Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Курсовой проект по курсу
«Операционные системы»

Студент: Скрипачев Фёдор Михайлович
Группа: М8О-209Б-23
Вариант: 16
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2025

Содержание

Репозиторий

https://github.com/gthcbr25/osi/tree/main/kp

Постановка задачи

Цель работы
Целью работы является:

- Приобретение практических навыков в использовании знаний, полученных в течении курса
- Проведение исследования в выбранной предметной области

Задание
Необходимо спроектировать и реализовать программный прототип в соответствии с выбранным вариантом. Произвести анализ и сделать вывод на основании данных, полученных при работе программного прототипа.
Спроектировать консольную клиент-серверную игру на основе технологии Pipes.
Вариант №16: необходимо сравнить два алгоритма аллокации: списки свободных блоков (наиболее подходящее) и алгоритм двойников.

Общие сведения о программе

Данный код реализует два различных подхода к управлению динамической памятью: аллокатор списка свободных блоков (List Allocator) и аллокатор двойников (Buddy Allocator). Оба метода направлены на эффективное использование выделенной области памяти, однако реализуют это разными способами, с разной степенью гибкости, эффективности и сложности.

Общий метод и алгоритм решения

Аллокатор списка свободных блоков использует связанный список блоков памяти. Каждый блок хранит размер, статус (свободен/занят) и указатель на следующий элемент. При выделении памяти ищется подходящий свободный блок. Если он слишком большой — делится на части. При освобождении блок помечается свободным и объединяется с соседними свободными блоками для снижения фрагментации. **Плюсы**: простота реализации, гибкость в размерах блоков.

Минусы: возможны замедления при большом количестве блоков, риск фрагментации при частых операциях. Аллокатор двойников делит память на блоки-степени двойки (64, 128 и т.д.). Для каждого размера есть отдельный список свободных блоков. Если подходящего блока нет, больший блок делится пополам до получения нужного размера. При освобождении блок объединяется с «двойником» (соседним блоком того же размера), если он свободен. **Плюсы**: высокая эффективность, минимизация фрагментации. **Минусы**: ограничение размеров степенями двойки, что снижает гибкость.

Исходный код

Main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <iostream>
#include <unistd.h>
#include <chrono>
#include <cstdlib>
#include <ctime>

