Московский Авиационный Институт (Национальный Исследовательский институт)

Факультет информационных технологий и прикладной математики Кафедра вычислительной математики и программирования

### Курсовая работа по курсу

**«Операционные системы»**

# «Сравнения алгоритмов аллокаторов памяти»

Группа: М8О-208Б-21

Студент: Иванов Е.Д. Преподаватель: Миронов Е.С. Оценка: Дата:

Москва, 2022

# Содержание

1. Постановка задачи
2. Сведения о программе
3. Аллокаторы памяти
4. Реализации аллокаторов
5. Тестирование
6. Заключение

# Постановка задачи

Необходимо сравнить два алгоритма аллокации: списки свободных блоков (первое подходящее) и алгоритм двойников.

# Сведения о программе

Программа написана на C++, в среде WSL Ubuntu 18.04.3, сборка произведена посредством make-файла. Программа использует библиотеку STL, в частности контейнеры (vector, list), алгоритмы(find\_if). Для тестирования применяются функции из заголовочного файла chrono. Программа состоит из д вух пар cpp/h файлов для каждого аллокатора и тестирующего исполняемого файла.

# Аллокаторы памяти

Операционная система управляет всей доступной физической памятью машины и производит ее выделение для остальных подсистем ядра и прикладных задач. Данной процедурой управляет ядро, оно же и освобождает память, когда это требуется.

Менеджером памяти(аллокатором) называется часть ОС, непосредственно обрабатывающая запросы на выделение и освобождение памяти.

Существуют разные алгоритмы для реализации аллокаторов. Каждый из них имеет свои особенности и недостатки. Согласно моему варианту в данной курсовой работы я реализовал следующие алгоритмы аллокации:

* Алгоритм с выбором первого подходящего участка памяти, основанный на списках
* Алгоритм двойников (Buddy Allocation).

Рассмотрим подробнее алгоритмы их реализации и характеристики.

## Первый подходящий участок(списки)

Этот способ отслеживает память с помощью связных списков распределенных и свободных сегментов памяти, где сегмент содержит либо свободную, либо выделенную память. Каждый элемент списка хранит внутри свое обозначение

— является ли он хранилищем выделенной или освобожденной памяти, а также размер участка памяти и указатель на его начало.

Список поддерживает инвариант отсортированности элементов по адресам с самой инициализации аллокатора. Благодаря этому, упрощается обновление списка при выделении или освобождении памяти. Для таких списков существует 4 алгоритма выделения памяти:

* **Первое подходящее** — список сегментов сканируется, пока не будет найдено пустое пространство подходящего размера. После этого сегмент разбивается на два сегмента, один из которых будет пустым. Данный алгоритм довольно быстр, ведь поиск ведется с наименьшими затратами времени.
* **Следующее подходящее** — работает примерно так же, как и предыдущий алгоритм, за исключением того, что запоминает свое местоположение при выделении. При следующем запросе на выделение памяти поиск начинается с того места, на котором алгоритм остановился в предыдущий раз. Исследование работы алгоритма показало, что его производительность несколько хуже, чем у **«первого подходящего»**
* **Наиболее подходящее** — при нем ведется линейный поиск наименьшего по размеру подходящего сегмента. Это делается для того, чтобы наилучшим образом соответствовать запросу и имеющимся пустым пространствам в списке
* **Наименее подходящее** — алгоритм, противоположный вышеописанному — при выделении используется наибольший возможный сегмент памяти. Моделирование показало, что использование данного алгоритма не является хорошей идеей.

Далее речь будет идти об алгоритме «первое подходящее». Этот алгоритм работает быстрее, чем «наиболее подходящее» (за счет того, что при каждом запросе не ведется поиск по всему списку). Парадоксально, но его применение даже приводит к менее расточительному использованию памяти, чем использование «наиболее подходящего» или «следующего подходящего».

Работа всех вышеописанных алгоритмов может быть ускорена за счет ведения отдельных списков для занятых и для пустых пространств. Это ускоряет выделение памяти, но замедляет процедуру освобождения памяти. Так или иначе, даже при всех улучшениях данные алгоритмы достаточно сильно страдают от фрагментации.

