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: Никитцев А.В.

Преподаватель: Миронов Е.С.

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

Дата: 24.12.24

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.

**Подробное описание каждого из исследуемых алгоритмов**

* **Алгоритм двойников.** При аллокации выделяется блок памяти размера 2^n где n – двоичный логарифм запрашиваемого размера, округленного вверх. Сначала аллокатор определяет размер блока, который нужно отдать, потом он ищет среди блоков данного размера свободный. Если блок был найден, то он резервирует его и возвращает указатель на начало блока, иначе он начинает искать на уровнях выше (размеры 2^(n + 1) и т.д. до размера страницы). Если там был найден блок, то он резервируется и делится пополам. Деление продолжается пока мы не получим блок нужного размера, который потом резервируем и аналогично отдаем. Таким образом мы всегда берем только 1 блок, а второй разделенный остается свободным. Если же блок не был найден, значит требуется аллоцировать новую страницу и повторить операцию разделения блоков. При освобождении блока мы ищем его, очищаем и делаем его свободным. После этого мы проверяем не является ли двойник свободным. Если да, то мы объединяем блок, удаляя более мелкие и переходим к родительскому блоку и освобождаем его. Таким образом мы проходим вверх пока либо не найдем неосвобожденного двойника, либо не дойдем до блока размером в страницу.
* **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**

При тестировании теоретически получены данные о том, сколько страниц будет выделено в каждый момент времени, а также в runtime будед посчитан общий теоретический размер объекта и данное значение будет сравнено с фактическим значением.

**Benchmarks**

При проведении проверки перформанса мы последовательно аллоцируем объектов и сравниваем ключевые метрики у каждого алгоритма. По итогу данные выгружаются в файл и потом при помощи сторонних утилит (jupiter notebook, seaborn, pandas, python) строятся графики метрик от количества аллоцируемых объектов. В качестве метрик были взяты суммарное время аллокации и деаллокации n объектов и фактор использования

**Реализация**

* **Двойники**. При реализации алгоритма двойников была использована хэш таблица с константым количеством ключей (4096 – 1 с шагом n / 2), которая группировала блоки по их размеру. Чтобы было легче искать блоки внутри внутри хеш таблицы размеров есть хэш таблица блков, в которой парой значений является идентификатор блока и сам блок. Данная реализация конечно увеличивает количество потребляемой памяти, однако позволяет не использовать связанные структуры данных, что упрощает программу
* **Свободные блоки.** В качестве упрощения в данном алгоритме мы храним не только свободные, но и занятые блоки. Таким образом это облегчит слияние блоков.

**Код программы**

**BaseMemoryResource.h**

#ifndef OS\_CP\_SRC\_BASEMEMORYRESOURCE\_H\_  
#define OS\_CP\_SRC\_BASEMEMORYRESOURCE\_H\_  
  
#include <cstdio>  
#include <memory\_resource>  
#include <memory>  
#include "Utils.h"  
  
namespace Memory {  
 class BaseMemoryResource : public std::pmr::memory\_resource {  
 size\_t totalAllocatedMemory;  
 size\_t totalUsedMemory;  
  
 protected:  
 std::vector<Utils::Page> pages;  
 void setTotalUsedMemory(size\_t val) { totalUsedMemory = val; }  
 void setTotalAllocatedMemory(size\_t val) { totalAllocatedMemory = val; }  
 public:  
 BaseMemoryResource(): pages(), totalUsedMemory(0), totalAllocatedMemory(0) {}  
 size\_t getTotalUsedMemory() const { return totalUsedMemory; }  
 size\_t getTotalAllocatedMemory() const { return totalAllocatedMemory; }  
 virtual void \* do\_allocate(std::size\_t bytes, std::size\_t alignment) override = 0;  
 void do\_deallocate(void \*p, std::size\_t bytes, std::size\_t alignment) override = 0;  
 virtual bool do\_is\_equal(const std::pmr::memory\_resource &other) const noexcept override = 0;  
 ~BaseMemoryResource() override = default;  
 };  
  