// ------------------- Список свободных блоков -------------------

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

```
typedef struct {
    Block* head;
    void* memory_pool;
} ListAllocator;

ListAllocator* createListAllocator(void* memory, size_t size) {
    ListAllocator* allocator = (ListAllocator*)malloc(sizeof(ListAllocator));
    allocator->memory_pool = memory;
    allocator->head = (Block*)memory;
    allocator->head->size = size - sizeof(Block);
    allocator->head->is_free = true;
    allocator->head->next = NULL;
    return allocator;
}

void* listAlloc(ListAllocator* allocator, size_t size) {
    Block* current = allocator->head;
    while (current) {
        if (current->is_free && current->size >= size) {
            if (current->size > size + sizeof(Block)) {
                // Разделение блока
                Block* new_block = (Block*)((char*)current + sizeof(Block) + size);
                new_block->size = current->size - size - sizeof(Block);
                new_block->is_free = true;
                new_block->next = current->next;
                current->next = new_block;
            }
            current->is_free = false;
            current->size = size;
```

```
            return (char*)current + sizeof(Block);
        }
        current = current->next;
    }
    return NULL; // Нет подходящего блока
}

void listFree(ListAllocator* allocator, void* ptr) {
    if (!ptr) return;
    Block* block = (Block*)((char*)ptr - sizeof(Block));
    block->is_free = true;

    // Объединение соседних свободных блоков
    Block* current = allocator->head;
    while (current) {
        if (current->is_free && current->next && current->next->is_free) {
            current->size += sizeof(Block) + current->next->size;
            current->next = current->next->next;
        } else {
            current = current->next;
        }
    }
}

void destroyListAllocator(ListAllocator* allocator) {
    allocator->head = NULL; // Блоки в memory_pool автоматически освобождаются.
    free(allocator);
}
```

```
// ------------------- Алгоритм двойников -------------------

#define DIV_ROUNDUP(A, B) (((A) + (B)-1) / (B))
#define ALIGN_UP(A, B) (DIV_ROUNDUP((A), (B)) * (B))

typedef struct BuddyBlock {
    size_t blockSize;
    bool isFree;
} BuddyBlock;

typedef struct BuddyAllocator {
    BuddyBlock *head;
    BuddyBlock *tail;
    void *data;
    size_t totalSize;
    bool expanded;
    bool debug;
} BuddyAllocator;

BuddyBlock *next(BuddyBlock *block) {
    return (BuddyBlock *)((uint8_t *)block + block->blockSize);
}

BuddyBlock *split(BuddyBlock *block, size_t size) {
    // Пока размер блока больше нужного, делим его пополам
    while (block->blockSize > size) {
```

```
        size_t newSize = block->blockSize >> 1;
        block->blockSize = newSize;
        BuddyBlock *buddy = next(block);
        buddy->blockSize = newSize;
        buddy->isFree = true;
    }
    block->isFree = false;
    return block;
}

BuddyBlock *findBest(BuddyAllocator *allocator, size_t size) {
    BuddyBlock *block = allocator->head;
    BuddyBlock *bestBlock = NULL;

    // Ищем первый блок, который подходит по размеру
    while (block < allocator->tail) {
        if (block->isFree && block->blockSize >= size &&
            (bestBlock == NULL || block->blockSize < bestBlock->blockSize)) {
            bestBlock = block;
        }
        block = next(block);
    }

    // Если нашли подходящий блок, но его размер больше требуемого,
разделяем его
    if (bestBlock) {
        // Если блок слишком велик, разделим его до нужного размера
        if (bestBlock->blockSize > size) {
            return split(bestBlock, size);
```

```
        } else {
            // Если блок идеально подходит, выделяем его
            bestBlock->isFree = false;
            return bestBlock;
        }
    }
    return NULL;
}

size_t requiredSize(size_t size) {
    size += sizeof(BuddyBlock);
    size = ALIGN_UP(size, sizeof(BuddyBlock));
    size_t actualSize = sizeof(BuddyBlock);
    while (actualSize < size) {
        actualSize <<= 1;
    }
    return actualSize;
}

void coalesce(BuddyAllocator *allocator) {
    BuddyBlock *block = allocator->head;

    // Пробегаем все блоки и сливаем соседние свободные блоки
    while (block < allocator->tail) {
        BuddyBlock *buddy = next(block);
        if (buddy >= allocator->tail) break;

        if (block->isFree && buddy->isFree && block->blockSize == buddy->blockSize) {
```

```
            block->blockSize <<= 1;
            continue;
        }
        block = next(block);
    }
}

void expand(BuddyAllocator *allocator, size_t size) {
    size_t currentSize = allocator->head ? allocator->head->blockSize : 0;
    size_t requiredSize = size + currentSize;

    // Округляем размер до ближайшей степени двойки
    size_t newSize = 1;
    while (newSize < requiredSize) {
        newSize <<= 1; // Сдвиг влево, чтобы удвоить размер
    }



    void *newData = realloc(allocator->data, newSize);
    if (!newData) {
        fprintf(stderr, "Failed to expand memory.\n");
        exit(EXIT_FAILURE);
    }

    // Обновляем указатели на новую память
    allocator->data = newData;
    allocator->head = (BuddyBlock *)allocator->data;
    allocator->tail = (BuddyBlock *)((uint8_t *)allocator->data + newSize);
```

```
    // Настраиваем блок
    allocator->head->blockSize = newSize;
    allocator->head->isFree = true;

    if (allocator->debug) {
        printf("Expanded heap to %zu bytes\n", newSize);
    }
}



BuddyAllocator *createBuddyAllocator(size_t size, bool debug) {
    BuddyAllocator *allocator = (BuddyAllocator
*)malloc(sizeof(BuddyAllocator));
    allocator->data = NULL;
    allocator->head = NULL;
    allocator->tail = NULL;
    allocator->totalSize = 0;
    allocator->expanded = false;
    allocator->debug = debug;

    expand(allocator, size);
    return allocator;
}

void destroyBuddyAllocator(BuddyAllocator *allocator) {
    if (allocator->data) {
        free(allocator->data);
    }
    free(allocator);
```

```
}

void *buddyMalloc(BuddyAllocator *allocator, size_t size) {
    if (size == 0) return NULL;

    size_t actualSize = requiredSize(size);
    BuddyBlock *block = findBest(allocator, actualSize);

    if (!block) {
        if (allocator->expanded) {
            return NULL;
        }
        allocator->expanded = true;
        expand(allocator, actualSize);
        return buddyMalloc(allocator, size);
    }

    allocator->expanded = false;
    return (void *)((uint8_t *)block + sizeof(BuddyBlock));
}

void buddyFree(BuddyAllocator *allocator, void *ptr) {
    if (!ptr) return;

    BuddyBlock *block = (BuddyBlock *)((uint8_t *)ptr - sizeof(BuddyBlock));
    if ((uint8_t *)block < (uint8_t *)allocator->data ||
        (uint8_t *)block >= (uint8_t *)allocator->tail) {
        //fprintf(stderr, "Invalid pointer passed to buddyFree.\n");
        return;
```

```
    }

    block->isFree = true;

    if (allocator->debug) {
        printf("Freed %zu bytes\n", block->blockSize - sizeof(BuddyBlock));
    }

    coalesce(allocator);
}

// ------------------- Тестирование -------------------
#define MEMORY_POOL_SIZE 2048*2048*10

void testListAllocator() {
    void* memory_pool = malloc(MEMORY_POOL_SIZE);
    ListAllocator* list_allocator = createListAllocator(memory_pool,
MEMORY_POOL_SIZE);

    const int NUM_ALLOCS = 100000;
    void* blocks[NUM_ALLOCS];

    // Измеряем время выделения памяти
    auto start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < NUM_ALLOCS/2; ++i) {
        blocks[i] = listAlloc(list_allocator, 25);
        if (blocks[i] == NULL) {
            std::cerr << "Failed to allocate block #" << i << std::endl;
        }
```

```
    }
    for (int i = NUM_ALLOCS/2; i < NUM_ALLOCS; ++i) {
        blocks[i] = listAlloc(list_allocator, 367);
        if (blocks[i] == NULL) {
            std::cerr << "Failed to allocate block #" << i << std::endl;
        }
    }
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> alloc_duration = end - start;
    std::cout << "ListAllocator: Time for allocations: " << alloc_duration.count() <<
" seconds" << std::endl;

    // Измеряем время освобождения памяти
    start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < NUM_ALLOCS; ++i) {
        listFree(list_allocator, blocks[i]);
    }
    end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> free_duration = end - start;
    std::cout << "ListAllocator: Time for frees: " << free_duration.count() << "
seconds" << std::endl;

    destroyListAllocator(list_allocator);
    free(memory_pool);
}

void testBuddyAllocator() {
    BuddyAllocator *allocator = createBuddyAllocator(MEMORY_POOL_SIZE,
false);
```

```
    const int NUM_ALLOCS = 100000;
    void* blocks[NUM_ALLOCS];

    // Измеряем время выделения памяти
    auto start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < NUM_ALLOCS/2; ++i) {
        blocks[i] = buddyMalloc(allocator, 25);
        if (blocks[i] == NULL) {
            std::cerr << "Failed to allocate block #" << i << std::endl;
        }
    }
    for (int i = NUM_ALLOCS/2; i < NUM_ALLOCS; ++i) {
        blocks[i] = buddyMalloc(allocator, 367);
        if (blocks[i] == NULL) {
            std::cerr << "Failed to allocate block #" << i << std::endl;
        }
    }
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> alloc_duration = end - start;
    std::cout << "BuddyAllocator: Time for allocations: " << alloc_duration.count()
<< " seconds" << std::endl;

    // Измеряем время освобождения памяти
    start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < NUM_ALLOCS; ++i) {
        buddyFree(allocator, blocks[i]);
    }
    end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> free_duration = end - start;
```

```
    std::cout << "BuddyAllocator: Time for frees: " << free_duration.count() << "
seconds" << std::endl;

    destroyBuddyAllocator(allocator);
}

int main() {
    std::cout << "Starting performance tests...\n";

    // Тестирование ListAllocator
    testListAllocator();

    // Тестирование BuddyAllocator
    testBuddyAllocator();

    std::cout << "Performance tests completed." << std::endl;

    return 0;
}
```

