Московский авиационный институт (национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование» Дисциплина «Операционные системы»

Курсовой проект

Тема: Сравнение алгоритмов аллокации: Алгоритм двойников, Алгоритм отложенного слияния.

Студент: Будникова В.П.

Группа: М8О-207Б-19

Преподаватель: Миронов Е. С.

Дата:

Оценка:

Москва, 2020

Содержание

Алгоритм двойников

Данный алгоритм является схемой выделения памяти, сочетающей в себе возможность слияния буферов и распределитель по степени числа два. Блоки могут быть в двух состояния: занятый, свободный. Также в блоках хранится информация о его размере и адрес.

При поступлении запроса на выделения памяти, размер запрашиваемой памяти обычно округляется до ближайшей степени двойки, далее, если блок пустой и большего размера, чем запрашиваемая память, то он делится на два до тех пор, пока размер получившегося блока не станет равным запрашиваемой памяти, далее блок помечается занятым и возвращается адрес. Теперь наш блок является последовательностью более маленьких блоков. Если поступил запрос на выделения памяти, ищется либо первый свободный блок, размер которого равен размеру запрашиваемой памяти, либо больший свободный блок(в нем происходит аналогичная ситуация деления).

При поступлении запроса на освобождение памяти, находится нужный блок и помечается свободным, далее, если его возможно слить с соседним блоком(их размеры должны быть равны), происходит слияние и повторная проверка на возможность слиться с соседним блоком. Слияние происходит до тех пор, пока это возможно.

Плюсы данного алгоритма:

1. Гибкость - возможность повторного использования памяти
2. Возможность слияния

Минусы:

1. Производительность - при частом выделении и освобождении время будет уходить на обьединение и слияние блоков.
2. Отсутствует возможность частичного освобождения
3. Для запросов, требующих выделения памяти, не кратной степени 2, выделяется большая память.

Реализация:

Так как в задании сказано, что при инициализации алгоритма нужно передать участок памяти, с которым будет работать данный алгоритм, то в начале программы с помощью malloc выделяется память и передается в конструктор.

Я решила в своей реализации для хранения страниц использовать контейнер вектор. Также существуют две структуры: block - блок, хранящий адрес и состояние - занят или свободен, page - страница, содержащая вектор блоков. Также класс хранит указатель на общую память(buff). Для работы с адресной арифметикой вся указатель преобразуется из указателя на void, в указатель на char.

Метод класса allocate - выделяет память, запрошенную в аргументе. Происходит округление поступившего размера до ближайшей степени двойки, и далее работает по алгоритму - разделяет блоки или ищет свободные, возвращает указатель типа void. Если памяти в странице не хватает для выделения, то производится добавление новой страницы, если в буфере достаточно памяти. Если же память не удается выделить, то бросается исключение.

Метод класса destroy - освобождает память, и производит слияние до тех пор, пока это возможно.

Метод Print_condition() - блоков в каждой странице.

