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»

Студент: Слободин Никита Алексеевич

Группа: М8О-203Б-23

Вариант: 19

Преподаватель: Миронов Евгений Сергеевич

Оценка: _____

Дата: _____

Подпись: _____

Москва, 2024

Постановка задачи

Исследование 2 аллокаторов памяти: необходимо реализовать два алгоритма аллокации памяти и сравнить их по следующим характеристикам:

1) Фактор использования
2) Скорость выделения блоков
3) Скорость освобождения блоков
4) 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. В отчете необходимо отобразить следующее:

1) Подробное описание каждого из исследуемых алгоритмов
2) Процесс тестирования
3) Обоснование подхода тестирования
4) Результаты тестирования
5) Заключение по проведенной работе

Вариант 19: Необходимо сравнить два алгоритма аллокации: списки свободных блоков (наиболее подходящее) и блоки по 2 в степени n

Описание каждого из исследуемых алгоритмов

Аллокатор с блоками размеров 2^n

Аллокатор с блоками размеров 2^n представляет собой алгоритм управления памятью, который делит доступную память на блоки фиксированных размеров, являющихся степенями двойки (например, 16, 32, 64 байт и т.д.). Этот подход обеспечивает простоту и эффективность при выделении и освобождении памяти, поскольку размеры блоков заранее определены и легко сопоставимы с бинарными операциями.

Основные особенности:

• **Фиксированные размеры блоков:** Все блоки имеют размеры, являющиеся степенями двойки, что упрощает управление памятью и снижает сложность алгоритмов.

• **Простота управления:** Быстрое выделение и освобождение памяти благодаря предопределённым размерам блоков.

• **Возможная внутренняя фрагментация:** Из-за фиксированных размеров блоков, запросы на память, которые не являются степенями двойки, приводят к сильной внутренней фрагментации.

• **Эффективное использование памяти:** За счёт предопределённых размеров блоков и упрощённого управления памятью, аллокатор может эффективно переиспользовать блоки.

Free List Allocator (Best fit)

Free List Allocator управляет памятью, используя связный список свободных блоков. Блоки могут быть любого размера. Для выделения памяти выбирается самый подходящий по размеру блок, а оставшаяся часть блока добавляется обратно в список.

Основные особенности:

- Гибкость в размере блоков.
- Возможность более эффективного использования памяти.
- Неэффективность при выделении памяти из-за перебора всех вариантов.
- Потенциально высокая внешняя фрагментация из-за разбиения блоков.

Процесс тестирования

Базовый нагрузочный тест 1

1. Выделяем по 32 МБ для каждого аллокатора
2. Генерируем 1 млн блоков случайного размера
3. Пытаемся аллоцировать максимальное кол-во этих блоков с помощью первого и второго аллокатора соответственно
4. Делаем замеры метрик (скорость выделения и освобождения, фактор использования)

Бенчмарк 1

1. Генерируем случайные операции (60% аллокаций, 40% деаллокаций)
2. Выделяем по 32 МБ для каждого аллокатора
3. Проделываем 100 тыс. операций
4. Измеряем фрагментацию и фактор использования

5. Промежуточные результаты записываем (каждые 10 тыс. операций)

Бенчмарк 2

1. Заводим фиксированный размер блока в 256 байт
2. Выделяем 256 МБ для каждого аллокатора
3. Делаем 100 тыс. аллокаций с промежуточной деаллокацией для переиспользования блоков
4. Измеряем общее и среднее время выделения памяти
5. Промежуточные результаты записываем (каждые 10 тыс. операций)

Бенчмарк 3

1. Предварительно выделяем 100 тыс. блоков (по 256 байт)
2. Выделяем по 256 МБ для каждого аллокатора
3. Измеряем общее и среднее время деаллокации
4. Промежуточные результаты записываем (каждые 10 тыс. операций)

Бенчмарк 4

1. Определяем шаги памяти от 16 МБ до 256 МБ
2. Генерируем 1 млн блоков случайного размера (16-8192 байт)
3. Тестируем производительность на каждом объеме памяти аналогично нагрузочному тесту 1
4. Измеряем скорость выделения/освобождения, фактор использования
5. Записываем результаты для каждого объема памяти соответственно

Пример вывода для базового нагрузочного теста (рандомные блоки размером до 8 кБ)

```
Запуск Benchmark для Block Allocator 2^n...

Block Allocator: Allocation failed at iteration 10703

Запуск Benchmark для Free List Allocator...

Free List Allocator: Allocation failed at iteration 15715


Результаты Benchmark:


Block Allocator 2^n:
```

