Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Курсовая работа**

**по курсу "Операционные системы"**

**3 семестр**

**Задание 19**

**Сравнение алгоритмов аллокации памяти**

Студент: Фролов М.А.

Группа: М8О–206Б–20

Вариант: 41

Преподаватель: Соколов Андрей Алексеевич

Оценка: 5

Дата: 29.12.21

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2021

**Постановка задачи**

Задание: Необходимо реализовать два алгоритма аллокации памяти и сравнить их.

Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям free и malloc. Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра. В отчёте необходимо отобразить следующее:

• Подробное описание каждого из исследуемых алгоритмов.

• Процесс тестирования.

• Обоснование подхода тестирования.

• Результат тестирования.

• Заключение по проведённой работе.

Каждый аллокатор должен обладать следующим интерфейсом:

• Allocator\* createMemoryAllocator(void\* realMemory, size\_t memory\_size) - создание аллокатор памяти размера memory\_size.

• void\* alloc(Allocator\* allocator, size\_t block\_size) - выделение памяти при помощи

аллокатора размера block\_size.

• void\* free(Allocator\* allocator, void\* block) - возвращает выделенную память аллокатору.

Вариант 19: сравнить алгоритм аллокации на списках свободных блоков и на блоках по 2 в степени n.

**Общая информация об аллокаторах**

Аллокатор — специализированный класс, реализующий и инкапсулирующий малозначимые (с прикладной точки зрения) детали распределения и освобождения ресурсов компьютерной памяти.

Стандартные функции malloc и free на самом деле имеют много проблем:

• Выделение нескольких байтов с помощью malloc происходит точно так же, как и выделение нескольких мегабайтов. В расчёт не берётся информация о том, что это за данные, где они будут располагаться и какой у них будет цикл жизни.

• Выделение памяти при помощи стандартных библиотечных функций или операторов обычно требует обращений к ядру операционной системы. Это может сказываться на производительности приложения.

• Они приводят к фрагментации кучи — состоянию, при котором информация в памяти разбросана в разных, не идущих последовательно блоках, из—за чего даже при достаточном суммарном объёме памяти возможна такая ситуация, что выделить блок в памяти для размещения информации будет невозможно.

• Плохая локальность указателей. Нет никакого способа узнать, какое именно место в память выделит вам malloc. Это может привести к тому, что будет происходить больше дорогостоящих промахов в кэше.

**Аллокатор на списках свободных блоков**

Суть аллокатора на списках свободных блоков заключается в том, что при каждом запросе на выделение памяти из доступной аллокатору памяти выделяется блока запрашиваемого размера (если это возможно). В конечном итоге вся память аллокатора будет разделена на список подряд идущих блоков, некоторые из которых будут заняты, а некоторые свободны.

Вся память аллокатора разделена на блоки, каждый из которых содержит заголовок. В заголовке представлена информация о размере этого блока, размере предыдущего блока и логическая переменная, показывающая свободен ли блок. При создании такого аллокатора сразу выделяется память запрашиваемого размера, в которой создаётся один блок. При запросе на выделение памяти определённого размера ищется первый подходящий свободный блок, из которого выделяется блок запрашиваемого размера. При деаллокации ранее выделенной памяти сначала проверяется валидность переданного указателя – если пользователь хочет его вернуть аллокатору, значит этот указатель должен был быть когда-то этим же аллокатором выдан. После просто в заголовке меняем информацию о том, что блок доступен для использования.

Важной деталью этого аллокатора является возможность дефрагментации. При деаллокации определённого блока памяти происходит проверка доступности соседних блоков — если какой-то из этих блоков доступен, то происходит слияние текущего блока со свободным соседним.

**Аллокатор на блоках размером 2n**

Принцип работы аллокатора на блоках размером 2n аналогичен вышеописанному аллокатору на списках свободных блоков за тем исключением, что при создании аллокатора или при запросе на выделение памяти размер блоков выравнивается под ближайшую большую степень числа 2. Таким образом все используемые в данный момент блоки обязательно будут иметь размер 2n. Такой алгоритм довольно близок к тому, что используется в стандартном системном аллокаторе, ведь он тоже оперирует с блоками,

размер которых является степенью числа 2.