Листинг

```
#pragma once
#include <iostream>
#include <vector>
#include <math.h>

namespace BuddyMemory {
struct Block {
    char * adr;
    int size;
    bool free;
};

struct Page {
    std::vector<Block> block;
};


class Memory {
    private:
        std::vector<Page> memory;
        void * buff;
        int memory_size;
        int page_size;
    public:

    Memory(void * new_memory, int size, int pageSize):
memory_size(size) {
        int mem = 1;
        while (pow(2, mem) < pageSize) {
            ++mem;
        }
        page_size = pow(2, mem);
        Block block1{(char *)new_memory, page_size, true};
        Page page {std::vector<Block>{block1}};
        memory.push_back(page);
```

```
    }

    ~Memory() {}
    char * allocate(const int &t) {
        int mem = 1;
        while (pow(2, mem) < t) {
            ++mem;
        }
        int mem_size = pow(2, mem);
        for (int i = 0; i < memory.size(); ++i) {
            int k = 0;
            while ((memory[i].block[k].size != mem_size || !
memory[i].block[k].free) && k < memory[i].block.size()) {
                if (memory[i].block[k].size > mem_size &&
memory[i].block[k].free) {
                    memory[i].block[k].size /= 2;
                    Block block_new{(memory[i].block[k].adr +
memory[i].block[k].size), memory[i].block[k].size, true};
                    memory[i].block.insert(memory[i].block.begin()
+ k, block_new);
                    --k;
                }
                ++k;
                if (k == memory[i].block.size()) {
                    break;
                }
            }
            if (k < memory[i].block.size()) {
                if (memory[i].block[k].free) {
                    memory[i].block[k].free = false;
                    return memory[i].block[k].adr;
                }
            } else if (i == memory.size() - 1) {
                if (memory.size() < memory_size / page_size) {
                    Block block1{memory[i].block[0].adr +
page_size, page_size, true};
                    Page page {std::vector<Block>{block1}};
                    memory.push_back(page);
                } else {
                    // std::cout << "Не удалось выделить память"
<< std::endl;
                    std::bad_alloc err;
                    throw err;
                    return NULL;
                }
            }
        }
        return NULL;
    }
    void Print_condition() {
        for (int u = 0; u < memory.size(); ++u) {
            std::cout << "page: " << u << std::endl;
```

```
            for (int l = 0; l < memory[u].block.size(); ++l) {
                std::cout << "block" << l << ": " <<
memory[u].block[l].size << " b   "<< memory[u].block[l].free <<
std::endl;
            }
            std::cout << std::endl;
        }
    }
    void destroy(void * t) {
        int a = 0;
        bool fl = false;
        for (int i = 0; i < memory.size(); ++i) {
            for (int j = 0; j < memory[i].block.size(); ++j) {
                if (!memory[i].block[j].free &&
memory[i].block[j].adr == (char *)t) {
                    fl = true;
                    memory[i].block[j].free = true;
                    break;
                }
            }
            if (fl) {
                break;
            }
        }
        for (int i = 0; i < memory.size(); ++i) {
            int j = 0;
            while (j < memory[i].block.size()) {
                if (memory[i].block[j].size == page_size) {
                    break;
                }
                if (j == 0 && memory[i].block[j].free &&
memory[i].block[j + 1].free && memory[i].block[j].size ==
memory[i].block[j + 1].size) {
                    memory[i].block[j].size *= 2;
                    memory[i].block[j].free = true;
                    memory[i].block.erase(memory[i].block.begin()
+ 1);
                    j = 0;
                } else if (j == memory[i].block.size() - 1 &&
memory[i].block[j].free && memory[i].block[j - 1].free &&
memory[i].block[j].size == memory[i].block[j - 1].size) {
                    memory[i].block[j - 1].size *= 2;
                    memory[i].block.pop_back();
                    j = j - 1;
                } else if (j != memory[i].block.size() - 1 &&
memory[i].block[j].free && memory[i].block[j + 1].free &&
memory[i].block[j].size == memory[i].block[j + 1].size) {
                    memory[i].block[j].size *= 2;
                    memory[i].block[j].free = true;
                    memory[i].block.erase(memory[i].block.begin()
+ j + 1);
```

```
                } else  if (j != 0 && memory[i].block[j].free &&
memory[i].block[j - 1].free && memory[i].block[j].size ==
memory[i].block[j - 1].size) {
                    memory[i].block[j - 1].size *= 2;
                    memory[i].block.erase(memory[i].block.begin()
+ j);
                    j = j - 1;
                } else {
                    ++j;
                }
            }
        }
    }

};
}
```

Пример использования:

make rLera:kp1 valeriabudnikova$ make run
./kp
Размер страницы - 32
Выделилось 4 байт
page: 0
block0: 4 b 0
block1: 4 b 1
block2: 8 b 1
block3: 16 b 1

Выделилось 8 байт
page: 0
block0: 4 b 0
block1: 4 b 1
block2: 8 b 0
block3: 16 b 1

Выделилось 12 байт
page: 0
block0: 4 b 0
block1: 4 b 1
block2: 8 b 0
block3: 16 b 0

Выделилось 16 байт
page: 0
block0: 4 b 0
block1: 4 b 1
block2: 8 b 0
block3: 16 b 0

page: 1
block0: 16 b 0
block1: 16 b 1

Удалился 1 блок
page: 0
block0: 8 b 1
block1: 8 b 0
block2: 16 b 0

page: 1
block0: 16 b 0
block1: 16 b 1

Удалился 0 блок
page: 0
block0: 16 b 1
block1: 16 b 0

page: 1
block0: 16 b 0
block1: 16 b 1

Удалился 1 блок
page: 0
block0: 32 b 1

page: 1
block0: 16 b 0
block1: 16 b 1

Удалился 0 блок
page: 0
block0: 32 b 1

page: 1
block0: 32 b 1

Алгоритм отложенного слияния

Алгоритм отложенного слияния случит улучшение производительности для алгоритма двойников. Если в алгоритме двойников много раз чередовать выделение и освобождение, то производительность сильно падает, так как затрачивается много времени на разделение и слияние блоков. Алгоритм отложенного слияния предлагает хранить дополнительные данные для страницы - количество свободных глобально блоков и количество локально свободных блоков. Глобально свободные блоки - блоки, доступные для объединения. Локально свободные блоки - свободные блоки, которые нельзя обьединить.