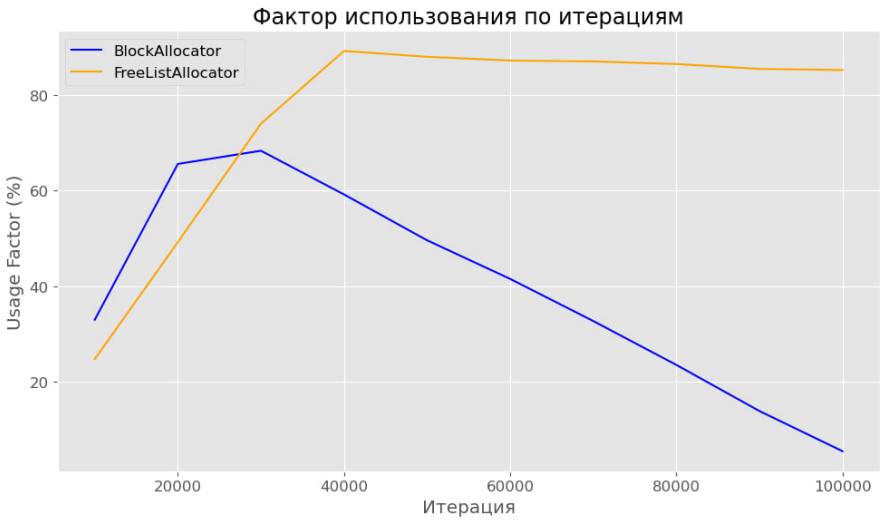
```
  Количество выделений: 10703
  Количество освобождений: 10703
  Общее время выделения: 34.2285 ms (Среднее: 2.17808 µs)
  Общее время освобождения: 0.35684ms (Среднее: 0.022707 µs)
  Фактор использования: 65.2781 %

Free List Allocator:
  Количество выделений: 15715
  Количество освобождений: 15715
  Общее время выделения: 247.55 ms (Среднее: 15.7525 µs)
  Общее время освобождения: 0.538845 ms (Среднее: 0.034288 µs)
  Фактор использования: 95.8507 %
```

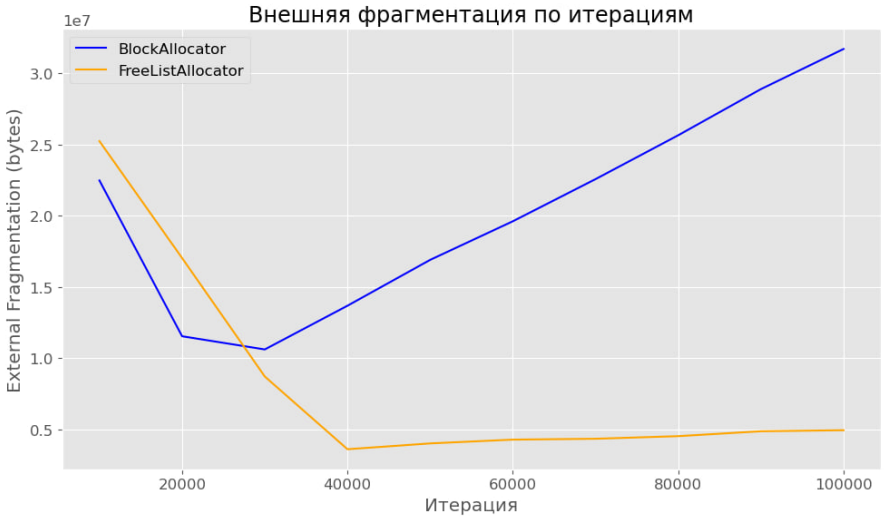
Критерии тестирования

- Время выполнения операций аллокации и деаллокации (позволяет определить производительность базовых операций для конкретной реализации алгоритма аллокатора);
- Фактор использования (как способоность выделить рандомные блоки в условиях ограниченной памяти, а также как процент используемой памяти от общего объема). Является хорошей метрикой, т.к. показывает долю реальной памяти, которая используется на хранение данных, исключая фрагментированные или неиспользуемые блоки.