**Основные файлы программы:**

Allocator.cpp:

#include <iostream>

#include "Allocator.h"

using namespace std;

Allocator::Allocator(const size\_t total\_size) : total\_size { total\_size }, used\_size { 0 }{}

Allocator::~Allocator() {

free(buffer\_begin);

}

void Allocator::split(Header \*header, size\_t chunk) {

//cout << chunk << endl;

size\_t block\_size = header->size;

header->size = chunk;

header->is\_free = false;

if (block\_size - chunk >= sizeof(Header)) {

auto \*next = header->next();

next->size = block\_size - chunk - sizeof(Header);

//cout << "()()" << next->size << endl;

next->is\_free = true;

used\_size += chunk + sizeof(Header);

auto \*next\_next = next->next();

} else {

header->size = block\_size;

used\_size += block\_size;

}

//cout << endl;

}

bool Allocator::validate(void \*ptr) {

auto \*header = static\_cast<Header \*>(buffer\_begin);

while (header < buffer\_end) {

if (header + 1 == ptr){ return true; }

header = header->next();

}

return false;

}

Allocator::Header \*Allocator::find\_block(size\_t size) {

auto \*header = static\_cast<Allocator::Header \*>(buffer\_begin);

while (!header->is\_free || header->size < size) {

header = header->next();

if (header >= buffer\_end) { return nullptr; }

}

return header;

}

Allocator.h:

#ifndef ALLOCATOR\_H

#define ALLOCATOR\_H

#include <iostream>

using namespace std;

class Allocator {

public:

void\* buffer\_begin = nullptr;

void\* buffer\_end = nullptr;

size\_t total\_size = 0;

size\_t used\_size = 0;

size\_t peak\_size = 0;

Allocator() = default;

Allocator(const size\_t total\_size);

~Allocator();

struct Header {

public:

size\_t size;

bool is\_free;

Header \*next() {

return (Header \*) ((char \*) (this + 1) + size);

}

};

virtual void\* allocate(size\_t size) = 0;

virtual void deallocate(void\* ptr) = 0;

Header \*find\_block(size\_t size);

void split(Header \*header, size\_t chunk);

bool validate(void \*ptr);

virtual void reset() = 0;

};

#endif

Benchmark.cpp:

#include <iostream>

#include <chrono>

#include <stdlib.h> /\* srand, rand \*/

#include <cassert>

#include "Benchmark.h"

using namespace std;

void Benchmark::SingleAllocation(Allocator\* allocator, size\_t size) {

cout << "BENCHMARK: ALLOCATION" << endl;

cout << "\tSize: \t" << size << endl;

StartRound();

auto operations = 0u;

while (operations < operationGoal) {

allocator->allocate(size);

++operations;

}

FinishRound();

BenchmarkResults results = buildResults(operationGoal, move(TimeElapsed), allocator->peak\_size);

PrintResults(results);

}

void Benchmark::SingleFree(Allocator\* allocator, size\_t size) {

cout << "BENCHMARK: ALLOCATION/FREE" << endl;

cout << "\tSize: \t" << size << endl;

void\* addresses[OPERATIONS];

StartRound();

auto operations = 0u;

while (operations < operationGoal) {

addresses[operations] = allocator->allocate(size);

++operations;

}

while (operations) {

allocator->deallocate(addresses[--operations]);

}

FinishRound();

BenchmarkResults results = buildResults(operationGoal, move(TimeElapsed), allocator->peak\_size);

PrintResults(results);

}

void Benchmark::MultipleAllocation(Allocator\* allocator, vector<size\_t>& allocationSizes) {

for (auto i = 0u; i < allocationSizes.size(); ++i) {

SingleAllocation(allocator, allocationSizes[i]);

}

}

void Benchmark::MultipleFree(Allocator\* allocator, vector<size\_t>& allocationSizes) {

for (auto i = 0u; i < allocationSizes.size(); ++i) {

SingleFree(allocator, allocationSizes[i]);

}

}

void Benchmark::RandomAllocation(Allocator\* allocator, vector<size\_t>& allocationSizes) {

srand(1);

cout << "BENCHMARK: ALLOCATION" << endl;

StartRound();

allocator->reset();

size\_t allocation\_size;

auto operations = 0u;

while (operations < operationGoal) {

this->RandomAllocationAttr(allocationSizes, allocation\_size);

allocator->allocate(allocation\_size);

++operations;

}

FinishRound();

BenchmarkResults results = buildResults(operationGoal, move(TimeElapsed), allocator->peak\_size);

PrintResults(results);