## Алгоритм двойников

В данном алгоритме свободная часть памяти разбивается до тех пор, пока не выйдет блок памяти нужного размера, в каждом блоке есть идентификатор, обозначающий занят или свободен блок. Если освобождается блок и его двойник оказывается свободен, то двойников сливают. Полученный блок пытаются слить с его двойником. Блок, который не удалось слить добавляют в список свободных блоков. Свободные блоки хранятся в двусвязном списке

# Реализации аллокаторов

# BlocksAllocator.h

#include <iostream>

#include <algorithm>

#include <list>

#include <string>

using namespace std;

struct MemoryNode {

    char\* begin;

    size\_t capacity;

    string type;

};

ostream& operator << (ostream& os, const MemoryNode& node);

class BlocksAllocator {

public:

    explicit BlocksAllocator(size\_t size\_of\_data);

    void\* alloc(size\_t memory\_size);

    void dealloc(void\* block);

    void print\_memory(ostream& os) const;

    ~BlocksAllocator();

private:

    list<MemoryNode> mem\_list;

    char\* data;

};

# BlocksAllocator.cpp

#include "BlocksAllocator.h"

using namespace std;

BlocksAllocator::BlocksAllocator(size\_t size\_of\_data) {

    data = (char \*)malloc(size\_of\_data);

    mem\_list.push\_front({data, size\_of\_data, "Freely"});

}

void \*BlocksAllocator::alloc(size\_t memory\_size) {

    if (memory\_size == 0) {

        return nullptr;

    }

    size\_t size\_of\_node = 0;

    auto needed\_node = mem\_list.end();

    for (auto it = mem\_list.begin(); it != mem\_list.end(); ++it) {

        if (it->type == "Freely" && it->capacity >= memory\_size) {

            size\_of\_node = it->capacity;

            needed\_node = it;

        }

    }

    if (size\_of\_node == 0) {

        throw std::bad\_alloc();

    }

    if (memory\_size == size\_of\_node) {

        needed\_node->type = "Occupied";

    } else {

        MemoryNode new\_node{needed\_node->begin + memory\_size, needed\_node->capacity - memory\_size, "Freely"};

        needed\_node->capacity = memory\_size;

        needed\_node->type = "Occupied";

        mem\_list.insert(next(needed\_node), new\_node);

    }

    return (void \*)(needed\_node->begin);

}

void BlocksAllocator::dealloc(void \*block) {

    auto it = find\_if(mem\_list.begin(), mem\_list.end(), [block](const MemoryNode &node) {

        return node.begin == (char \*) block && node.type == "Occupied";

    });

    if (it == mem\_list.end()) {

        cout << "This pointer wasnt allocated by this allocator";

    }

    it->type = "Freely";

    if (it != mem\_list.begin() && prev(it)->type == "Freely") {

        auto prev\_it = prev(it);

        prev\_it->capacity += it->capacity;

        mem\_list.erase(it);

        it = prev\_it;

    }

    if (next(it) != mem\_list.end() && next(it)->type == "Freely") {

        auto next\_it = next(it);

        it->capacity += next\_it->capacity;

        mem\_list.erase(next\_it);

    }

}

void BlocksAllocator::print\_memory(ostream& os) const {

    int occ\_sum = 0, free\_sum = 0;

    for (auto& elem : mem\_list) {

        os << elem << endl;

        if (elem.type == "Freely") {

            free\_sum += elem.capacity;

        } else {

            occ\_sum += elem.capacity;

        }

    }

    os << "Occupied memory " << occ\_sum << endl;

    os << "Free memory " << free\_sum << endl;

}

std::ostream& operator << (std::ostream& os, const MemoryNode& node) {

    if ("Freely" == node.type) {

        return os << "Node: capacity " << node.capacity << ", type " << "Freely";

    } else {

        return os << "Node: capacity " << node.capacity << ", type " << "Occupied";

    }

}

BlocksAllocator::~BlocksAllocator() {

    free(data);