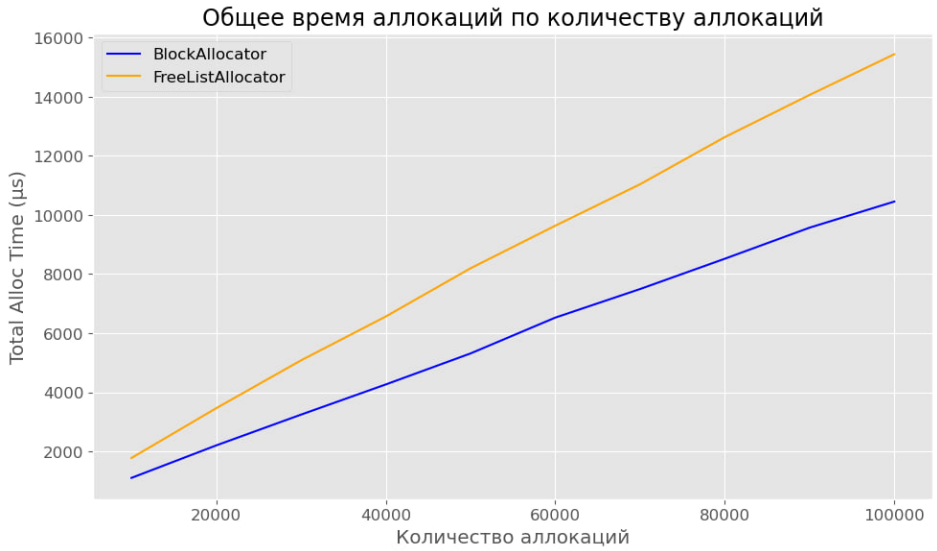
Графики (результаты бенчмарков)

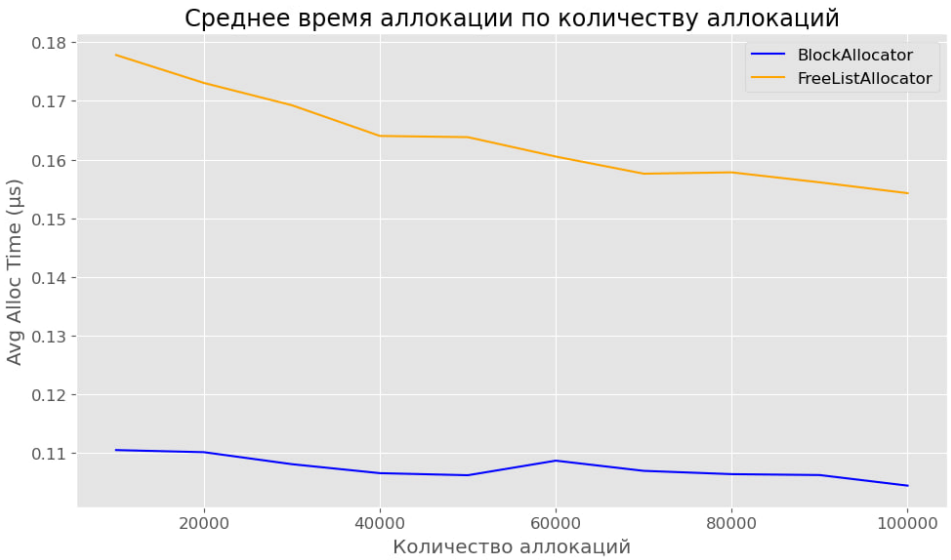
Бенчмарк 3


Фактор использования по итерациям
BlockAllocator
FreeListAllocator
Usage Factor (%)
20
40
60
80
20000
40000
60000
80000
100000
Итерация

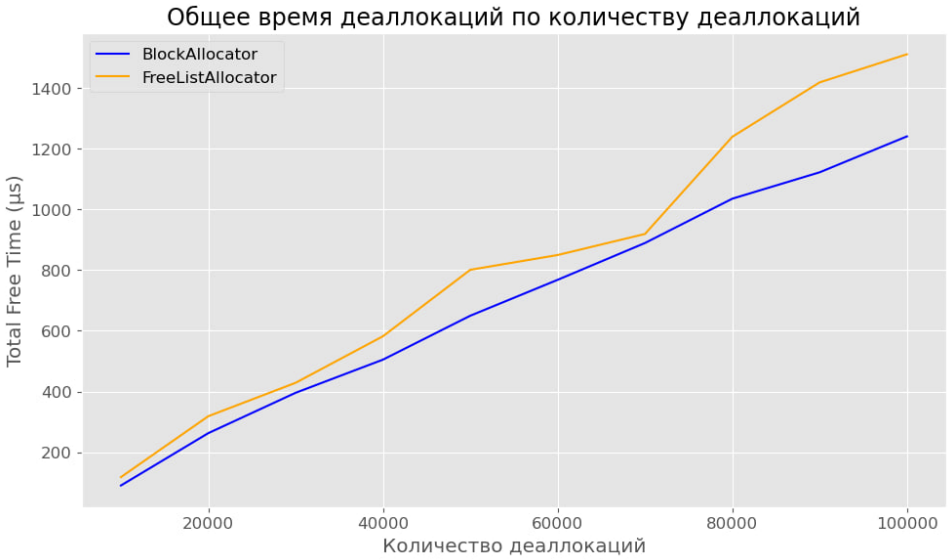

Внешняя фрагментация по итерациям
1e7
BlockAllocator
FreeListAllocator
External Fragmentation (bytes)
0.5
1.0
1.5
2.0
2.5
3.0
20000
40000
60000
80000
100000
Итерация