} // Memory  
  
#endif //OS\_CP\_SRC\_BASEMEMORYRESOURCE\_H\_

**Benchmark.h**

//  
// Created by nikit on 12/23/2024.  
//  
  
#ifndef OS\_CP\_SRC\_BENCHMARK\_H\_  
#define OS\_CP\_SRC\_BENCHMARK\_H\_  
  
#include <type\_traits>  
#include <cstddef>  
#include "BaseMemoryResource.h"  
namespace Benchmark{  
 template<class T>  
 requires std::is\_default\_constructible\_v<T>  
 size\_t benchmark\_allocator(size\_t iterations, Memory::BaseMemoryResource\* memory\_resource){  
 std::pmr::polymorphic\_allocator<T> allocator(memory\_resource);  
  
 std::vector<T\*> objects;  
 for (auto i = 0; i < iterations; ++i){  
 objects.push\_back(new (allocator.allocate(1)) T());  
 }  
  
 auto usedMem = memory\_resource->getTotalUsedMemory();  
  
 for (auto obj: objects){  
 allocator.deallocate(obj, 1);  
 }  
  
 return usedMem;  
 }  
}  
  
#endif //OS\_CP\_SRC\_BENCHMARK\_H\_

**SimpleBlock.h**

//  
// Created by nikit on 12/22/2024.  
//  
  
#ifndef OS\_CP\_SRC\_SIMPLEBLOCK\_H\_  
#define OS\_CP\_SRC\_SIMPLEBLOCK\_H\_  
  
#include <cstdio>  
namespace Memory {  
  
class SimpleBlock {  
 size\_t start, end, page, sz;  
 bool used;  
  
 public:  
 SimpleBlock() = default;  
 SimpleBlock(size\_t start, size\_t end, size\_t page, bool used) : start(start), end(end), page(page), sz(end - start + 1), used(used) {}  
  
 SimpleBlock(const SimpleBlock& block) = default;  
 SimpleBlock(SimpleBlock&& block) noexcept = default;  
 SimpleBlock& operator=(const SimpleBlock& block) = default;  
 SimpleBlock& operator=(SimpleBlock&& block) noexcept;  
  
 size\_t get\_start() const { return start; }  
 size\_t get\_end() const { return end; }  
 size\_t get\_page() const { return page; }  
 size\_t get\_size() const { return sz; }  
 size\_t get\_used() const { return used; }  
  
 void set\_used(bool val) { used = val; }  
 void set\_start(size\_t val) {  
 start = val;  
 sz = end - start + 1;  
 }  
 void set\_end(size\_t val) {  
 end = val;  
 sz = end - start + 1;  
 }  
  
 ~SimpleBlock() = default;  
  
 bool operator==(const SimpleBlock& other) const;  
 bool operator<(const SimpleBlock& other) const;  
};  
  
} // Memory  
  
#endif //OS\_CP\_SRC\_SIMPLEBLOCK\_H\_

**SimpleBlock.cpp**

//  
// Created by nikit on 12/22/2024.  
//  
  
#include "SimpleBlock.h"  
  
namespace Memory {  
 SimpleBlock &SimpleBlock::operator=(SimpleBlock &&block) noexcept {  
 start = block.start;  
 end = block.end;  
 page = block.page;  
 sz = block.sz;  
 used = block.used;  
 return \*this;  
 }  
  
 bool SimpleBlock::operator==(const SimpleBlock &other) const {  
 return start == other.start && end == other.end && page == other.page ;  
 }  
  
 bool SimpleBlock::operator<(const SimpleBlock &other) const {  
 if (page != other.page) return page < other.page;  
 return start < other.start;  
 }  
} // Memory

**SimpleMemoryResource.h**

//  
// Created by nikit on 12/22/2024.  
//  
  
#ifndef OS\_CP\_SRC\_SIMPLEMEMORYRESOURCE\_H\_  
#define OS\_CP\_SRC\_SIMPLEMEMORYRESOURCE\_H\_  
  
#include <cstdio>  
#include <memory\_resource>  
#include <memory>  
#include <list>  
#include <algorithm>  
#include <iostream>  
#include "SimpleBlock.h"  
#include "BaseMemoryResource.h"  
  
std::tuple<Memory::SimpleBlock, size\_t> findPlaceToInsert(const size\_t sz, const std::vector<Memory::SimpleBlock>& blocks){  
 size\_t idx = 0;  
 for (auto& block: blocks) {  
 if (block.get\_used()) {  
 idx++;  
 continue;  
 }  
  
 if (block.get\_size() >= sz){  
 return {{block.get\_start(), block.get\_start() + sz - 1, block.get\_page(), true}, idx};  
 }  
  
 idx++;  
 }  
 int lastPage = -1;  
 if (!blocks.empty()) lastPage = blocks.back().get\_page();  
 return {{0, sz - 1, static\_cast<size\_t>(lastPage + 1), true}, blocks.size()};  
}  
  