}

void Benchmark::RandomFree(Allocator\* allocator, vector<size\_t>& allocationSizes) {

srand(1);

cout << "BENCHMARK: ALLOCATION/FREE" << endl;

StartRound();

allocator->reset();

void\* addresses[OPERATIONS];

size\_t allocation\_size;

auto operations = 0u;

while (operations < operationGoal) {

this->RandomAllocationAttr(allocationSizes, allocation\_size);

addresses[operations] = allocator->allocate(allocation\_size);

++operations;

}

while (operations) {

allocator->deallocate(addresses[--operations]);

}

FinishRound();

BenchmarkResults results = buildResults(operationGoal, move(TimeElapsed), allocator->peak\_size);

PrintResults(results);

}

void Benchmark::PrintResults(BenchmarkResults& results) {

cout << "\tRESULTS:" << endl;

cout << "\t\tOperations: \t" << results.Operations << endl;

cout << "\t\tTime elapsed: \t" << results.Milliseconds.count() << " ms" << endl;

cout << "\t\tOp per sec: \t" << results.OperationsPerSec << " ops/ms" << endl;

cout << "\t\tTimer per op: \t" << results.TimePerOperation << " ms/ops" << endl;

cout << "\t\tMemory peak: \t" << results.MemoryPeak << " bytes" << endl;

cout << endl;

}

BenchmarkResults Benchmark::buildResults(size\_t newOperationGoal, chrono::milliseconds&& elapsedTime, size\_t memoryPeak) {

BenchmarkResults results;

results.Operations = newOperationGoal;

results.Milliseconds = move(elapsedTime);

results.OperationsPerSec = results.Operations / static\_cast<double>(results.Milliseconds.count());

results.TimePerOperation = static\_cast<double>(results.Milliseconds.count()) / static\_cast<double>(results.Operations);

results.MemoryPeak = memoryPeak;

return results;

}

void Benchmark::RandomAllocationAttr(vector<size\_t>& allocationSizes, size\_t & size) {

int r = rand() % allocationSizes.size();

size = allocationSizes[r];

Benchmark.h:

#ifndef BENCHMARK\_H

#define BENCHMARK\_H

#include <time.h> // timespec

#include <cstddef> // size\_t

#include <chrono>

#include <ratio>

#include <vector>

#include "Allocator.h" // base class allocator

using namespace std;

#define OPERATIONS 1000

struct BenchmarkResults

{

size\_t Operations;

chrono::milliseconds Milliseconds;

double OperationsPerSec;

double TimePerOperation;

size\_t MemoryPeak;

};

class Benchmark {

public:

Benchmark() = delete;

Benchmark(unsigned int newOperationGoal) : operationGoal { newOperationGoal } { }

void SingleAllocation(Allocator\* allocator, size\_t size);

void SingleFree(Allocator\* allocator, size\_t size);

void MultipleAllocation(Allocator\* allocator, vector<size\_t>& allocationSizes);

void MultipleFree(Allocator\* allocator, vector<size\_t>& allocationSizes);

void RandomAllocation(Allocator\* allocator, vector<size\_t>& allocationSizes);

void RandomFree(Allocator\* allocator, vector<size\_t>& allocationSizes);

private:

void PrintResults(BenchmarkResults& results);

void RandomAllocationAttr(vector<size\_t>& allocationSizes, size\_t & size);

BenchmarkResults buildResults(size\_t newOperationGoal, chrono::milliseconds&& ellapsedTime, size\_t memoryUsed);

void SetStartTime() noexcept { Start = chrono::high\_resolution\_clock::now(); }

void SetFinishTime() noexcept { Finish = chrono::high\_resolution\_clock::now(); }

void SetElapsedTime() noexcept { TimeElapsed = chrono::duration\_cast<chrono::milliseconds>(Finish - Start); }

void StartRound() noexcept { SetStartTime(); }

void FinishRound() noexcept

{

SetFinishTime();

SetElapsedTime();

}

private:

size\_t operationGoal;

chrono::time\_point<chrono::high\_resolution\_clock> Start;

chrono::time\_point<chrono::high\_resolution\_clock> Finish;

chrono::milliseconds TimeElapsed;

};

#endif

BinaryAllocator.h:

#ifndef BINARYALLOCATOR\_H

#define BINARYALLOCATOR\_H

#include <cmath>

#include <unistd.h>

#include <stdlib.h>

#include "Allocator.h"

class BinaryAllocator : public Allocator {

public:

static size\_t align(size\_t size);