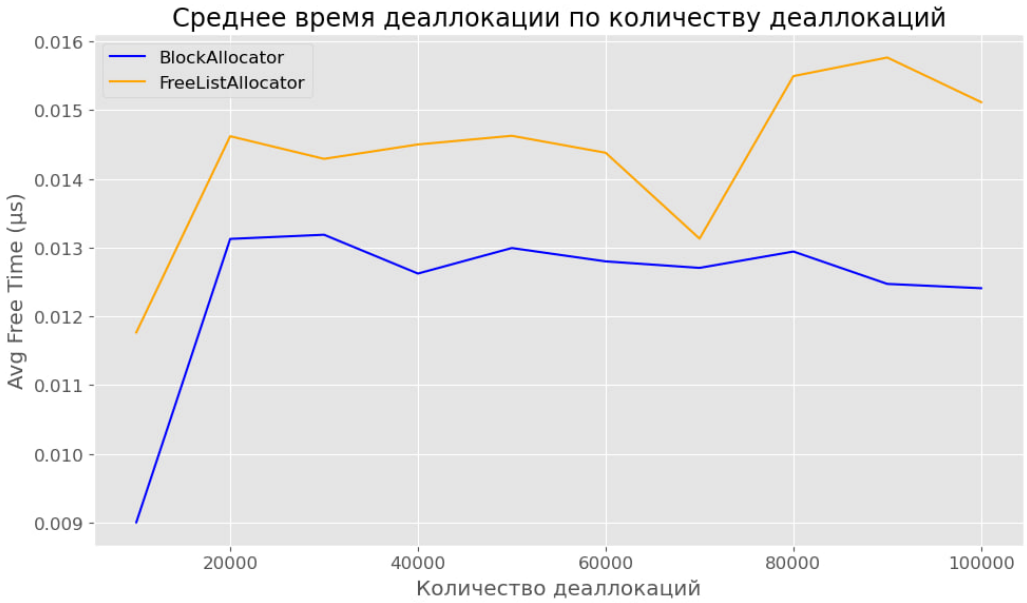
Бенчмарк 1


Общее время аллокаций по количеству аллокаций
BlockAllocator
FreeListAllocator
Total Alloc Time (µs)
2000
4000
6000
8000
10000
12000
14000
16000
20000
40000
60000
80000
100000
Количество аллокаций