namespace Memory {  
  
 template<size\_t *basePageSize* = 4096>  
 class SimpleMemoryResource : public BaseMemoryResource {  
 std::vector<SimpleBlock> blocks;  
 public:  
 SimpleMemoryResource() : blocks(), BaseMemoryResource() {}  
  
 void \* do\_allocate(std::size\_t bytes, std::size\_t alignment) override {  
 auto [block, emplaceBlockIdx] = findPlaceToInsert(bytes, blocks);  
  
 if (emplaceBlockIdx == blocks.size()){  
 pages.push\_back(Utils::mallocPage(*basePageSize*));  
 blocks.push\_back(block);  
 if (*basePageSize* != bytes) blocks.emplace\_back(block.get\_end() + 1, *basePageSize* - 1, block.get\_page(), false);  
 std::sort(blocks.begin(), blocks.end());  
 setTotalUsedMemory(getTotalUsedMemory() + bytes);  
 setTotalAllocatedMemory(getTotalAllocatedMemory() + *basePageSize*);  
 return pages[block.get\_page()].get() + block.get\_start();  
 }  
  
 if (blocks[emplaceBlockIdx].get\_size() == bytes) {  
 blocks[emplaceBlockIdx].set\_used(true);  
 setTotalUsedMemory(getTotalUsedMemory() + bytes);  
 return pages[block.get\_page()].get() + block.get\_start();  
 }  
  
 blocks[emplaceBlockIdx].set\_start(block.get\_end() + 1);  
 blocks.push\_back(block);  
 std::sort(blocks.begin(), blocks.end());  
 setTotalUsedMemory(getTotalUsedMemory() + bytes);  
 return pages[block.get\_page()].get() + block.get\_start();  
 }  
  
 void do\_deallocate(void \*p, std::size\_t bytes, std::size\_t alignment) override {  
 auto idx = 0;  
 for (auto& block : blocks){  
 if (!block.get\_used()) {  
 idx++;  
 continue;  
 }  
 if (p == pages[block.get\_page()].get() + block.get\_start()){  
 break;  
 }  
 idx++;  
 }  
  
 if (idx == blocks.size() || !blocks[idx].get\_used()) throw std::logic\_error("Can't free unused memory");  
 blocks[idx].set\_used(false);  
  
 if (idx != 0 && blocks[idx].get\_start() != 0 && !blocks[idx - 1].get\_used()) {  
 auto start = blocks[idx - 1].get\_start();  
 blocks[idx].set\_start(start);  
 blocks.erase(std::remove(blocks.begin(), blocks.end(), blocks[idx - 1]), blocks.end());  
 }  
  
 if (idx != blocks.size() - 1 && blocks[idx].get\_end() != *basePageSize* - 1 && !blocks[idx + 1].get\_used()) {  
 auto end = blocks[idx + 1].get\_end();  
 blocks[idx].set\_end(end);  
 blocks.erase(std::remove(blocks.begin(), blocks.end(), blocks[idx + 1]), blocks.end());  
 }  
  
 setTotalUsedMemory(getTotalUsedMemory() - bytes);  
 std::sort(blocks.begin(), blocks.end());  
 }  
  
 bool do\_is\_equal(const std::pmr::memory\_resource &other) const noexcept override {  
 return this == &other;  
 }  
  
 ~SimpleMemoryResource() override = default;  
 };  
  
} // Memory  
  
#endif //OS\_CP\_SRC\_SIMPLEMEMORYRESOURCE\_H\_

**TestClass.h**

//  
// Created by nikit on 12/22/2024.  
//  
  
#ifndef OS\_CP\_SRC\_TESTCLASS\_H\_  
#define OS\_CP\_SRC\_TESTCLASS\_H\_  
  
#include <string>  
#include <vector>  
#include "../src/BaseMemoryResource.h"  
  
namespace TestInternal {  
  
 class TestClass {  
 public:  
 int id;  
 std::string name;  
 std::vector<int> data;  
 size\_t data\_size;  
  
 TestClass() {  
 id = 1;  
 name = "HERE IS MY NAME";  
 data = std::vector<int>({1,2,3,4,5,6,7,8});  
 data\_size = 8;  
 }  
  
  
 ~TestClass() = default;  
 };  
  