}

# BuddyAllocator.h

#include <exception>

#include <map>

#include <iostream>

using namespace std;

class BuddyAllocator {

public:

    using MapType = multimap<size\_t, char\*>;

    BuddyAllocator(size\_t minBlockSizeArg, size\_t maxSizeArg);

    void\* alloc(const size\_t size);

    void dealloc(void\* ptr, size\_t size);

    void print\_memory(ostream& os) const;

    ~BuddyAllocator();

private:

    size\_t Rounder(const size\_t size);

    size\_t minBlockSize, maxSize;

    MapType allocMap;

    char \*allocatedBlocks;

};

# BuddyAllocator.cpp

#include "BuddyAllocator.h"

using MapType = multimap<size\_t, char\*>;

BuddyAllocator::BuddyAllocator(size\_t minBlockSizeArg, size\_t maxSizeArg):minBlockSize(minBlockSizeArg),maxSize(maxSizeArg) {

    allocatedBlocks = static\_cast<char\*>(::operator new(maxSize));

    allocMap.insert({maxSize, allocatedBlocks});

}

void \*BuddyAllocator::alloc(const size\_t size) {

    if (size == 0) {

        return nullptr;

    }

    size\_t roundUp = Rounder(size);

    for (MapType::iterator it = allocMap.begin(); it != allocMap.end(); ++it) {

        if (it->first == size) { *// Found the block of appropriate size*

            void\* result {

                it->second

            };

            allocMap.erase(it);

            return result;

        } else if (it->first > size) { *// Map stores block sizes in ascending order*

            int allocSize = it->first;

            char \*blockStart = it->second;

            while (allocSize > size) {

                allocSize /= 2;

                if (allocSize < size) {

                    break;

                }

                allocMap.insert(pair<size\_t, char\*> (allocSize, blockStart + allocSize));

            }

            allocMap.erase(it);

            return blockStart;

        }

    }

    throw std::bad\_alloc();

}

void BuddyAllocator::dealloc(void\* ptr, size\_t size) {

    size\_t bufferedSize = size;

    while(true) {

        pair<MapType::iterator, MapType::iterator> range (allocMap.equal\_range(bufferedSize));

        if (range.first != allocMap.end() && range.first->first == bufferedSize) { *// Free blocks of same size found*

            auto it = range.first;

            short index = ((char\*)ptr - allocatedBlocks) / bufferedSize % 2;

*// If index is 1 then element is odd, else even*

            while (it != range.second) {

*// If difference between the two pointers equals to size,*

*// then we can merge them into one, since they are both free*

                if (index == 1) {

                    if (((char\*)ptr - it->second) == bufferedSize) {

                        bufferedSize \*= 2;

                        ptr = it->second;

                        allocMap.erase(it);

                        break; *// Element found, break of while loop*

                    }

                } else {

                    if ((it->second - (char\*)ptr) == bufferedSize) {

                        bufferedSize \*= 2;

                        allocMap.erase(it);

                        break; *// Element found, break of while loop*

                    }

                }

                ++it;

            }

            if (it == range.second) { *// No elements found in this size range*

                allocMap.insert(std::pair<size\_t, char\*> (bufferedSize, (char\*)ptr));

                break;

            } else {

                continue;

            }

        } *// No elements of this size found; break out*

        else {

            allocMap.insert(std::pair<size\_t, char\*> (bufferedSize, (char\*)ptr));

            break;

        }

    }

    return;

}

void BuddyAllocator::print\_memory(ostream& os) const {

    int occ\_mem = 0;

    for (auto& p: allocMap) {

        occ\_mem += p.first;

    }

    int free\_mem = maxSize - occ\_mem;

    os << "Occupied memory " << occ\_mem << "\n" << "Free memory " << free\_mem  << "\n\n";

}

BuddyAllocator::~BuddyAllocator() {

    ::operator delete(allocatedBlocks);

    allocMap.clear();

}

size\_t BuddyAllocator::Rounder(const size\_t size) {

        if (size <= minBlockSize) return minBlockSize;

        size\_t roundUp = 1;

        while (roundUp < size) {

            roundUp <<= 1;

