "BuddyAllocator allocation failed at block " << i;

BuddyAllocator::freeBlock(buddyAllocator, block);

}

});

std::cout << "ListAllocator allocation/free time: " << listAllocTime << " seconds." << std::endl;

std::cout << "BuddyAllocator allocation/free time: " << buddyAllocTime << " seconds." << std::endl;

}

// Тест для проверки выделения и освобождения блоков разного размера

TEST(AllocatorComparisonTest, MixedBlockSizes) {

char listMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для ListAllocator

char buddyMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для BuddyAllocator

ListAllocator::Allocator\* listAllocator = ListAllocator::createMemoryAllocator(listMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(listAllocator, nullptr) << "ListAllocator creation failed.";

BuddyAllocator::Allocator\* buddyAllocator = BuddyAllocator::createMemoryAllocator(buddyMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(buddyAllocator, nullptr) << "BuddyAllocator creation failed.";

std::vector<size\_t> blockSizes = {32, 64, 128, 256, 512, 1024};

for (size\_t size : blockSizes) {

void\* listBlock = ListAllocator::alloc(listAllocator, size);

ASSERT\_NE(listBlock, nullptr) << "ListAllocator allocation failed for size " << size;

ListAllocator::freeBlock(listAllocator, listBlock);

void\* buddyBlock = BuddyAllocator::alloc(buddyAllocator, size);

ASSERT\_NE(buddyBlock, nullptr) << "BuddyAllocator allocation failed for size " << size;

BuddyAllocator::freeBlock(buddyAllocator, buddyBlock);

}

}

// Тест для проверки повторного использования освобожденных блоков

TEST(AllocatorComparisonTest, ReuseFreedBlocks) {

char listMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для ListAllocator

char buddyMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для BuddyAllocator

ListAllocator::Allocator\* listAllocator = ListAllocator::createMemoryAllocator(listMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(listAllocator, nullptr) << "ListAllocator creation failed.";

BuddyAllocator::Allocator\* buddyAllocator = BuddyAllocator::createMemoryAllocator(buddyMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(buddyAllocator, nullptr) << "BuddyAllocator creation failed.";

size\_t blockSize = 64;

// Выделяем блоки

void\* listBlock1 = ListAllocator::alloc(listAllocator, blockSize);

ASSERT\_NE(listBlock1, nullptr) << "ListAllocator allocation failed.";

void\* buddyBlock1 = BuddyAllocator::alloc(buddyAllocator, blockSize);

ASSERT\_NE(buddyBlock1, nullptr) << "BuddyAllocator allocation failed.";

// Освобождаем блоки

ListAllocator::freeBlock(listAllocator, listBlock1);

BuddyAllocator::freeBlock(buddyAllocator, buddyBlock1);

// Выделяем блоки снова

void\* listBlock2 = ListAllocator::alloc(listAllocator, blockSize);

ASSERT\_NE(listBlock2, nullptr) << "ListAllocator failed to reuse freed block.";

void\* buddyBlock2 = BuddyAllocator::alloc(buddyAllocator, blockSize);

ASSERT\_NE(buddyBlock2, nullptr) << "BuddyAllocator failed to reuse freed block.";

// Освобождаем блоки

ListAllocator::freeBlock(listAllocator, listBlock2);

BuddyAllocator::freeBlock(buddyAllocator, buddyBlock2);

}

// Тест большого числа блоков

TEST(AllocatorComparisonTest, LargeNumberOfBlocks) {

char listMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для ListAllocator

char buddyMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для BuddyAllocator

ListAllocator::Allocator\* listAllocator = ListAllocator::createMemoryAllocator(listMemory, TEST\_MEMORY\_SIZE);

BuddyAllocator::Allocator\* buddyAllocator = BuddyAllocator::createMemoryAllocator(buddyMemory, TEST\_MEMORY\_SIZE);

const size\_t numBlocks = 10000;

const size\_t blockSize = 64;

double listTime = measureTime([&]() {

for (size\_t i = 0; i < numBlocks; ++i) {

void\* block = ListAllocator::alloc(listAllocator, blockSize);

ASSERT\_NE(block, nullptr);

ListAllocator::freeBlock(listAllocator, block);

}

});

double buddyTime = measureTime([&]() {

for (size\_t i = 0; i < numBlocks; ++i) {

void\* block = BuddyAllocator::alloc(buddyAllocator, blockSize);

ASSERT\_NE(block, nullptr);

BuddyAllocator::freeBlock(buddyAllocator, block);

}

});

std::cout << "ListAllocator time for " << numBlocks << " blocks: " << listTime << " seconds." << std::endl;

std::cout << "BuddyAllocator time for " << numBlocks << " blocks: " << buddyTime << " seconds." << std::endl;