} // Test  
  
#endif //OS\_CP\_SRC\_TESTCLASS\_H\_

**TwinBlock.h**

//  
// Created by nikit on 12/22/2024.  
//  
  
#ifndef OS\_CP\_SRC\_TWINBLOCK\_H\_  
#define OS\_CP\_SRC\_TWINBLOCK\_H\_  
  
#include <cstdio>  
#include <optional>  
#include <tuple>  
  
static inline std::string constructIdx(size\_t page, size\_t start, size\_t end) { return std::to\_string(page) + std::to\_string(start) + std::to\_string(end); }  
  
namespace Memory {  
  
 class TwinBlock {  
 size\_t start, end, page;  
 size\_t reservedSize;  
 std::optional<std::string> twinIdx;  
 std::optional<std::string> parentIdx;  
 std::string currentIdx;  
 bool used;  
  
 TwinBlock(  
 size\_t start,  
 size\_t end,  
 size\_t page,  
 std::string twinIdx,  
 std::string parentIdx,  
 bool used  
 ) : start(start),  
 end(end),  
 page(page),  
 reservedSize(end - start + 1),  
 twinIdx(twinIdx),  
 parentIdx(parentIdx),  
 used(used),  
 currentIdx(constructIdx(page, start, end))  
 {}  
  
 public:  
 TwinBlock(  
 size\_t start,  
 size\_t end,  
 size\_t page  
 ) : start(start),  
 end(end),  
 page(page),  
 reservedSize(end - start + 1),  
 twinIdx(std::nullopt),  
 parentIdx(std::nullopt),  
 used(false),  
 currentIdx(constructIdx(page, start, end))  
 {}  
  
 bool getIsUsed() const { return used; }  
 size\_t getStart() const { return start; }  
 size\_t getEnd() const { return end; }  
 size\_t getPage() const { return page; }  
 std::optional<std::string> getTwinIdx() const { return twinIdx; }  
 std::optional<std::string> getParentIdx() const { return parentIdx; }  
 std::string getCurrentIdx() const { return currentIdx; }  
  
 std::optional<std::tuple<TwinBlock, TwinBlock>> split();  
 std::tuple<size\_t, size\_t> use();  
 void free();  
 };  
  
} // Memory  
  
#endif //OS\_CP\_SRC\_TWINBLOCK\_H\_

**TwinBlock.cpp**

//  
// Created by nikit on 12/22/2024.  
//  
  
#include <stdexcept>  
#include "TwinBlock.h"  
  
namespace Memory {  
 std::optional<std::tuple<TwinBlock, TwinBlock>> TwinBlock::split() {  
 if (reservedSize == 1 || used) return std::nullopt;  
 used = true;  
 auto block1Idx = constructIdx(page, start, start + reservedSize / 2 - 1);  
 auto block2Idx = constructIdx(page, start + reservedSize / 2, end);  
 auto block1 = TwinBlock(start, start + reservedSize / 2 - 1, page, block2Idx, currentIdx, false);  
 auto block2 = TwinBlock(start + reservedSize / 2, end, page, block1Idx, currentIdx, false);  
 return std::tuple<TwinBlock, TwinBlock>(block1, block2);  
 }  
  
 void TwinBlock::free() {  
 if (!used) throw std::runtime\_error("Can't free unused block");  
 used = false;  
 }  
  
 std::tuple<size\_t, size\_t> TwinBlock::use() {  
 if (used) throw std::runtime\_error("Can't use already used block");  
 used = true;  
 return {start, page};  
 }  
} // Memory

**TwinsMemoryResource.h**

//  
// Created by nikit on 12/22/2024.  
//  
  
#ifndef OS\_CP\_SRC\_TWINSMEMORYRESOURCE\_H\_  
#define OS\_CP\_SRC\_TWINSMEMORYRESOURCE\_H\_  
  
#include <map>  
#include <list>  
#include "BaseMemoryResource.h"  
#include "cmath"  
#include "TwinBlock.h"  
  
size\_t getBlockSize(size\_t requestedSize) { return static\_cast<size\_t>(std::pow(2, std::ceil(std::log2(requestedSize)))); }  
  
std::optional<std::string> tryFindFreeBlock(const std::unordered\_map<std::string, Memory::TwinBlock>& blocks) {  
 for (const auto& [key, block] : blocks){  
 if (!block.getIsUsed()) return key;  
 }  
 return std::nullopt;  
}  
  
std::string recSplit(  
 std::unordered\_map<size\_t, std::unordered\_map<std::string, Memory::TwinBlock>>& blocks,  
 size\_t curSize,  
 std::string idxToSplit,  
 size\_t targetSize  
) {  
 while (curSize != targetSize){  
 auto [block1, block2] = blocks.at(curSize).at(idxToSplit).split().value();  
 idxToSplit = block1.getCurrentIdx();  
 blocks[curSize / 2].insert({block1.getCurrentIdx(), block1});  
 blocks[curSize / 2].insert({block2.getCurrentIdx(), block2});  
 curSize /= 2;  
 }  
  
