Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математикиКафедра вычислительной математики и программирования

**Курсовая работа по курсу**
**«Операционные системы»**

# «Аллокаторы памяти»

Студент: Москвин Артём Артурович
Группа: М8О–208Б–20
Вариант: 19
Преподаватель: Миронов Е. С.
Оценка:
Дата: 26 марта 2022 г.

Подпись: _____

Москва, 2022

# Содержание

# Постановка задачи

Необходимо реализовать два алгоритма аллокации памяти и провести их тщательное сравнение по различным характеристикам.

# Сведения о программе

Программа использует библиотеку STL, в частности контейнер vector. Для тестирования применяются функции из заголовочного файла chrono. Программа состоит из трёх h файлов и двух исполняемых файлов main.cpp и main0.cpp. Кроме того, мною написан файл generate.py, который генерирует случайные тесты для программы и записывает их в один файл test.txt. Всего тестов в файле 50000.

# Аллокаторы

**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 кеше.

# 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 знатяы, 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 списках по 2 в степени n

Списки свободной памяти, основанные на степени числа 2, чаще всего применяются для реализации процедур malloc() и free() в библиотеке C прикладного уровня. Методика использует набор списков свободной памяти. В каждом списке хранятся буферы определенного размера. Размер буфера всегда кратен степени числа 2.

Каждый буфер имеет заголовок длиной в одно слово, этим фактом ограничивая возможности использования соотносимой с ним области памяти. Если буфер свободен, в его заголовке хранится указатель на следующий свободный буфер. В другом случае в заголовок буфера помещается указатель на список, в который он должен быть возвращен при освобождении. В некоторых реализациях заголовок содержит вместо этой информации размер выделенной области.

Для выделения памяти клиент вызывает malloc(). В качестве входного аргумента функции передается желаемая величина участка. Распределитель вычисляет минимальный размер буфера, подходящий для удовлетворения запроса. Для этого необходимо прибавить к заданной величине слово, в котором будет размещен заголовок, и округлить полученное значение кверху до числа, являющегося степенью двойки. Буферы размером 32 байта подходят для выделения памяти объемом 0-28 байтов, 64-байтовые буферы — для объемов 29-60 байтов и т. д. После вычисления распределитель извлекает буфер из соответствующего списка и оставляет в заголовке указатель на список свободных участков памяти. Процесс, запрашивающий участок памяти, получает в ответ указатель на следующий после заголовка байт.

# Реализация программы:

# allocator.h

```
#ifndef CP_ALLOCATOR_H
#define CP_ALLOCATOR_H

#include <iostream>

class Allocator {
public:
    typedef void value_type;
    typedef value_type *pointer;
    typedef size_t size_type;

    Allocator() = default;

    ~Allocator() {
        ::free(startPointer);
    }

    /**
     * Выделяет блок памяти заданного размера.
     * @param size - размер блока памяти.
     * @return указатель на выделенный объект памяти или значение null, если память
не была выделена
     */
    virtual pointer allocate(size_type size) = 0;

    /**
     * Освобождает переданный указатель.
     * @param ptr - указатель для освобождения памяти.
     */
    virtual void deallocate(pointer ptr) = 0;

    /**
     Освобождает всю память, выделенную для аллокатора
     */
    void free() {
        auto *header = static_cast<Header *>(startPointer);
        header->isAvailable = true;
        header->size = (totalSize - headerSize);
        usedSize = headerSize;
    };

    /**
     * Выводит информацию о состоянии памяти.
     */
    void memoryDump() {
```

```
        std::cout << "Total size: " << totalSize << std::endl;
        std::cout << "Used: " << usedSize << std::endl;
        std::cout << "Header size: " << headerSize << std::endl;
        auto *header = static_cast<Header *>(startPointer);
        while (header != endPointer) {
            auto isAvailable = header->isAvailable ? "+" : "-";
            std::cout << isAvailable << " " << header << " " << header->size <<
std::endl;
            header = header->next();
        }
        std::cout << std::endl;
    }

protected:

    struct Header {
    public:
        size_type size;
        size_type previousSize;
        bool isAvailable;

        /**
         * @return место в памяти заголовка следующего блока
         */
        inline Header *next() {
            return (Header *) ((char *) (this + 1) + size);
        }

        /**
         * @return место в памяти заголовка предыдущего блока
         */
        inline Header *previous() {
            return (Header *) ((char *) this - previousSize) - 1;
        }

    };

    const size_type headerSize = sizeof(Header);
    const size_type blockAlignment = 4;
    pointer startPointer = nullptr;
    pointer endPointer = nullptr;
    size_type totalSize = 0;
    size_type usedSize = 0;

    /**
     * Находит ближайший свободный блок памяти.
     * @param size - размер запрашиваемого блока.
     * @return указатель на заголовок найденного блока.
     */
    Header *find(size_type size) {
```

```
        auto *header = static_cast<Header *>(startPointer);
        while (!header->isAvailable || header->size < size) {
            header = header->next();
            if (header >= endPointer) { return nullptr; }
        }
        return header;
    }

    /**
     * Разделяет блок на 2 части.
     * @param header - заголовок блока для разделения
     * @param chunk - размер блока, который нужно оставить
     */
    void splitBlock(Header *header, size_type chunk) {
        size_type blockSize = header->size;
        header->size = chunk;
        header->isAvailable = false;
        if (blockSize - chunk >= headerSize) {
            auto *next = header->next();
            next->previousSize = chunk;
            next->size = blockSize - chunk - headerSize;
            next->isAvailable = true;
            usedSize += chunk + headerSize;
            auto *followed = next->next();
            if (followed < endPointer) {
                followed->previousSize = next->size;
            }
        } else {
            header->size = blockSize;
            usedSize += blockSize;
        }
    }

    /**
     * Проверяет валидность переданного адреса.
     */
    bool validateAddress(void *ptr) {
        auto *header = static_cast<Header *>(startPointer);
        while (header < endPointer) {
            if (header + 1 == ptr){ return true; }
            header = header->next();
        }
        return false;
    }

};

#endif //CP_ALLOCATOR_H
```