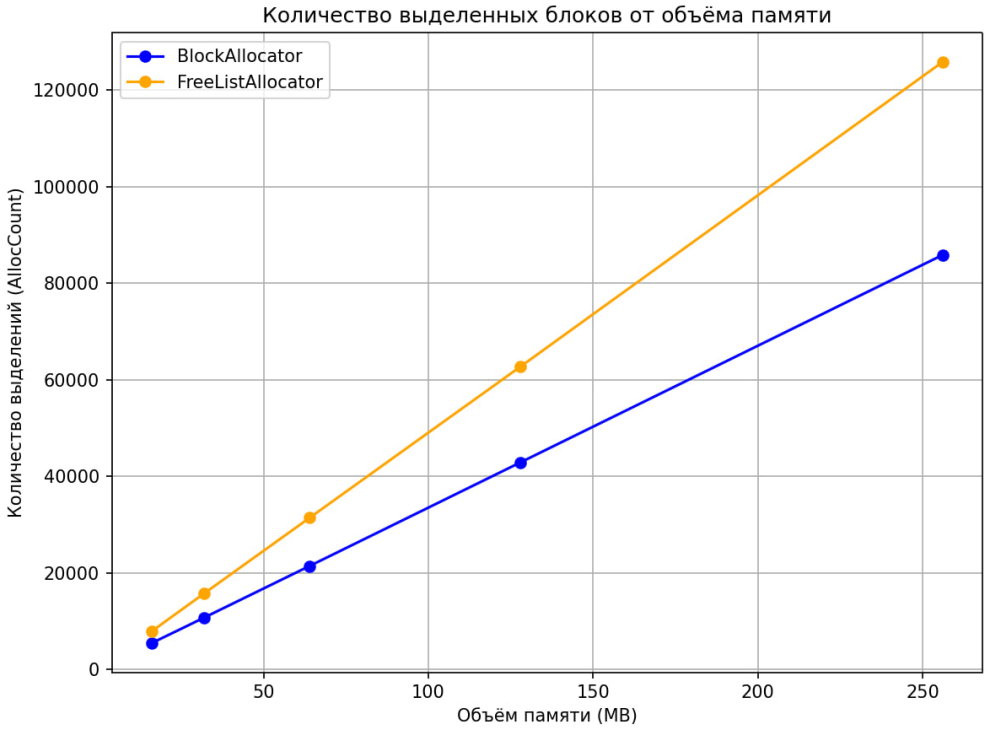

Среднее время аллокации по количеству аллокаций
BlockAllocator
FreeListAllocator
Avg Alloc Time (µs)
0.18
0.17
0.16
0.15
0.14
0.13
0.12
0.11
20000
40000
60000
80000
100000
Количество аллокаций

Бенчмарк 2


Общее время деаллокаций по количеству деаллокаций
BlockAllocator
FreeListAllocator
Total Free Time (µs)
1400
1200
1000
800
600
400
200
20000
40000
60000
80000
100000
Количество деаллокаций



Бенчмарк 4 (AllocCount – кол-во успешных попыток выделения блоков рандомного размера в условиях ограничения памяти)



Анализ результатов

- **Фактор использования:** Free List Allocator показал более высокий фактор использования, нежели Block 2^n аллокатор. Результат ожидаем, т.к. хоть best fit аллокация и дольше, чем поиск первого свободного блока размером 2^n, но это позволило по максимуму занимать память полезными данными. Block 2^n аллокатор может также показывать

высокую эффективность, но в случае, если объем большинства аллокаций происходит близко к степеням двойки

- **Производительность выделения/освобождения:** В текущей реализации Block 2^n ожидаемо оказался быстрее по всем пунктам. Обусловлено это как минимум тем, что этому аллокатору легко подобрать нужный размер блока ($\log_2 n + 1$, где n – запрашиваемый размер), в то время как Free List аллокатор из за поиска O(F), где F – кол-во блоков, каждый раз имеет почти полное линейное время поиско подходящего блока. Деаллокация же в обоих аллокаторах реализована за примерно O(1) время, но Block 2^n аллокатор побеждает и тут за счет меньшего кол-ва тактов внутри этой операции.

Вывод

Результат:

- Block 2^n аллокатор быстрее выполняет операции выделения и освобождения, однако страдает от фрагментации при запросах выделения блоков, не кратных степеням двойки. Также наиболее прост в написании и использовании
- Free List аллокатор выполняет операции дольше, но на тестах с переиспользованием блоков и при рассчетах фактора использования показывает себя сильно лучше

Возможные улучшения:

- **Block 2^n аллокатор:** реализация механизма слияния (подобно алгоритму двойников), а также реализация поиска блока для аллокации через AVL-дерево
- **Free List аллокатор:** реализация поиска через хэш-таблицы для ускорения операций