 return idxToSplit;  
}  
  
std::optional<std::string> tryFindBlockPostSplit(  
 std::unordered\_map<size\_t, std::unordered\_map<std::string, Memory::TwinBlock>>& blocks,  
 size\_t targetSize,  
 size\_t maxSize  
) {  
 size\_t size = targetSize \* 2;  
 std::optional<std::string> idx = std::nullopt;  
 while (size <= maxSize){  
 auto foundIdx = tryFindFreeBlock(blocks.at(size));  
 if (foundIdx.has\_value()) {  
 idx = foundIdx.value();  
 break;  
 }  
 size \*= 2;  
 }  
  
 if (!idx.has\_value()) return std::nullopt;  
  
 return recSplit(blocks, size, idx.value(), targetSize);  
}  
  
std::string addPageAndGetBlock(  
 std::vector<Utils::Page>& pages,  
 std::unordered\_map<size\_t, std::unordered\_map<std::string, Memory::TwinBlock>>& blocks,  
 size\_t targetSize,  
 size\_t pageSize  
) {  
 size\_t newPageIdx = pages.size();  
 pages.push\_back(Utils::mallocPage(pageSize));  
 auto block = Memory::TwinBlock(0, pageSize, newPageIdx);  
 blocks[pageSize].insert({block.getCurrentIdx(), block});  
  
 return recSplit(blocks, pageSize, block.getCurrentIdx(), targetSize);  
}  
  
  
std::tuple<std::string, std::optional<size\_t>> findBlock(  
 std::vector<Utils::Page>& pages,  
 std::unordered\_map<size\_t, std::unordered\_map<std::string, Memory::TwinBlock>>& blocks,  
 size\_t size,  
 size\_t maxSize  
) {  
 auto existingPreSplitBlockIdx = tryFindFreeBlock(blocks.at(size));  
 if (existingPreSplitBlockIdx.has\_value()) return {existingPreSplitBlockIdx.value(), std::nullopt};  
 auto existingPostSplitBlockIdx = tryFindBlockPostSplit(blocks, size, maxSize);  
 if (existingPostSplitBlockIdx.has\_value()) return {existingPostSplitBlockIdx.value(), std::nullopt};  
 return {addPageAndGetBlock(pages, blocks, size, maxSize), maxSize};  
}  
  
void reCouple(  
 std::unordered\_map<size\_t, std::unordered\_map<std::string, Memory::TwinBlock>>& blocks,  
 std::string blockToFreeIdx,  
 size\_t size,  
 size\_t maxSize  
) {  
 blocks.at(size).at(blockToFreeIdx).free();  
 while (size <= maxSize) {  
 auto curIdx = blocks.at(size).at(blockToFreeIdx).getCurrentIdx();  
 auto twinIdx = blocks.at(size).at(blockToFreeIdx).getTwinIdx();  
 if (!twinIdx.has\_value() || blocks.at(size).at(twinIdx.value()).getIsUsed()) break;  
 blockToFreeIdx = blocks.at(size).at(blockToFreeIdx).getParentIdx().value();  
 blocks.at(size).erase(curIdx);  
 blocks.at(size).erase(twinIdx.value());  
 size \*= 2;  
 blocks.at(size).at(blockToFreeIdx).free();  
 }  
}  
  
namespace Memory {  
  
 template<size\_t basePageSize = 4096>  
 class TwinsMemoryResource : public BaseMemoryResource {  
 std::unordered\_map<size\_t, std::unordered\_map<std::string, TwinBlock>> blocks;  
 public:  
 TwinsMemoryResource() : blocks(), BaseMemoryResource() {  
 for (size\_t size = basePageSize; size >= 1; size /= 2){  
 blocks.insert({size, {}});  
 }  
 }  
  
 void \* do\_allocate(std::size\_t bytes, std::size\_t alignment) override {  
 auto sz = getBlockSize(bytes);  
 if (sz > basePageSize) throw std::runtime\_error("Size exceeds base page size");  
  
 auto [idx, allocatedSize] = findBlock(pages, blocks, sz, basePageSize);  
 if (allocatedSize.has\_value()) setTotalAllocatedMemory(getTotalAllocatedMemory() + allocatedSize.value());  
 auto [start, page] = blocks.at(sz).at(idx).use();  
 setTotalUsedMemory(getTotalUsedMemory() + bytes);  
 return pages[page].get() + start;  
 }  
  