# binaryAlloctor.h

```
#ifndef CP_BINARYALLOCATOR_H
#define CP_BINARYALLOCATOR_H

#include <cmath>
#include "allocator.h"

class BinaryAllocator : public Allocator {
public:

    explicit BinaryAllocator(size_type size) {
        size = align(size);
        if ((startPointer = malloc(size)) == nullptr) {
            std::cerr << "Failed to allocate memory\n";
            return;
        }
        totalSize = size;
        endPointer = static_cast<void *>(static_cast<char *>(startPointer) +
totalSize);
        auto *header = (Header *) startPointer;
        header->isAvailable = true;
        header->size = (totalSize - headerSize);
        header->previousSize = 0;
        usedSize = headerSize;
    };

    /**
     * Выравнивает размер запрашиваемой памяти до ближайшей степени 2.
     */
    static size_type align(size_type size) {
        int i = 0;
        while (pow(2, i) < size){
            i++;
        }
        return (size_type) pow(2, i);
    }

    /**
     * Выделяет блок памяти заданного размера.
     * @param size - размер блока памяти.
     * @return указатель на выделенный объект памяти или значение null, если память
не была выделена
     */
    pointer allocate(size_type size) override {
        if (size <= 0) {
            std::cerr << "Size must be bigger than 0\n";
            return nullptr;
        }
```

```
        size += sizeof(int);
        size = align(size);
        if (size > totalSize - usedSize) { return nullptr; }
        auto *header = find(size);
        if (header == nullptr) { return nullptr; }
        splitBlock(header, size);
        return header + 1;
    };

    /**
     * Освобождает переданный указатель.
     * @param ptr - указатель для освобождения памяти.
     */
    void deallocate(pointer ptr) override {
        if (!validateAddress(ptr)) {
            return;
        }
        auto *header = static_cast<Header *>(ptr) - 1;
        header->isAvailable = true;
        usedSize -= header->size;
        //defragmentation(header);
    };

private:

    /**
     * Функции проверки доступности близлежащих блоков.
     */
    bool isPreviousFree(Header *header) {
        auto *previous = header->previous();
        return header != startPointer && previous->isAvailable;
    }

    bool isNextFree(Header *header) {
        auto *next = header->next();
        return header != endPointer && next->isAvailable;
    }

    /**
     * Функция слияния свободных блоков.
     */
    void defragmentation(Header *header) {
        if (isPreviousFree(header)) {
            auto *previous = header->previous();
            if (header->next() < endPointer) {
                header->next()->previousSize += previous->size + headerSize;
            }
            previous->size += header->size + headerSize;
            usedSize -= headerSize;
            header = previous;
```

```
        }
        if (isNextFree(header)) {
            header->size += headerSize + header->next()->size;
            usedSize -= headerSize;
            auto *next = header->next();
            if (next != endPointer) { next->previousSize = header->size; }
        }
    }

};

#endif //CP_BINARYALLOCATOR_H
```