        }

        return roundUp;

}

# main.cpp

#include <iostream>

#include "BuddyAllocator.h"

#include <chrono>

#include "BlocksAllocator.h"

#include <vector>

using namespace std;

int main() {

    using namespace std::chrono;

    {

        steady\_clock::time\_point list\_allocator\_init\_start = steady\_clock::now();

        BlocksAllocator list\_allocator(4096);

        steady\_clock::time\_point list\_allocator\_init\_end = steady\_clock::now();

        cerr << "List allocator initialization with one page of memory :"

                  << chrono::duration\_cast<std::chrono::nanoseconds>(

                          list\_allocator\_init\_end - list\_allocator\_init\_start).count()

                  << " ns" << endl;

        steady\_clock::time\_point buddy\_allocator\_init\_start = steady\_clock::now();

        BuddyAllocator bAlloc(32, 4096);

        steady\_clock::time\_point buddy\_allocator\_init\_end = steady\_clock::now();

        cerr << "Buddy allocator initialization with one page of memory :"

                  << chrono::duration\_cast<std::chrono::nanoseconds>(

                  buddy\_allocator\_init\_end - buddy\_allocator\_init\_start).count()

                  << " ns" << endl;

        cerr << endl;

    }

    cerr << "First test: Allocate 10 char[256] arrays, free 5 of them, allocate 10 char[128] arrays:" << endl;

    {

        BlocksAllocator allocator(4096);

        vector<char \*> pointers(15, nullptr);

        steady\_clock::time\_point test1\_start = steady\_clock::now();

        for (int i = 0; i < 10; ++i) {

            pointers[i] = (char \*) allocator.alloc(256);

        }

        for (int i = 5; i < 10; ++i) {

            allocator.dealloc(pointers[i]);

        }

        for (int i = 5; i < 15; ++i) {

            pointers[i] = (char \*) allocator.alloc(128);

        }

        steady\_clock::time\_point test1\_end = steady\_clock::now();

        cerr << "List allocator first test: "

             << std::chrono::duration\_cast<chrono::microseconds>(test1\_end - test1\_start).count()

             << " microseconds" << endl;

        allocator.print\_memory(cerr);

        for (int i = 0; i < 15; ++i) {

            allocator.dealloc(pointers[i]);

        }

    }

    {

        BuddyAllocator bAlloc(16, 4096);

        vector<char \*> pointer(15, nullptr);

        steady\_clock::time\_point buddy\_test1\_start = steady\_clock::now();

        for (int i = 0; i < 10; ++i) {

            pointer[i] = (char \*) bAlloc.alloc(256);

        }

        for (int i = 5; i < 10; ++i) {

            bAlloc.dealloc(pointer[i], 256);

        }

        for (int i = 5; i < 15; ++i) {

            pointer[i] = (char \*) bAlloc.alloc(128);

        }

        steady\_clock::time\_point buddy\_test1\_end = steady\_clock::now();

        cerr << "Buddy allocator first test: "

                 << chrono::duration\_cast<std::chrono::microseconds>(buddy\_test1\_end - buddy\_test1\_start).count()

                 << " microseconds" << std::endl;

        for (int i = 0; i < 5; ++i) {

            bAlloc.dealloc(pointer[i], 256);

        }

        bAlloc.print\_memory(cerr);

        for (int i = 5; i<15; ++i) {

            bAlloc.dealloc(pointer[i], 128);

        }

    }

    cerr << "Second test: Allocate and free 750 20 bytes arrays:\n";

    {

        BlocksAllocator allocator(16000);

        std::vector<char \*> pointers(900, nullptr);

        int arr\_size = 10;

        steady\_clock::time\_point alloc\_start = steady\_clock::now();

        for (int i = 0; i < 900; ++i) {

            pointers[i] = (char \*) allocator.alloc(arr\_size);

        }

        steady\_clock::time\_point alloc\_end = steady\_clock::now();

        for (int i = 0; i < 900; ++i) {

            allocator.dealloc(pointers[i]);