Код программы

```
// freeListAllocator.hpp
#ifndef FREELIST_ALLOCATOR_HPP
#define FREELIST_ALLOCATOR_HPP

#include <iostream>
#include <cstdlib>
#include <cstddef>
#include <cstring>

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

class FreeListAllocator {
public:
    FreeListAllocator(void *realMemory, size_t memorySize)
        : memory(realMemory), memorySize(memorySize) {
        freeList = (Block *)realMemory;
        freeList->size = memorySize;
        freeList->next = NULL;
    }

    void *alloc(size_t blockSize) {
        Block *bestFitPrev = NULL;
        Block *bestFit = NULL;
        Block *prev = NULL;
        Block *current = freeList;

        while (current) {
            if (current->size >= blockSize) {
                if (!bestFit || current->size < bestFit->size) {
                    bestFitPrev = prev;
                    bestFit = current;
                }
            }
            prev = current;
            current = current->next;
        }

        if (!bestFit) {
            return NULL;
        }

        if (bestFit->size > blockSize + sizeof(Block)) {
            Block *newBlock = (Block *)((char *)bestFit + blockSize);
            newBlock->size = bestFit->size - blockSize;
```

```
            newBlock->next = bestFit->next;

            bestFit->size = blockSize;
            bestFit->next = newBlock;
        }

        if (bestFitPrev) {
            bestFitPrev->next = bestFit->next;
        } else {
            freeList = bestFit->next;
        }

        return (char *)bestFit + sizeof(Block);
    }

    void freeBlock(void *ptr) {
        Block *blockToFree = (Block *)((char *)ptr - sizeof(Block));
        blockToFree->next = freeList;
        freeList = blockToFree;
    }

    void printMemoryLayout() const {
        Block *current = freeList;
        size_t offset = 0;

        while (current) {
            std::cout << offset << " - " << offset + current->size
                      << " [free]\n";
            offset += current->size;
            current = current->next;
        }
    }

    void calculateFragmentation(size_t &internalFragmentation, size_t
&externalFragmentation) const {
        internalFragmentation = 0;
        externalFragmentation = 0;

        Block *current = freeList;
        while (current) {
            externalFragmentation += current->size;
            current = current->next;
        }
    }

    double calculateUsageFactor() const {
        size_t externalFrag;
        size_t internalFrag;
        calculateFragmentation(internalFrag, externalFrag);
        size_t usedMemory = memorySize - externalFrag;
```

```
        return (static_cast<double>(usedMemory) / memorySize) * 100.0;
    }

private:
    Block *freeList;
    void *memory;
    size_t memorySize;

    bool isBlockFree(Block *block) const {
        Block *current = freeList;
        while (current) {
            if (current == block) {
                return true;
            }
            current = current->next;
        }
        return false;
    }
};

#endif // FREELIST_ALLOCATOR_HPP
```

```
// blockAllocator.hpp
#ifndef BLOCK_ALLOCATOR_HPP
#define BLOCK_ALLOCATOR_HPP

#include <iostream>
#include <cstdlib>
#include <cstddef>
#include <cmath>

struct BlockPowerOfTwo {
    size_t size;
    BlockPowerOfTwo *next;
};

class BlockAllocator {
public:
    BlockAllocator(void *realMemory, size_t memorySize)
        : memory(realMemory), memorySize(memorySize) {
        freeList = (BlockPowerOfTwo *)realMemory;
        freeList->size = memorySize;
        freeList->next = NULL;
    }

    void *alloc(size_t blockSize) {
        blockSize = roundUpToPowerOfTwo(blockSize);
```

```
        BlockPowerOfTwo *prev = NULL;
        BlockPowerOfTwo *current = freeList;

        while (current) {
            if (current->size >= blockSize) {
                if (current->size > blockSize) {
                    BlockPowerOfTwo *newBlock = (BlockPowerOfTwo *)((char *)current +
blockSize);
                    newBlock->size = current->size - blockSize;
                    newBlock->next = current->next;

                    current->size = blockSize;
                    current->next = newBlock;
                }

                if (prev) {
                    prev->next = current->next;
                } else {
                    freeList = current->next;
                }

                return (char *)current + sizeof(BlockPowerOfTwo);
            }

            prev = current;
            current = current->next;
        }

        return NULL;
    }

    void freeBlock(void *ptr) {
        if (!ptr) return;

        BlockPowerOfTwo *blockToFree = (BlockPowerOfTwo *)((char *)ptr -
sizeof(BlockPowerOfTwo));
        blockToFree->next = freeList;
        freeList = blockToFree;
    }

    void printMemoryLayout() {
        BlockPowerOfTwo *current = freeList;
        size_t offset = 0;