Вводится дополнительная проверка перед слиянием. Для этого необходимо понять, в каком состоянии находится система, состояние определяет параметром, называемым допуском, который определяется по формуле - slack = 2L-G. Где L - количество локально свободных блоков, G - количество глобально свободных блоков. Если допуск ниже 2, то система нуждается в слиянии. Такая проверка улучшает производительность, так как в равновесном состоянии системы не будет тратиться время на не необходимое слияние, разделение блоков как и в предыдущем алгоритме производится по-необходимости.

Реализация:

Реадизация данного алгоритма схожя с реализаций алгоритма двойников. Выделение памяти происходит аналогичный образом - происходит округление поступившего размера до ближайшей степени двойки и выделяется соответствующая память, при этом, если после разделения блока и выделения одной части, состояние страницы меняется - счетчик локальных страниц увеличивается, а глобальных - уменьшается. При выделении нужного блока, мы смотрим на его брата - если он свободен, значит блоки были глобально свободными, а теперь один из них станет занят, следовательно нужно изменить счетчики. Также при разделении блоков счетчик глобально свободных блоков увеличивается.

При освобождении памяти, так как все блоки в моей реализации хранятся в векторе, происходит обработка данного вектора(страницы), счетчики глобально свободных и локально свободных, в зависимости от ситуации меняются или нет.