 void do\_deallocate(void \*p, std::size\_t bytes, std::size\_t alignment) override {  
 auto sz = getBlockSize(bytes);  
  
 std::optional<std::string> idx;  
 for (auto& [key, block] : blocks.at(sz)){  
 if (!block.getIsUsed()) continue;  
 if (p == pages[block.getPage()].get() + block.getStart()){  
 idx = key;  
 break;  
 }  
 }  
  
 if (!idx.has\_value()) throw std::runtime\_error("Can't find block to free");  
  
 setTotalUsedMemory(getTotalUsedMemory() - bytes);  
  
 reCouple(blocks, idx.value(), sz, basePageSize);  
 }  
  
 bool do\_is\_equal(const std::pmr::memory\_resource &other) const noexcept override {  
 return this == &other;  
 }  
  
 ~TwinsMemoryResource() override = default;  
 };  
  
} // Memory  
  
#endif //OS\_CP\_SRC\_TWINSMEMORYRESOURCE\_H\_

**Utils.h**

//  
// Created by nikit on 12/22/2024.  
//  
  
#ifndef OS\_CP\_SRC\_UTILS\_H\_  
#define OS\_CP\_SRC\_UTILS\_H\_  
  
#include <functional>  
#include <memory>  
  
namespace Utils {  
 using Page = std::unique\_ptr<char[]>;  
  
 std::tuple<double, size\_t> measure\_time(const std::function<size\_t()>& func);  
 Page mallocPage(size\_t pageSize);  
}  
  
#endif //OS\_CP\_SRC\_UTILS\_H\_

**Utils.cpp**

//  
// Created by nikit on 12/22/2024.  
//  
  
#include <chrono>  
#include "Utils.h"  
  
namespace Utils {  
 std::tuple<double, size\_t> measure\_time(const std::function<size\_t ()>& func) {  
 auto start = std::chrono::high\_resolution\_clock::now();  
 auto res = func();  
 auto end = std::chrono::high\_resolution\_clock::now();  
 std::chrono::duration<double> elapsed = end - start;  
 return {elapsed.count(), res};  
 }  
  
 Page mallocPage(size\_t pageSize) {  
 auto pagePtr = std::unique\_ptr<char[]>(static\_cast<char\*>(malloc(pageSize \* sizeof (char))));  
 if (pagePtr == nullptr) throw std::runtime\_error("Can't allocate new page");  
 return std::move(pagePtr);  
 }  
}

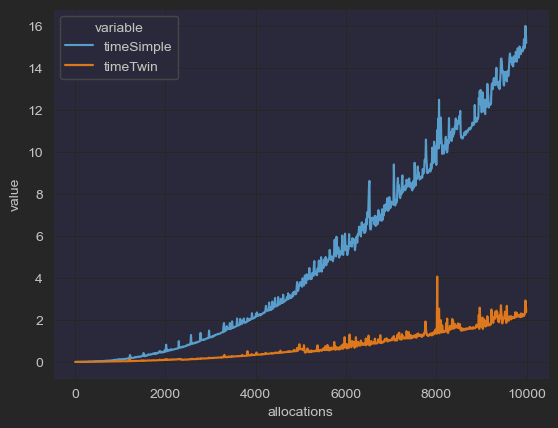
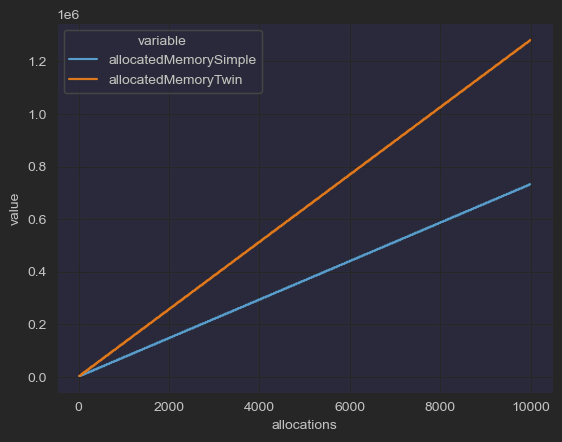
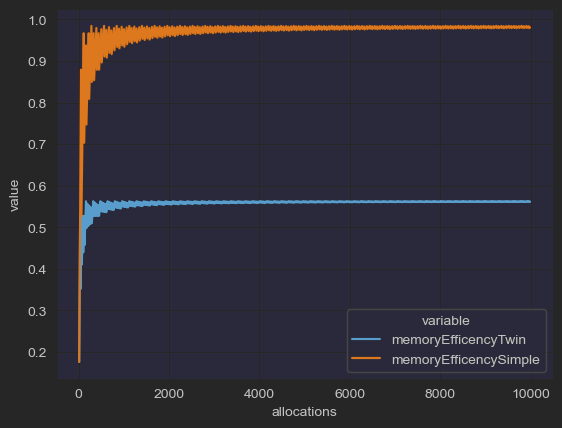
**Test.cpp**