BinaryAllocator(size\_t new\_size);

void\* allocate(size\_t new\_size) override;

void deallocate(void\* ptr) override;

void reset();

};

#endif

BinaryAllocator.cpp:

#include <cmath>

#include <unistd.h>

#include <stdlib.h>

#include "BinaryAllocator.h"

using namespace std;

size\_t BinaryAllocator::align(size\_t size) { // closest next power of 2

size\_t i = 1;

size\_t k = 0;

while (i < size){

i \*= 2; ++k;

}

return i / 2;

}

BinaryAllocator::BinaryAllocator(size\_t new\_size) {

if (buffer\_begin != nullptr) {

free(buffer\_begin);

buffer\_begin = nullptr;

}

total\_size = align(new\_size) \* 2;

buffer\_begin = malloc(total\_size);

if (buffer\_begin == nullptr) throw runtime\_error("Failed to allocate memory.");

buffer\_end = (void\*)((char\*)(buffer\_begin) + total\_size);

auto \*header = (Header \*) buffer\_begin;

header->is\_free = true;

header->size = total\_size - sizeof(Header);

used\_size = sizeof(Header);

peak\_size = used\_size;

}

void\* BinaryAllocator::allocate(size\_t new\_size) {

if (new\_size == 0) throw runtime\_error("Size must be bigger than 0.");

size\_t size = align(new\_size) \* 2;

if (size > total\_size - used\_size) throw runtime\_error("Not enough memory.");

auto \*header = find\_block(size);

if (header == nullptr) throw runtime\_error("No required blocks.");

split(header, size);

used\_size += size;

peak\_size += size;

return header + 1; //find\_block returns header\*

}

void BinaryAllocator::deallocate(void\* ptr) {

if (!validate(ptr)) return; // if address is not valid

auto \*header = (Header\*)ptr - 1;

header->is\_free = true;

used\_size -= header->size;

}

void BinaryAllocator::reset() {

if (buffer\_begin != nullptr) {

free(buffer\_begin);

buffer\_begin = nullptr;

}

buffer\_begin = malloc(total\_size);

if (buffer\_begin == nullptr) throw runtime\_error("Failed to allocate memory.");

buffer\_end = (void\*)((char\*)(buffer\_begin) + total\_size);

auto \*header = (Header \*) buffer\_begin;

header->is\_free = true;

header->size = total\_size - sizeof(Header);

used\_size = sizeof(Header);

peak\_size = used\_size;

}

FreeListAllocator.cpp:

#include "FreeListAllocator.h"

#include <stdlib.h>

#include <algorithm>

using namespace std;

FreeListAllocator::FreeListAllocator(size\_t size) {

if (buffer\_begin != nullptr) {

free(buffer\_begin);

buffer\_begin = nullptr;

}

buffer\_begin = malloc(size);

if (buffer\_begin == nullptr) throw runtime\_error("Failed to allocate memory.");

total\_size = size;

buffer\_end = (void\*)((char\*)(buffer\_begin) + total\_size);

auto \*header = (Header \*) buffer\_begin;

header->is\_free = true;

header->size = (total\_size - sizeof(Header));

used\_size = sizeof(Header);

peak\_size = used\_size;

}

void\* FreeListAllocator::allocate(std::size\_t new\_size) {

if (new\_size == 0) throw std::runtime\_error("Size must be bigger than 0.");

if (new\_size > total\_size - used\_size) throw std::runtime\_error("Not enough memory.");

auto \*header = find\_block(new\_size);

if (header == nullptr) throw std::runtime\_error("No required blocks.");

split(header, new\_size);

peak\_size += new\_size;

return header + 1; //find\_block returns header\*

}

void FreeListAllocator::deallocate(void\* ptr) {

if (!validate(ptr)) return; // if address is not valid

auto \*header = (Header\*)ptr - 1;

header->is\_free = true;

used\_size -= header->size;

merge(header);

}

void FreeListAllocator::reset() {

if (buffer\_begin != nullptr) {

free(buffer\_begin);

buffer\_begin = nullptr;

}

buffer\_begin = malloc(total\_size);

if (buffer\_begin == nullptr) throw runtime\_error("Failed to allocate memory.");

buffer\_end = (void\*)((char\*)(buffer\_begin) + total\_size);

auto \*header = (Header \*) buffer\_begin;

header->is\_free = true;

header->size = (total\_size - sizeof(Header));

used\_size = sizeof(Header);

peak\_size = used\_size;