Демонстрация работы программы

./a.out

```
==27330== Memcheck, a memory error detector
==27330== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==27330== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==27330== Command: ./a.out
==27330==
Starting performance tests...
ListAllocator: Time for allocations: 81.6395 seconds
ListAllocator: Time for frees: 85.0911 seconds
BuddyAllocator: Time for allocations: 173.03 seconds
BuddyAllocator: Time for frees: 332.167 seconds
Performance tests completed.
```

```
==27330==
==27330== HEAP SUMMARY:
==27330== in use at exit: 0 bytes in 0 blocks
==27330== total heap usage: 6 allocs, 6 frees, 109,125,688 bytes allocated
==27330==
==27330== All heap blocks were freed -- no leaks are possible
==27330==
==27330== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Выводы

List Allocator использует связанный список свободных блоков произвольного размера: при выделении памяти ищет подходящий блок (делит его при необходимости), при освобождении объединяет соседние свободные блоки. Гибкий, но медленный из-за обхода списка и склонный к фрагментации. Buddy Allocator оперирует блоками-степенями двойки: разделяет большие блоки пополам для выделения памяти и объединяет освобожденные блоки с «двойниками». Быстрый и устойчивый к фрагментации, но неэффективен для мелких/нестандартных запросов из-за внутренней фрагментации. List выигрывает в задачах с массовым выделением одинаковых блоков, Buddy — в сценариях с частым перераспределением больших объемов памяти, где критична минимизация внешней фрагментации.