#include <gtest/gtest.h>  
#include "../src/SimpleMemoryResource.h"  
#include "../src/TestClass.h"  
#include "../src/TwinsMemoryResource.h"  
  
TEST(SimpleAllocatorTests, ShouldAllocateAndDeallocateCorrectly){  
 auto memory\_resource = new Memory::SimpleMemoryResource();  
 std::pmr::polymorphic\_allocator<TestInternal::TestClass> allocator(memory\_resource);  
 std::pmr::polymorphic\_allocator<int> allocator\_int(memory\_resource);  
  
 std::vector<TestInternal::TestClass\*> objects;  
 for (auto i = 0; i < 100; ++i){  
 objects.push\_back(new (allocator.allocate(1)) TestInternal::TestClass());  
 }  
  
 EXPECT\_EQ(memory\_resource->getTotalUsedMemory(), 100 \* sizeof (TestInternal::TestClass));  
 EXPECT\_EQ(memory\_resource->getTotalAllocatedMemory(), 2 \* 4096);  
  
 for (auto i = 0; i < objects.size() - 10; ++i){  
 allocator.deallocate(objects[i], 1);  
 }  
  
 EXPECT\_EQ(memory\_resource->getTotalUsedMemory(), 10 \* sizeof (TestInternal::TestClass));  
 EXPECT\_EQ(memory\_resource->getTotalAllocatedMemory(), 2 \* 4096);  
  
 auto val = allocator\_int.allocate(1);  
  
 EXPECT\_EQ(memory\_resource->getTotalUsedMemory(), 10 \* sizeof (TestInternal::TestClass) + sizeof (int));  
 EXPECT\_EQ(memory\_resource->getTotalAllocatedMemory(), 2 \* 4096);  
  
 for (auto i = 0; i < 100 - 10; ++i){  
 objects[i] = new (allocator.allocate(1)) TestInternal::TestClass();  
 }  
  
 EXPECT\_EQ(memory\_resource->getTotalAllocatedMemory(), 2 \* 4096);  
  
 for (auto i = 0; i < 100; ++i){  
 objects.emplace\_back(new (allocator.allocate(1)) TestInternal::TestClass());  
 }  
  
 EXPECT\_EQ(memory\_resource->getTotalUsedMemory(), 200 \* sizeof (TestInternal::TestClass) + sizeof (int));  
 EXPECT\_EQ(memory\_resource->getTotalAllocatedMemory(), 4 \* 4096);  
  
 for (auto obj: objects){  
 allocator.deallocate(obj, 1);  
 }  
  
 allocator\_int.deallocate(val, 1);  
  
 EXPECT\_EQ(memory\_resource->getTotalUsedMemory(), 0);  
 EXPECT\_EQ(memory\_resource->getTotalAllocatedMemory(), 4 \* 4096);  
}  
  
TEST(TwinAllocatorTests, ShouldAllocateAndDeallocateCorrectly){  
 auto memory\_resource = new Memory::TwinsMemoryResource();  
 std::pmr::polymorphic\_allocator<TestInternal::TestClass> allocator(memory\_resource);  
 std::pmr::polymorphic\_allocator<int> allocator\_int(memory\_resource);  
  
 std::vector<TestInternal::TestClass\*> objects;  
 for (auto i = 0; i < 100; ++i){  
 objects.push\_back(new (allocator.allocate(1)) TestInternal::TestClass());  
 }  
  
 EXPECT\_EQ(memory\_resource->getTotalUsedMemory(), 100 \* sizeof (TestInternal::TestClass));  
 EXPECT\_EQ(memory\_resource->getTotalAllocatedMemory(), 4 \* 4096);  
  
 for (auto i = 0; i < objects.size() - 10; ++i){  
 allocator.deallocate(objects[i], 1);  
 }  
  
 EXPECT\_EQ(memory\_resource->getTotalUsedMemory(), 10 \* sizeof (TestInternal::TestClass));  
 EXPECT\_EQ(memory\_resource->getTotalAllocatedMemory(), 4 \* 4096);  
  
 auto val = allocator\_int.allocate(1);  
  