}

void FreeListAllocator::merge(Header \*header) {

auto \*next = header->next();

if (header != buffer\_end && next->is\_free) { // if next free

header->size += sizeof(Header) + header->next()->size;

used\_size -= sizeof(Header);

}

}

FreeListAllocator.h:

#ifndef FREEBLOCKSALLOCATOR\_H

#define FREEBLOCKSALLOCATOR\_H

#include "Allocator.h"

class FreeListAllocator : public Allocator {

public:

FreeListAllocator(size\_t size);

void\* allocate(std::size\_t new\_size) override;

void deallocate(void\* ptr) override;

void reset();

private:

void merge(Header \*header);

};

#endif

Main.cpp:

#include <iostream>

#include <vector>

#include "FreeListAllocator.h"

#include "Benchmark.h"

#include "BinaryAllocator.h"

using namespace std;

/\*\*

\* Вариант 13

\* Сравнить 2 алгоритма аллокации: списки свободных блоков и блоки по 2 в степени n.

\* Каждый аллокатор должен обладать следующим интерфейсом:

\* Allocator\* createMemoryAllocator(void\* realMemory, size\_t memorySize) - создание аллокатора

\* void\* alloc(Allocator\* allocator, size\_t block\_size) - выделение памяти

\* void\* free(Allocator\* allocator, void\* block) - возвращение выделенной памяти

\*/

int main(){

try{

size\_t size = 1e8;

vector<std::size\_t> allocations {32, 64, 256, 512, 1024, 2048, 4096};

Allocator \* freeListAllocator = new FreeListAllocator(size);

Benchmark benchmark(OPERATIONS);

cout << "FreeListAllocator:" << endl;

benchmark.MultipleAllocation(freeListAllocator, allocations);

benchmark.MultipleFree(freeListAllocator, allocations);

benchmark.RandomAllocation(freeListAllocator, allocations);

benchmark.RandomFree(freeListAllocator, allocations);

} catch(runtime\_error& err){

cout << err.what() << endl;

}

try{

size\_t size = 1e8;

vector<std::size\_t> allocations {32, 64, 256, 512, 1024, 2048, 4096};

Allocator \* binaryAllocator = new BinaryAllocator(size);

Benchmark benchmark(OPERATIONS);

cout << "BinaryAllocator:" << endl;

benchmark.MultipleAllocation(binaryAllocator, allocations);

benchmark.MultipleFree(binaryAllocator, allocations);

benchmark.RandomAllocation(binaryAllocator, allocations);

benchmark.RandomFree(binaryAllocator, allocations);

} catch(runtime\_error& err){

cout << err.what() << endl;

}

return 0;

}

**Сравнение алгоритмов аллокации**

Для сравнения алгоритмов аллокации будем замерять общее время работы аллокатора по аллокации / деаллокации блоков памяти. Для того, чтобы значения были более наглядными, тестовый файл содержит 4000 различных запросов на аллокацию / деаллокацию. На основе тестирования были получены следующие показатели:

• Аллокатор на списке свободных блоков с дефрагментацией: 96 мс.

• Аллокатор на списке свободных блоков без дефрагментации: 100 мс.

• Аллокатор на блоках по 2n без дефрагментации: 91 мс.

**Вывод**

По результатам тестирования видно, что самый худший результат показывает алгоритм аллокации на списках свободных блоков без дефрагментации, а самый лучший – аллокатор на блоках по 2n, причём так же без дефрагментации. Причиной этого является то, что процессор и вся система памяти изначально настроена на работу в бинарной системе, то есть с блоками памяти, размер которых является степенью числа 2. При аллокации блока памяти, размер которого отличается от степени 2, система всё равно приводит его к такому виду.

В случае с аллокатором на блоках по 2n мы сразу работаем с "более удобными" для процессора блоками. Причина того, почему лучше всего работает алгоритм без дефрагментации также проста — при выбранной реализации аллокатора в общем случае невозможно провести дефрагментацию двух соседних блоков памяти так, чтобы размер полученного блока памяти всё равно был степенью числа 2 из-за того, что вместе с каждым блоком памяти хранится его заголовок, который при слиянии становится доступной памятью. То есть при слиянии двух блоков размер полученного блока равен s. В общем случае s не будет являться степенью числа 2 (хоть это и возможно).