        while (current) {
            std::cout << offset << " - " << offset + current->size
                      << " [free]\n";
            offset += current->size;
            current = current->next;
```

```
        }
    }

    void calculateFragmentation(size_t &internalFragmentation, size_t
&externalFragmentation) const {
        internalFragmentation = 0;
        externalFragmentation = 0;

        BlockPowerOfTwo *current = freeList;
        while (current) {
            externalFragmentation += current->size;
            current = current->next;
        }
    }

    double calculateUsageFactor() const {
        size_t externalFrag;
        calculateFragmentation(/*internalFragmentation=*/std::ignore, externalFrag);
        size_t usedMemory = memorySize - externalFrag;
        return (static_cast<double>(usedMemory) / memorySize) * 100.0;
    }

private:
    BlockPowerOfTwo *freeList;
    void *memory;
    size_t memorySize;

    size_t roundUpToPowerOfTwo(size_t size) const {
        size_t power = 1;
        while (power < size) {
            power <<= 1;
        }
        return power;
    }

    bool isBlockFree(BlockPowerOfTwo *block) const {
        BlockPowerOfTwo *current = freeList;
        while (current) {
            if (current == block) {
                return true;
            }
            current = current->next;
        }
        return false;
    }
};

#endif // BLOCK_ALLOCATOR_HPP
```

```cpp
#include "blockAllocator.hpp"
#include "freeListAllocator.hpp"
#include <vector>
#include <random>
#include <chrono>
#include <iostream>

int main() {
    const size_t memorySize = 1024 * 1024 * 32; // 32 MB
    const size_t maxBlockSize = 1024 * 8;       // 8 KB максимальный размер блока
    const size_t numAllocations = 1000000;       // Количество аллокаций

    void* blockMemory = malloc(memorySize);      // Для аллокатора блоков 2^n
    void* freeListMemory = malloc(memorySize);   // Для Free List Allocator

    if (!blockMemory || !freeListMemory) {
        std::cerr << "Не удалось выделить память для аллокаторов." << std::endl;
        return 1;
    }

    BlockAllocator blockAllocator(blockMemory, memorySize);
    FreeListAllocator freeListAllocator(freeListMemory, memorySize);

    std::vector<size_t> allocationSizes;
    allocationSizes.reserve(numAllocations);

    std::mt19937 rng(42);
    std::uniform_int_distribution<size_t> dist(16, maxBlockSize);
    for (size_t i = 0; i < numAllocations; ++i) {
        allocationSizes.push_back(dist(rng));
    }

    std::vector<void*> blockPointers;
    blockPointers.reserve(numAllocations);
    std::vector<void*> freeListPointers;
    freeListPointers.reserve(numAllocations);

    size_t allocationCount = 0;
    size_t deallocationCount = 0;

    size_t blockTotalAllocated = 0;
    size_t freeListTotalAllocated = 0;

    // Benchmark для Block Allocator (2^n)
    std::cout << "Запуск Benchmark для Block Allocator 2^n..." << std::endl;
    auto startBlockAlloc = std::chrono::high_resolution_clock::now();
    auto totalBlockAllocTime = std::chrono::duration<double, std::micro>::zero();
    auto totalBlockFreeTime = std::chrono::duration<double, std::micro>::zero();

    for (size_t i = 0; i < allocationSizes.size(); ++i) {
```

```
        auto allocStart = std::chrono::high_resolution_clock::now();
        void* ptr = blockAllocator.alloc(allocationSizes[i]);
        auto allocEnd = std::chrono::high_resolution_clock::now();

        if (ptr == NULL) {
            std::cerr << "Block Allocator: Allocation failed at iteration " << i <<
std::endl;
            break;
        }

        blockPointers.push_back(ptr);
        allocationCount++;
        blockTotalAllocated += allocationSizes[i];
        totalBlockAllocTime += allocEnd - allocStart;

        if (allocationCount % 2 == 0) {
            size_t deallocIndex = allocationCount - 2;
            if (deallocIndex < blockPointers.size() && blockPointers[deallocIndex] !=
nullptr) {
                auto freeStart = std::chrono::high_resolution_clock::now();
                blockAllocator.freeBlock(blockPointers[deallocIndex]);
                auto freeEnd = std::chrono::high_resolution_clock::now();

                totalBlockFreeTime += freeEnd - freeStart;
                blockTotalAllocated -= allocationSizes[deallocIndex];
                blockPointers[deallocIndex] = nullptr;
                deallocationCount++;
            }
        }
    }

    double blockUtilization = static_cast<double>(blockTotalAllocated) / memorySize *
100.0;

    // Освобождение оставшихся блоков для Block Allocator
    for (size_t i = 0; i < blockPointers.size(); ++i) {
        if (blockPointers[i] != nullptr) {
            auto freeStart = std::chrono::high_resolution_clock::now();
            blockAllocator.freeBlock(blockPointers[i]);
            auto freeEnd = std::chrono::high_resolution_clock::now();