Листинг

```
#pragma once
#include <iostream>
#include <vector>
#include <math.h>
```

```
namespace BuddyMemoryPro {
struct Block {
    char * adr;
    int size;
    bool free;
};

struct Page {
    std::vector<Block> block;
    int l;
    int g;
};


class Memory {
    private:
        std::vector<Page> memory;
        void * buff;
        int memory_size;
        int page_size;
    public:

    Memory(void * new_memory, int size, int pageSize):
memory_size(size) {
        int mem = 1;
        while (pow(2, mem) < pageSize) {
            ++mem;
        }
        page_size = pow(2, mem);
        Block block1{(char *)new_memory, page_size, true};
        Page page {std::vector<Block>{block1}, 1, 0};
        memory.push_back(page);
    }

    ~Memory() {}
    char * allocate(const int &t) {
        int mem = 1;
        while (pow(2, mem) < t) {
            ++mem;
        }
        int mem_size = pow(2, mem);
        for (int i = 0; i < memory.size(); ++i) {
            int k = 0;
            bool flag = false;
            while ((memory[i].block[k].size != mem_size || !
memory[i].block[k].free) && k < memory[i].block.size()) {
                if (memory[i].block[k].size > mem_size &&
memory[i].block[k].free) {
                    if (memory[i].block[k].size == page_size || !
flag) {
                        ++memory[i].g;
```

```
                        ++memory[i].g;
                        --memory[i].l;
                    } else {
                        ++memory[i].l;
                    }
                    flag = true;
                    memory[i].block[k].size /= 2;
                    Block block_new{(memory[i].block[k].adr +
memory[i].block[k].size), memory[i].block[k].size, true};
                    memory[i].block.insert(memory[i].block.begin()
+ k, block_new);
                    --k;
                }
                ++k;
                if (k == memory[i].block.size()) {
                    break;
                }
            }

            if (k < memory[i].block.size()) {
                if (memory[i].block[k].free) {

                    if (k == 0 && memory[i].block[k].free &&
memory[i].block[k + 1].free && memory[i].block[k].size ==
memory[i].block[k + 1].size) {
                        --memory[i].g;
                        --memory[i].g;
                        ++memory[i].l;
                    } else if (k == memory[i].block.size() - 1 &&
memory[i].block[k].free && memory[i].block[k - 1].free &&
memory[i].block[k].size == memory[i].block[k - 1].size) {
                        --memory[i].g;
                        --memory[i].g;
                        ++memory[i].l;
                    } else if (k != memory[i].block.size() - 1 &&
memory[i].block[k].free && memory[i].block[k + 1].free &&
memory[i].block[k].size == memory[i].block[k + 1].size) {
                        --memory[i].g;
                        --memory[i].g;
                        ++memory[i].l;
                    } else  if (k != 0 && memory[i].block[k].free
&& memory[i].block[k - 1].free && memory[i].block[k].size ==
memory[i].block[k - 1].size) {
                        --memory[i].g;
                        --memory[i].g;
                        ++memory[i].l;
                    } else {
                        --memory[i].l;
                    }
                    memory[i].block[k].free = false;
                    if (memory[i].block[k].size == page_size) {
                        memory[i].g = 0;
```

```
                    memory[i].l = 0;
                }
                return memory[i].block[k].adr;
            }
        } else if (i == memory.size() - 1) {
            if (memory.size() < memory_size / page_size) {
                Block block1{memory[i].block[0].adr +
page_size, page_size, true};
                Page page {std::vector<Block>{block1}, 1, 0};
                memory.push_back(page);
            } else {
                // std::cout << "Не удалось выделить память"
<< std::endl;
                std::bad_alloc err;
                throw err;
                return NULL;
            }
        }
        return NULL;
    }
    void Print_condition() {
        for (int u = 0; u < memory.size(); ++u) {
            std::cout << "page: " << u << std::endl;
            for (int l = 0; l < memory[u].block.size(); ++l) {
                std::cout << "block" << l << ": " <<
memory[u].block[l].size << " b   "<< memory[u].block[l].free <<
std::endl;
            }
            std::cout << std::endl;
        }
    }
    void destroy(void * t) {
        int a = 0;
        int page = 0;
        bool fl = false;
        for (int i = 0; i < memory.size(); ++i) {
            for (int j = 0; j < memory[i].block.size(); ++j) {
                if (!memory[i].block[j].free &&
memory[i].block[j].adr == (char *)t) {
                    fl = true;
                    page = i;
                    memory[i].block[j].free = true;
                    if  ((j == 0 && memory[i].block[j].size ==
memory[i].block[j + 1].size && !memory[i].block[j + 1].free) ||
                        (j == memory[i].block.size() - 1 &&
memory[i].block[j].size == memory[i].block[j - 1].size && !