}

// Тест для проверки производительности при работе с блоками размером 2^n

TEST(AllocatorComparisonTest, PowerOfTwoBlocks) {

char listMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для ListAllocator

char buddyMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для BuddyAllocator

ListAllocator::Allocator\* listAllocator = ListAllocator::createMemoryAllocator(listMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(listAllocator, nullptr) << "ListAllocator creation failed.";

BuddyAllocator::Allocator\* buddyAllocator = BuddyAllocator::createMemoryAllocator(buddyMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(buddyAllocator, nullptr) << "BuddyAllocator creation failed.";

const size\_t numBlocks = 1000;

std::vector<size\_t> blockSizes = {32, 64, 128, 256, 512, 1024}; // Блоки размером 2^n

double listTime = measureTime([&]() {

for (size\_t i = 0; i < numBlocks; ++i) {

for (size\_t size : blockSizes) {

void\* block = ListAllocator::alloc(listAllocator, size);

ASSERT\_NE(block, nullptr) << "ListAllocator allocation failed for size " << size;

ListAllocator::freeBlock(listAllocator, block);

}

}

});

double buddyTime = measureTime([&]() {

for (size\_t i = 0; i < numBlocks; ++i) {

for (size\_t size : blockSizes) {

void\* block = BuddyAllocator::alloc(buddyAllocator, size);

ASSERT\_NE(block, nullptr) << "BuddyAllocator allocation failed for size " << size;

BuddyAllocator::freeBlock(buddyAllocator, block);

}

}

});

std::cout << "ListAllocator time for " << numBlocks << " blocks of power-of-two sizes: " << listTime << " seconds." << std::endl;

std::cout << "BuddyAllocator time for " << numBlocks << " blocks of power-of-two sizes: " << buddyTime << " seconds." << std::endl;

}

TEST(AllocatorComparisonTest, MemoryUsageCheck) {

char listMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для ListAllocator

char buddyMemory[TEST\_MEMORY\_SIZE]; // Отдельная память для BuddyAllocator

ListAllocator::Allocator\* listAllocator = ListAllocator::createMemoryAllocator(listMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(listAllocator, nullptr) << "ListAllocator creation failed.";

BuddyAllocator::Allocator\* buddyAllocator = BuddyAllocator::createMemoryAllocator(buddyMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(buddyAllocator, nullptr) << "BuddyAllocator creation failed.";

size\_t blockSize = 64;

// Выделяем блок

void\* listBlock = ListAllocator::alloc(listAllocator, blockSize);

ASSERT\_NE(listBlock, nullptr) << "ListAllocator allocation failed.";

ASSERT\_EQ(getHeapUsage(listAllocator), blockSize) << "ListAllocator memory usage is incorrect.";

void\* buddyBlock = BuddyAllocator::alloc(buddyAllocator, blockSize);

ASSERT\_NE(buddyBlock, nullptr) << "BuddyAllocator allocation failed.";

ASSERT\_EQ(getHeapUsage(buddyAllocator), blockSize) << "BuddyAllocator memory usage is incorrect.";

// Освобождаем блок

ListAllocator::freeBlock(listAllocator, listBlock);

ASSERT\_EQ(getHeapUsage(listAllocator), 0) << "ListAllocator memory usage is incorrect after free.";

BuddyAllocator::freeBlock(buddyAllocator, buddyBlock);

ASSERT\_EQ(getHeapUsage(buddyAllocator), 0) << "BuddyAllocator memory usage is incorrect after free.";

// Очистка памяти

memset(listMemory, 0, TEST\_MEMORY\_SIZE);

memset(buddyMemory, 0, TEST\_MEMORY\_SIZE);

}

TEST(AllocatorComparisonTest, BuddyAllocatorPerformance) {

const size\_t TEST\_MEMORY\_SIZE = 1024 \* 1024 \* 1024; // 1 GB (увеличьте размер памяти при необходимости)

// Выделяем память на куче вместо стека

char\* listMemory = new char[TEST\_MEMORY\_SIZE];

char\* buddyMemory = new char[TEST\_MEMORY\_SIZE];

ListAllocator::Allocator\* listAllocator = ListAllocator::createMemoryAllocator(listMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(listAllocator, nullptr) << "ListAllocator creation failed.";

BuddyAllocator::Allocator\* buddyAllocator = BuddyAllocator::createMemoryAllocator(buddyMemory, TEST\_MEMORY\_SIZE);