            totalBlockFreeTime += freeEnd - freeStart;
            blockTotalAllocated -= allocationSizes[i];
            deallocationCount++;
            blockPointers[i] = nullptr;
        }
    }

    auto endBlockAlloc = std::chrono::high_resolution_clock::now();
```

```
    // Benchmark для Free List Allocator
    std::cout << "Запуск Benchmark для Free List Allocator..." << std::endl;

    auto startFreeListAlloc = std::chrono::high_resolution_clock::now();
    auto totalFreeListAllocTime = std::chrono::duration<double, std::micro>::zero();
    auto totalFreeListFreeTime = std::chrono::duration<double, std::micro>::zero();

    allocationCount = 0;
    deallocationCount = 0;
    freeListTotalAllocated = 0;

    for (size_t i = 0; i < allocationSizes.size(); ++i) {
        auto allocStart = std::chrono::high_resolution_clock::now();
        void* ptr = freeListAllocator.alloc(allocationSizes[i]);
        auto allocEnd = std::chrono::high_resolution_clock::now();

        if (ptr == NULL) {
            std::cerr << "Free List Allocator: Allocation failed at iteration " << i <<
std::endl;
            break;
        }

        freeListPointers.push_back(ptr);
        allocationCount++;
        freeListTotalAllocated += allocationSizes[i];
        totalFreeListAllocTime += allocEnd - allocStart;

        if (allocationCount % 2 == 0) {
            size_t deallocIndex = allocationCount - 2;
            if (deallocIndex < freeListPointers.size() && freeListPointers[deallocIndex]
!= nullptr) {
                auto freeStart = std::chrono::high_resolution_clock::now();
                freeListAllocator.freeBlock(freeListPointers[deallocIndex]);
                auto freeEnd = std::chrono::high_resolution_clock::now();

                totalFreeListFreeTime += freeEnd - freeStart;
                freeListTotalAllocated -= allocationSizes[deallocIndex];
                freeListPointers[deallocIndex] = nullptr;
                deallocationCount++;
            }
        }
    }

    double freeListUtilization = static_cast<double>(freeListTotalAllocated) / memorySize
* 100.0;

    // Освобождение оставшихся блоков для Free List Allocator
    for (size_t i = 0; i < freeListPointers.size(); ++i) {
        if (freeListPointers[i] != nullptr) {
            auto freeStart = std::chrono::high_resolution_clock::now();
```

```
            freeListAllocator.freeBlock(freeListPointers[i]);
            auto freeEnd = std::chrono::high_resolution_clock::now();

            totalFreeListFreeTime += freeEnd - freeStart;
            freeListTotalAllocated -= allocationSizes[i];
            deallocationCount++;
            freeListPointers[i] = nullptr;
        }
    }

    auto endFreeListAlloc = std::chrono::high_resolution_clock::now();

    // Расчет времени в миллисекундах
    double blockAllocTimeMs = totalBlockAllocTime.count() / 1000.0;
    double blockFreeTimeMs = totalBlockFreeTime.count() / 1000.0;

    double freeListAllocTimeMs = totalFreeListAllocTime.count() / 1000.0;
    double freeListFreeTimeMs = totalFreeListFreeTime.count() / 1000.0;

    // Расчет среднего времени в микросекундах
    double avgBlockAllocUs = (allocationCount > 0) ? (totalBlockAllocTime.count() /
allocationCount) : 0.0;
    double avgBlockFreeUs = (deallocationCount > 0) ? (totalBlockFreeTime.count() /
deallocationCount) : 0.0;

    double avgFreeListAllocUs = (allocationCount > 0) ? (totalFreeListAllocTime.count() /
allocationCount) : 0.0;
    double avgFreeListFreeUs = (deallocationCount > 0) ? (totalFreeListFreeTime.count() /
deallocationCount) : 0.0;

    // Вывод результатов Benchmark
    std::cout << "\nРезультаты Benchmark:\n\n";

    std::cout << "Block Allocator 2^n:\n";
    std::cout << "  Количество выделений: " << allocationCount << "\n";
    std::cout << "  Количество освобождений: " << deallocationCount << "\n";
    std::cout << "  Общее время выделения: " << blockAllocTimeMs << " ms (Среднее: "
              << avgBlockAllocUs << " µs)\n";
    std::cout << "  Общее время освобождения: " << blockFreeTimeMs << " ms (Среднее: "
              << avgBlockFreeUs << " µs)\n";
    std::cout << "  Фактор использования: " << blockUtilization << " %\n\n";

    std::cout << "Free List Allocator:\n";
    std::cout << "  Количество выделений: " << allocationCount << "\n";
    std::cout << "  Количество освобождений: " << deallocationCount << "\n";
    std::cout << "  Общее время выделения: " << freeListAllocTimeMs << " ms (Среднее: "
              << avgFreeListAllocUs << " µs)\n";
    std::cout << "  Общее время освобождения: " << freeListFreeTimeMs << " ms (Среднее: "
              << avgFreeListFreeUs << " µs)\n";
    std::cout << "  Фактор использования: " << freeListUtilization << " %\n\n";
```

```
    // Освобождение выделенной памяти
    free(blockMemory);
    free(freeListMemory);

    return 0; // Завершаем программу
}
```