memory[i].block[j - 1].free)) {
                        ++memory[i].l;
                        break;
                    } else
```

```
                    if ((j != 0 && memory[i].block[j].size ==
memory[i].block[j - 1].size && memory[i].block[j - 1].free) ||
                        (j != memory[i].block.size() - 1 &&
memory[i].block[j].size == memory[i].block[j + 1].size &&
memory[i].block[j + 1].free)) {

                        int count = 0;
                        int count2 = 0;
                        memory[i].l = 0;
                        memory[i].g = 0;
                        for (int l = 0; l < memory[i].block.size()
- 2; ++l) {
                            if (memory[i].block[l].size ==
memory[i].block[l + 1].size && memory[i].block[l].free &&
memory[i].block[l + 1].free) {
                                ++memory[i].g;
                                ++memory[i].g;
                                count += memory[i].block[l].size *
2;
                                count2 = memory[i].block[l].size *
2;
                                ++l;
                            } else if (memory[i].block[l].free &&
(memory[i].block[l].size == count || memory[i].block[l].size ==
count2)) {
                                count += memory[i].block[l].size;
                                count2 += memory[i].block[l].size;
                                ++memory[i].g;
                            } else if (memory[i].block[l].free) {
                                count = 0;
                                count2 = 0;
                                ++memory[i].l;
                            } else {
                                count2 = 0;
                                count = 0;
                            }
                        }
                        if (memory[i].block[memory[i].block.size()
- 2].free && (memory[i].block[memory[i].block.size() - 2].size ==
count || memory[i].block[memory[i].block.size() - 2].size ==
count2)) {
                            count +=
memory[i].block[memory[i].block.size() - 2].size;
                            count2 +=
memory[i].block[memory[i].block.size() - 2].size;
                            ++memory[i].g;
                            if
(memory[i].block[memory[i].block.size() - 1].free &&
(memory[i].block[memory[i].block.size() - 1].size == count ||
memory[i].block[memory[i].block.size() - 1].size == count2)) {
                                count +=
memory[i].block[memory[i].block.size() - 1].size;
```

```
                                ++memory[i].g;
                            } else {
                                ++memory[i].l;
                            }
                        } else if
(memory[i].block[memory[i].block.size() - 1].free &&
memory[i].block[memory[i].block.size() - 1].free &&
memory[i].block[memory[i].block.size() - 1].size !=
memory[i].block[memory[i].block.size() - 2].size) {
                            ++memory[i].l;
                            ++memory[i].l;
                        }
                    } else {
                        ++memory[i].l;
                    }
                    break;
                }
                if (fl) break;
            }
            if (fl) break;
        }
        if (fl) {
            if ((memory[page].block.size() - 2 * memory[page].l -
memory[page].g) < 2 ) {
                int j = 0;
                while (j < memory[page].block.size()) {
                    if (memory[page].block[j].size == page_size) {
                        break;
                    }
                    if (j == 0 && memory[page].block[j].free &&
memory[page].block[j + 1].free && memory[page].block[j].size ==
memory[page].block[j + 1].size) {
                        memory[page].block[j].size *= 2;
                        memory[page].block[j].free = true;

memory[page].block.erase(memory[page].block.begin() + 1);
                        j = 0;
                    } else if (j == memory[page].block.size() - 1
&& memory[page].block[j].free && memory[page].block[j - 1].free &&
memory[page].block[j].size == memory[page].block[j - 1].size) {
                        memory[page].block[j - 1].size *= 2;
                        memory[page].block.pop_back();
                        j = j - 1;
                    } else if (j != memory[page].block.size() - 1
&& memory[page].block[j].free && memory[page].block[j + 1].free &&
memory[page].block[j].size == memory[page].block[j + 1].size) {
                        memory[page].block[j].size *= 2;
                        memory[page].block[j].free = true;

memory[page].block.erase(memory[page].block.begin() + j + 1);
```

```
                    } else  if (j != 0 &&
memory[page].block[j].free && memory[page].block[j - 1].free &&
memory[page].block[j].size == memory[page].block[j - 1].size) {
                        memory[page].block[j - 1].size *= 2;

memory[page].block.erase(memory[page].block.begin() + j);
                        j = j - 1;
                    } else {
                        ++j;
                    }
                }
            }
        }
    }

};
}
```