 EXPECT\_EQ(memory\_resource->getTotalUsedMemory(), 10 \* sizeof (TestInternal::TestClass) + sizeof (int));  
 EXPECT\_EQ(memory\_resource->getTotalAllocatedMemory(), 4 \* 4096);  
  
 for (auto i = 0; i < 100 - 10; ++i){  
 objects[i] = new (allocator.allocate(1)) TestInternal::TestClass();  
 }  
  
 for (auto i = 0; i < 100; ++i){  
 objects.emplace\_back(new (allocator.allocate(1)) TestInternal::TestClass());  
 }  
  
 EXPECT\_EQ(memory\_resource->getTotalUsedMemory(), 200 \* sizeof (TestInternal::TestClass) + sizeof (int));  
 EXPECT\_EQ(memory\_resource->getTotalAllocatedMemory(), 7 \* 4096);  
  
 for (auto obj: objects){  
 allocator.deallocate(obj, 1);  
 }  
  
 allocator\_int.deallocate(val, 1);  
  
 EXPECT\_EQ(memory\_resource->getTotalUsedMemory(), 0);  
 EXPECT\_EQ(memory\_resource->getTotalAllocatedMemory(), 7 \* 4096);  
}  
  
  
int main(int argc, char \*\*argv) {  
 testing::InitGoogleTest(&argc, argv);  
 return RUN\_ALL\_TESTS();  
}

**Main.cpp**

#include <iostream>  
#include <fstream>  
#include "src/Utils.h"  
#include "src/TestClass.h"  
#include "src/SimpleMemoryResource.h"  
#include "src/TwinsMemoryResource.h"  
#include "src/Benchmark.h"  
  
void benchmark(size\_t iterations){  
 std::cout << "Running benchmark for " << iterations << " iterations" << std::endl;  
 auto simpleMemRes = new Memory::SimpleMemoryResource();  
 auto twinsMemRes = new Memory::TwinsMemoryResource();  
  
 std::ofstream file;  
 file.open("data.txt", std::ios\_base::app);  
  
 auto [timeSimple, usedMemSimple] = Utils::measure\_time([&simpleMemRes, iterations](){  
 return Benchmark::benchmark\_allocator<TestInternal::TestClass>(iterations, simpleMemRes);  
 });  
  
 auto [timeTwin, usedMemTwin] = Utils::measure\_time([&twinsMemRes, iterations](){  
 return Benchmark::benchmark\_allocator<TestInternal::TestClass>(iterations, twinsMemRes);  
 });  
  
 file << iterations << " ";  
 file << timeSimple << " " << simpleMemRes->getTotalAllocatedMemory() << " " << usedMemSimple << " ";  
 file << timeTwin << " " << twinsMemRes->getTotalAllocatedMemory() << " " << usedMemTwin;  
 file << std::endl;  
}  
  
int main(int argv, char\*\* argc){  
 {  
 std::ofstream file;  
 file.open("data.txt", std::ios\_base::app);  
 file << "allocations timeSimple allocatedMemorySimple usedMemSimple timeTwin allocatedMemoryTwin usedMemTwin" << std::endl;  
 }  
 for (auto i = 10; i < 10000; i+=10){  
 benchmark(i);  
 }  
}

**Итоги**

**** **** 

Проведя исследование, я получил, что алгоритм двойников значительно быстрее выполнял операции аллокации/деаллокации, но в то же время показал себя хуже на факторе использования. Если говорить про время операций, то это понятно. При линейном поиске, сложность будет O(N) в то время как при алгоритме двойников структура хранения сводится к бинарному дереву, сложность поиска в котором уже логарифмическая. По поводу памяти все тоже относительно логично, хотя фактор у алгоритма свободных блоков редко когда настолько близок к 1. В данном тестировании так получилось чисто из-за того, что аллоцируются объекты одного и того же типа последовательно, поэтому объекты плотно прилегают друг к другу, что как раз плохо для алгоритма двойников, если конечно запрашиваемые размеры не всегда степени двойки, так как он всегда аллоцирует блоки размером 2^n и если размер не степень двойки, то есть пустое никем не заполненное место. Таким образом метрика фактора использования могла быть лучше для алгоритма двойников и хуже для алгоритма свободных блоков в зависимости от ситуации.

**Заключение**

Подводя итог, можно сказать, что у каждой стратегии есть свои плюсы и минусы. Например алгоритм свободных блоков прост в реализации относительно алгоритма двойников, однако по перформансу в некоторых метриках от обгоняет свободные блоки.