# freeBlocksAlloctor.h

```
#ifndef CP_FREEBLOCKSALLOCATOR_H
#define CP_FREEBLOCKSALLOCATOR_H

#include "allocator.h"

class FreeBlocksAllocator : public Allocator {
public:

    explicit FreeBlocksAllocator(size_type size) {
        if ((startPointer = malloc(size)) == nullptr) {
            std::cerr << "Failed to allocate memory\n";
            return;
        }
        totalSize = size;
        endPointer = static_cast<void *>(static_cast<char *>(startPointer) +
totalSize);
        auto *header = (Header *) startPointer;
        header->isAvailable = true;
        header->size = (totalSize - headerSize);
        header->previousSize = 0;
        usedSize = headerSize;
    };

    /**
     * Выделяет блок памяти заданного размера.
     * @param size - размер блока памяти.
     * @return указатель на выделенный объект памяти или значение null, если память
не была выделена
     */
    pointer allocate(size_type size) override {
        if (size <= 0) {
            std::cerr << "Size must be bigger than 0\n";
```

```
            return nullptr;
        }
        size += sizeof(int);
        if (size > totalSize - usedSize) { return nullptr; }
        auto *header = find(size);
        if (header == nullptr) { return nullptr; }
        splitBlock(header, size);
        return header + 1;
    };

    /**
     * Освобождает переданный указатель.
     * @param ptr - указатель для освобождения памяти.
     */
    void deallocate(pointer ptr) override {
        if (!validateAddress(ptr)) {
            return;
        }
        auto *header = static_cast<Header *>(ptr) - 1;
        header->isAvailable = true;
        usedSize -= header->size;
        defragmentation(header);
    };

private:

    /**
     * Функции проверки доступности близлежащих блоков.
     */
    bool isPreviousFree(Header *header) {
        auto *previous = header->previous();
        return header != startPointer && previous->isAvailable;
    }

    bool isNextFree(Header *header) {
        auto *next = header->next();
        return header != endPointer && next->isAvailable;
    }

    /**
     * Функция слияния свободных блоков.
     */
    void defragmentation(Header *header) {
        if (isPreviousFree(header)) {
            auto *previous = header->previous();
            if (header->next() != endPointer) {
                header->next()->previousSize += previous->size + headerSize;
            }
            previous->size += header->size + headerSize;
            usedSize -= headerSize;
```

```
            header = previous;
        }
        if (isNextFree(header)) {
            header->size += headerSize + header->next()->size;
            usedSize -= headerSize;
            auto *next = header->next();
            if (next != endPointer) { next->previousSize = header->size; }
        }
    }

};

#endif //CP_FREEBLOCKSALLOCATOR_H
```

# main.cpp

```
#include <iostream>
#include <vector>
#include "freeBlocksAllocator.h"
#include "binaryAllocator.h"
#include <chrono>



int main(){

    std::chrono::time_point<std::chrono::system_clock> start =
std::chrono::system_clock::now();
    auto allocator = BinaryAllocator(1024000);
    std::vector<void *> pointers;
    char mod;
    while (std::cin >> mod) {
        if (mod == '+') {
            int size;
            std::cin >> size;
            auto ptr = allocator.allocate(size);
            if (ptr) {
                pointers.push_back(ptr);
            }
        }
        if (mod == '-') {
            int index;
            std::cin >> index;
            allocator.deallocate(pointers[index]);
        }
    }
```

```
    std::chrono::time_point<std::chrono::system_clock> finish =
std::chrono::system_clock::now();

    std::chrono::duration<double> difference = finish - start;
    const double ms = difference.count() * 1000;
    std::cout << ms << " ms\n";

}
```

# main0.cpp

```
#include <iostream>
#include <vector>
#include "freeBlocksAllocator.h"
#include "binaryAllocator.h"
#include <chrono>


int main(){

    std::chrono::time_point<std::chrono::system_clock> start =
std::chrono::system_clock::now();
    auto allocator = FreeBlocksAllocator(1024000);
    std::vector<void *> pointers;
    char mod;
    while (std::cin >> mod) {
        if (mod == '+') {
            int size;
            std::cin >> size;
            auto ptr = allocator.allocate(size);
            if (ptr) {
                pointers.push_back(ptr);
            }
        }
        if (mod == '-') {
            int index;
            std::cin >> index;
            allocator.deallocate(pointers[index]);
        }
    }
    std::chrono::time_point<std::chrono::system_clock> finish =
std::chrono::system_clock::now();

    std::chrono::duration<double> difference = finish - start;
    const double ms = difference.count() * 1000;
    std::cout << ms << " ms\n";

}
```

## Тестирование:

Для сравнения алогоритмов аллокации будем замерять общее время работы аллокатора по аллокации. Для того, чтобы значения были более наглядными, тестовый файл содержит 50000 запросов на аллокацию / деаллокацию. На основе тестирования были получены следующие показатели:

Аллокатор на блоках два в степени n: **63.8501 ms, 72.7656 ms,  63.0471 ms**

Аллокатор на списках свободных блоков: **86.5274 ms, 89.5759 ms, 84.3868 ms**

## Вывод:

По результатам тестирования видно, что хуже результат показывает алгоритм аллокации на списках свободных блоков, чем аллокатор на блоках по 2 в степени n. 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.