        }

        cerr << "List allocator second test:" << endl

                  << "Allocation: " << duration\_cast<chrono::microseconds>(alloc\_end - alloc\_start).count()

                  << " microseconds" << endl << endl;

    }

    {

        BuddyAllocator allocator(2, 16000);

        vector<char \*> pointers(900, nullptr);

        steady\_clock::time\_point alloc\_start = steady\_clock::now();

        int arr\_size = 10;

        for (int i = 0; i < 900; ++i) {

            pointers[i] = (char \*) allocator.alloc(arr\_size);

        }

        steady\_clock::time\_point alloc\_end = steady\_clock::now();

        for (int i = 0; i < 900; ++i) {

            allocator.dealloc(pointers[i], arr\_size);

        }

        steady\_clock::time\_point test\_end = steady\_clock::now();

        cerr << "Buddy allocator second test:" << endl

                  << "Allocation :" << duration\_cast<chrono::microseconds>(alloc\_end - alloc\_start).count()

                  << " microseconds" << endl;

    }

}

# Тестирование

**Будем тестировать следующие характеристики:**

* Скорость выделения и освобождения блоков
* Фрагментацию
* Экономичность

**Демонстрация работы программы**

evgeny@LAPTOP-F153AKTP:~/OS/kp$ g++ BuddyAllocator.cpp BlocksAllocator.cpp main.cpp

evgeny@LAPTOP-F153AKTP:~/OS/kp$ ./a.out

List allocator initialization with one page of memory :7404 ns

Buddy allocator initialization with one page of memory :6632 ns

First test: Allocate 10 char[256] arrays, free 5 of them, allocate 10 char[128] arrays:

List allocator first test: 24 microseconds

Node: capacity 256, type Occupied

Node: capacity 256, type Occupied

Node: capacity 256, type Occupied

Node: capacity 256, type Occupied

Node: capacity 256, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 1536, type Freely

Occupied memory 2560

Free memory 1536

Buddy allocator first test: 26 microseconds

Occupied memory 2816

Free memory 1280

Second test: Allocate and free 750 20 bytes arrays:

List allocator second test:

Allocation: 18143 microseconds

Buddy allocator second test:

Allocation :554 microseconds

## Результаты тестов

Как видно из вывода, программа была запущена на 3 тестах

* Проверка времени, требуемого для инициализации —алгоритму на списках требуется больше времени для инициализации заголовков блоков, однако разница с алгоритмом двойников несущественная.
* **Аллокация 256 байт 10 раз, освобождение 5 из полученных указателей, аллокация 128 байт 10 раз**. Оба алгоритма хорошо справились с этим тестом, однако алгоритм списка блоков был чуть эффективнее как по времени, так и по занимаемой памяти.
* **Аллокация и удаление 750 раз по 20 байт.** Данный тест призван сравнить быстродействие аллокаторов. Как видно, аллокатор, основанный на алгоритме двойников справился значительно быстрее. Это связано с тем, что аллокатору «первое подходящее» приходится при каждой аллокации итерироваться по всему списку в поисках наиболее подходящего сегмента памяти, в то время как у другого аллокатора поиск сегмента происходит практически за константное время.

Очевидно, что алгоритм, основанный на двойниках гораздо более устойчив к увеличению числа запросов, что в очередной раз подчеркивает его преимущество над списковым аллокатором.

# Заключение

Из сравнения стало ясно, что аллокатор «первое подхоядщее» имеет гораздо больше недостатков, чем достоинств, и в целом проигрывает «спискам, основанным на двойниках» почти по всем критериям — он работает относительно медленно, и неустойчив к увеличению числа запросов(время работы при увеличении числа входных данных в 2 раза увеличилось в 4). Его оппонент достаточно быстр и прост в реализации, не имеет проблемы фрагментации, но не поддерживает слияние и приводит к не очень экономному расходу памяти.

При выполнении данной курсовой работы я познакомился с несколькими видами аллокаторов, а так же более подробно исследовал два из них. Благодаря этой работе я чуть лучше изучил принципы и особенности работы UNIX систем и узнал много нового.