ASSERT\_NE(buddyAllocator, nullptr) << "BuddyAllocator creation failed.";

const size\_t numBlocks = 10000; // Количество блоков

const size\_t maxBlockSize = 1024 \* 1024; // Максимальный размер блока (1 MB)

std::vector<size\_t> blockSizes(numBlocks);

// Генерация случайных размеров блоков

std::srand(static\_cast<unsigned int>(std::time(nullptr)));

for (size\_t i = 0; i < numBlocks; ++i) {

blockSizes[i] = (std::rand() % maxBlockSize) + 1; // Размеры от 1 до maxBlockSize

}

// Тест для ListAllocator

size\_t listFailures = 0;

double listTime = measureTime([&]() {

for (size\_t i = 0; i < numBlocks; ++i) {

void\* block = ListAllocator::alloc(listAllocator, blockSizes[i]);

if (block == nullptr) {

listFailures++;

} else {

ListAllocator::freeBlock(listAllocator, block);

}

}

});

// Тест для BuddyAllocator

size\_t buddyFailures = 0;

double buddyTime = measureTime([&]() {

for (size\_t i = 0; i < numBlocks; ++i) {

void\* block = BuddyAllocator::alloc(buddyAllocator, blockSizes[i]);

if (block == nullptr) {

buddyFailures++;

} else {

BuddyAllocator::freeBlock(buddyAllocator, block);

}

}

});

std::cout << "ListAllocator time for " << numBlocks << " random-sized blocks: " << listTime << " seconds. Failures: " << listFailures << std::endl;

std::cout << "BuddyAllocator time for " << numBlocks << " random-sized blocks: " << buddyTime << " seconds. Failures: " << buddyFailures << std::endl;

// Ожидаем, что BuddyAllocator будет быстрее

EXPECT\_LT(buddyTime, listTime) << "BuddyAllocator should be faster than ListAllocator.";

// Проверяем, что количество сбоев BuddyAllocator не превышает допустимого предела

EXPECT\_LE(buddyFailures, 0) << "BuddyAllocator had too many allocation failures.";

// Освобождаем память

delete[] listMemory;

delete[] buddyMemory;

}

int main(int argc, char\*\* argv) {

::testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

**Результат тестов**

unix@DESKTOP-MPQDBS2:~/labs/osLabs/build/tests$ ./CP\_test

[==========] Running 9 tests from 1 test suite.

[----------] Global test environment set-up.

[----------] 9 tests from AllocatorComparisonTest

[ RUN ] AllocatorComparisonTest.CreationPerformance

Memory used by ListAllocator: 0 bytes.

Memory used by BuddyAllocator: 0 bytes.

ListAllocator creation time: 3.4e-05 seconds.

BuddyAllocator creation time: 1.28e-05 seconds.

[ OK ] AllocatorComparisonTest.CreationPerformance (4 ms)

[ RUN ] AllocatorComparisonTest.OutOfMemory

BuddyAllocator out of memory at block 32768

[ OK ] AllocatorComparisonTest.OutOfMemory (718 ms)

[ RUN ] AllocatorComparisonTest.AllocationAndFreePerformance

ListAllocator allocation/free time: 4.93e-05 seconds.

BuddyAllocator allocation/free time: 0.0001249 seconds.

[ OK ] AllocatorComparisonTest.AllocationAndFreePerformance (0 ms)

[ RUN ] AllocatorComparisonTest.MixedBlockSizes

[ OK ] AllocatorComparisonTest.MixedBlockSizes (0 ms)

[ RUN ] AllocatorComparisonTest.ReuseFreedBlocks

[ OK ] AllocatorComparisonTest.ReuseFreedBlocks (0 ms)

[ RUN ] AllocatorComparisonTest.LargeNumberOfBlocks

ListAllocator time for 10000 blocks: 0.0005093 seconds.

BuddyAllocator time for 10000 blocks: 0.0010015 seconds.

[ OK ] AllocatorComparisonTest.LargeNumberOfBlocks (1 ms)

[ RUN ] AllocatorComparisonTest.PowerOfTwoBlocks

ListAllocator time for 1000 blocks of power-of-two sizes: 0.0002786 seconds.

BuddyAllocator time for 1000 blocks of power-of-two sizes: 0.0006006 seconds.

[ OK ] AllocatorComparisonTest.PowerOfTwoBlocks (0 ms)

[ RUN ] AllocatorComparisonTest.MemoryUsageCheck

[ OK ] AllocatorComparisonTest.MemoryUsageCheck (0 ms)

[ RUN ] AllocatorComparisonTest.BuddyAllocatorPerformance

ListAllocator time for 10000 random-sized blocks: 0.0010778 seconds. Failures: 0

BuddyAllocator time for 10000 random-sized blocks: 0.0008886 seconds. Failures: 0

[ OK ] AllocatorComparisonTest.BuddyAllocatorPerformance (2 ms)

[----------] 9 tests from AllocatorComparisonTest (728 ms total)

[----------] Global test environment tear-down

[==========] 9 tests from 1 test suite ran. (728 ms total)

[ PASSED ] 9 tests.

