Московский Авиационный Институт (Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики Кафедра вычислительной математики и программирования

# Курсовой проект по курсу

**«Операционные системы»**

Студент: Магомедов Э.Х

Группа: М8О-209Б-23

Вариант: 21

Преподаватель: Миронов Евгений Сергеевич

Оценка: Дата: 10.02.2025 Подпись:

Москва, 2024

# Содержание

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Сборка программы
7. Демонстрация работы программы
8. Выводы

# Репозиторий

https://github.com/HGRaicer/MAI\_OS/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.

Сравнение алгоритмов аллокаторов памяти.

**Вариант №21:** 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 (List Allocator)**

Аллокатор списка свободных блоков использует структуру данных, представляющую собой связанный список. Каждый элемент этого списка — это блок памяти, который содержит информацию о своем размере, флаге занятости и указателе на следующий блок. Когда требуется выделить память, аллокатор проходит по списку блоков, ищет подходящий свободный блок и выделяет необходимое количество памяти. Если блок слишком большой, он может быть разделен на несколько меньших блоков. Когда память освобождается, блок помечается как свободный, и, если возможно, происходит слияние с соседними свободными блоками, чтобы избежать фрагментации памяти.

Преимущества:

Простота реализации.

Гибкость в плане размера выделяемых блоков.

Недостатки:

Проблемы с производительностью при большом количестве блоков.

Риск фрагментации памяти при частом выделении и освобождении блоков разного размера.

**Аллокатор двойников (Buddy Allocator)**

Аллокатор двойников работает немного иначе. Он делит память на блоки фиксированного размера, которые всегда являются степенью двойки (например, 64, 128, 256 и т. д.). Аллокатор поддерживает несколько списков свободных блоков, каждый из которых соответствует определенному размеру блока. Когда требуется выделить блок памяти, аллокатор ищет первый подходящий блок. Если в списке для этого размера нет свободного блока, происходит разделение блока большего размера на два меньших, пока не будет найден блок, подходящий для запроса.

Когда блок освобождается, аллокатор пытается объединить его с соседним блоком того же размера (если он также свободен), чтобы уменьшить фрагментацию памяти. Это объединение происходит по принципу "друзей" — блоки считаются соседними, если они имеют одинаковый размер и находятся друг с другом в памяти.

Преимущества:

Большая эффективность благодаря структуре.

Минимизация фрагментации памяти.

Недостатки:

Ограничен только блоками, размерами которых являются степени двойки.

Может быть недостаточно гибким в некоторых случаях, когда требуется более гибкое распределение памяти.

Сравнение

**Гибкость:** Аллокатор списка свободных блоков более гибок в плане размера выделяемых блоков, тогда как аллокатор двойников ограничен блоками, размеры которых являются степенями двойки.

**Производительность**: Аллокатор двойников обычно более эффективен и минимизирует фрагментацию памяти, но может быть менее гибким.

**Фрагментация**: Оба аллокатора стремятся минимизировать фрагментацию памяти, но аллокатор двойников делает это более эффективно благодаря своей структуре.

Выбор между этими аллокаторами зависит от конкретных требований вашей системы и задач, которые она решает.

# Исходный код

#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

rai@rai-laptop:~/cods/os/kp$ valgrind -s ./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

Разные аллокаторы предлагают методы для динамического управления памятью, но их подходы различаются. Аллокатор списка свободных блоков более гибок, но менее эффективен, так как требует прохода по списку и может вызывать фрагментацию. В свою очередь, аллокатор двойников обеспечивает большую эффективность, но ограничен размерами блоков и может приводить к внутренней фрагментации, если размер блока превышает необходимый.

При тестировании было установлено, что аллокатор двойников демонстрирует худшую производительность в задачах с простыми и массовыми операциями выделения и освобождения памяти. Его преимущества проявляются только при частом перераспределении больших объемов памяти, где важно избегать фрагментации. Для задач с однотипными операциями, такими как выделение блоков одинакового размера, аллокатор списка свободных блоков остается более эффективным.