**Где используются?**

#### **Блоки памяти размером 2n.**

* **Операционные системы:   
  Linux используется buddy allocator (аллокатор памяти), который работает с блоками 2n. Это помогает быстро выделять память для ядра системы.**
* **Графика и игры:**  
  В графических процессорах (GPU) память для текстур и буферов кадров выделяется блоками 2n
* **Сетевые технологии:**  
  В сетевых картах память для пакетов данных выделяется блоками 2n, чтобы ускорить обработку.

#### **Алгоритм свободных блоков.**

* **Программирование на C/C++:  
  Функции** malloc **и** free **используют алгоритм свободных блоков для выделения памяти.  
  (создаем массив или строку, память выделяется с помощью этого алгоритма.)**
* **Базы данных:   
  В базах данных память выделяется для хранения записей, индексов и временных данных.  
  (при добавлении новой записи в таблицу, база данных выделяет память подходящего размера.)**
* **Игры:  
  В играх память выделяется для объектов разного размера: текстуры, модели, звуки.   
  (игра загружает уровень, она выделяет память для всех объектов на карте.)**

#### ****Сравнительная таблица****

| Критерий | Блоки 2n | Алгоритм свободных блоков |
| --- | --- | --- |
| Скорость выделения | Быстрое (O(1)) | Медленное (O(n)) |
| Фрагментация | Внутренняя | Внешняя |
| Гибкость | Низкая | Высокая |
| Сложность реализации | Простая | Сложная |
| Эффективность памяти | Низкая (внутренняя фрагментация) | Высокая (минимизация потерь) |

**Итоги**

В ходе тестирования двух аллокаторов — **ListAllocator**, реализующего алгоритм "Первое подходящее" (First Fit), и **BuddyAllocator**, основанного на алгоритме "Блоки по степеням двойки" (Buddy System), — были выявлены их ключевые особенности, преимущества и недостатки. ListAllocator продемонстрировал высокую скорость выделения и освобождения памяти, особенно для небольших блоков. Это связано с простотой его работы: он ищет первый подходящий блок в списке свободной памяти, что позволяет быстро удовлетворять запросы. Однако такой подход может приводить к фрагментации памяти, особенно при длительной работе, когда свободные блоки разбиваются на мелкие части, которые сложно использовать эффективно. Это делает ListAllocator подходящим для систем, где важна скорость и простота, а фрагментация не является критичной, например, в простых приложениях или ранних версиях операционных систем.

BuddyAllocator, напротив, показал себя как более сложный, но эффективный инструмент для управления памятью. Его основное преимущество заключается в минимизации фрагментации благодаря использованию блоков, размер которых соответствует степеням двойки. Это позволяет эффективно управлять памятью, особенно при работе с большими блоками и случайными размерами. Однако BuddyAllocator требует больше времени для выделения и освобождения памяти из-за необходимости разделения и объединения блоков. В тестах BuddyAllocator показал лучшую производительность при работе с блоками случайного размера, что подтверждает его эффективность в сложных сценариях. Этот аллокатор подходит для систем, где критична минимизация фрагментации и требуется эффективное управление памятью, например, в ядре Linux, игровых движках или сетевых устройствах.

#### **ListAllocator:**

* **Минимальное время**: 0.000034 секунд.
* **Максимальное время**: **0.0010778** секунд.
* **Среднее время**: **0.00038774** секунд.

#### **BuddyAllocator:**

* **Минимальное время**: 0.0000128 секунд.
* **Максимальное время**: **0.0010015** секунд. .
* **Среднее время**: **0.00052556** секунд.

**Заключение**

Общие наблюдения показывают, что ListAllocator быстрее в большинстве тестов, особенно для небольших блоков, в то время как BuddyAllocator лучше справляется с большими блоками и случайными размерами, но требует больше времени для управления памятью. Выбор между этими аллокаторами зависит от конкретных требований системы. Если приложение работает с большим количеством небольших блоков и требует высокой скорости выделения памяти, предпочтение стоит отдать ListAllocator. Если же приложение работает с блоками переменного размера и требует минимизации фрагментации, BuddyAllocator будет более подходящим выбором. Таким образом, каждый из аллокаторов имеет свои сильные стороны и области применения, и выбор между ними должен основываться на специфике задач, которые решает система.