Пример использования:

make rLera:kp1 valeriabudnikova$ make run
./kp
Размер страницы - 32
Выделилось 4 байт
page: 0
block0: 4 b 0
block1: 4 b 1
block2: 8 b 1
block3: 16 b 1

Выделилось 8 байт
page: 0
block0: 4 b 0
block1: 4 b 1
block2: 8 b 0
block3: 16 b 1

Выделилось 12 байт
page: 0
block0: 4 b 0
block1: 4 b 1
block2: 8 b 0
block3: 16 b 0

Выделилось 16 байт
page: 0
block0: 4 b 0
block1: 4 b 1
block2: 8 b 0
block3: 16 b 0

page: 1
block0: 16 b 0
block1: 16 b 1

Удалился 1 блок
page: 0
block0: 4 b 1
block1: 4 b 1
block2: 8 b 0
block3: 16 b 0

page: 1
block0: 16 b 0
block1: 16 b 1

Удалился 0 блок
page: 0
block0: 16 b 1
block1: 16 b 0

page: 1
block0: 16 b 0
block1: 16 b 1

Удалился 1 блок
page: 0
block0: 16 b 1
block1: 16 b 1

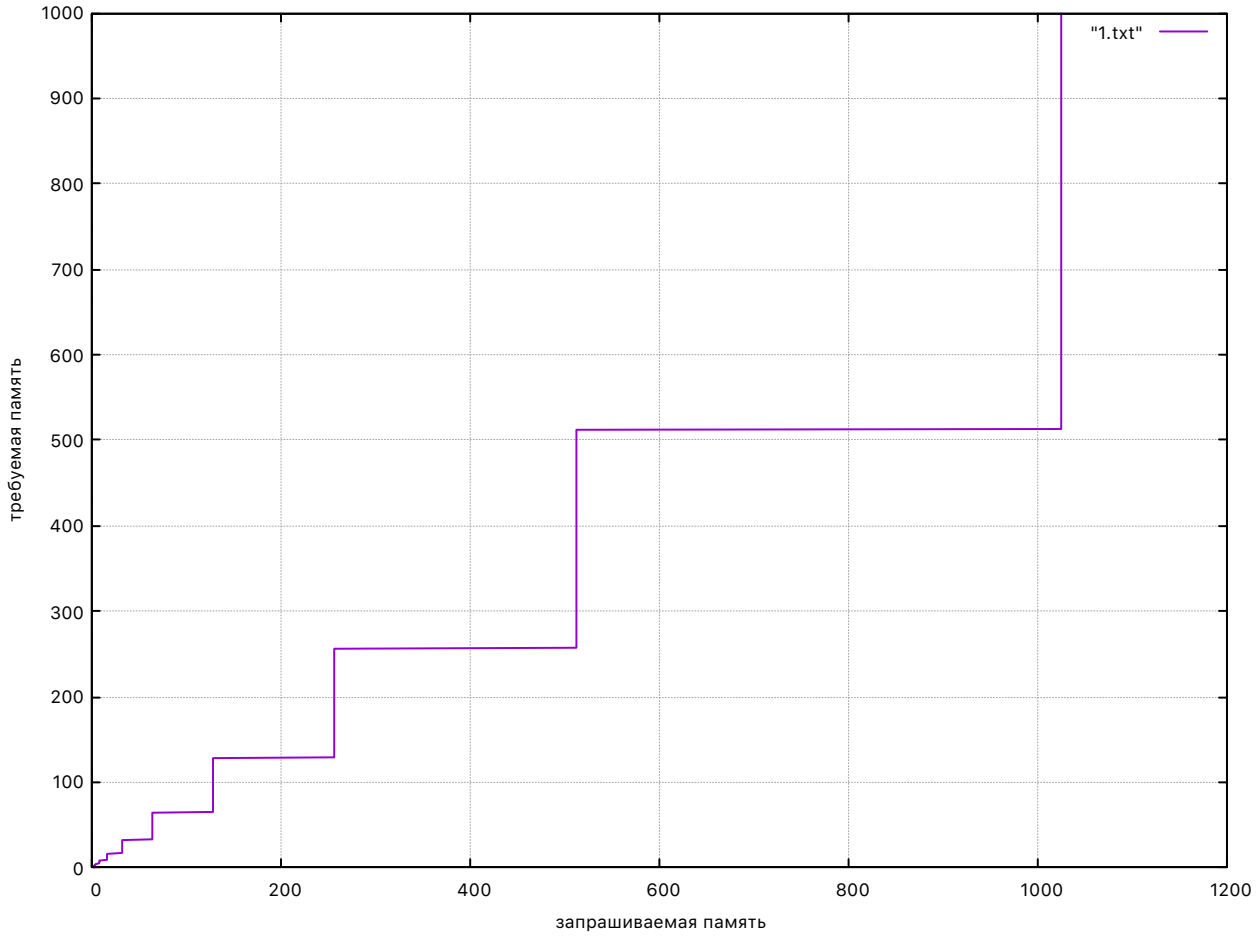
page: 1
block0: 16 b 0
block1: 16 b 1

Удалился 0 блок
page: 0
block0: 16 b 1
block1: 16 b 1

page: 1
block0: 16 b 1
block1: 16 b 1

Сравнение фактора использования

Так как в каждом из алгоритмов при запросе на выделение - происходит округление до степени двойки, то фактор использования у данных алгоритмах будет одинаковый. Фактор использования определяется запрашиваемой памятью и требуемой памятью. Определяется путем деления первого параметра на второй. Для анализа фактора использования приведу таблицу зависимости этих параметров:



Так как при выделении памяти происходит округление до степени двойки, то при выделении памяти произвольного размера фактор использования будет меньше, чем при выделении памяти, кратной степени двойки.

При выделения памяти на такие стандартные типы, как short(2 bytes), int(4 bytes), long(4 bytes), long long(8 bytes), float(4 bytes), double(8 bytes) фактор использования будет равен единице, так как блоки будут использованы полностью, следовательно не будет оставаться "лишнего" свободного места, что обеспечивает эффективное использование памяти.

Если же будут поступать запросы на выделения памяти только для типов bool(1 byte) или char(1 byte), то эффективность использования памяти будет гораздо ниже, так как блоки будут заняты только на половину и, при занятости всей страницы, фактически она будет занята только на половину.

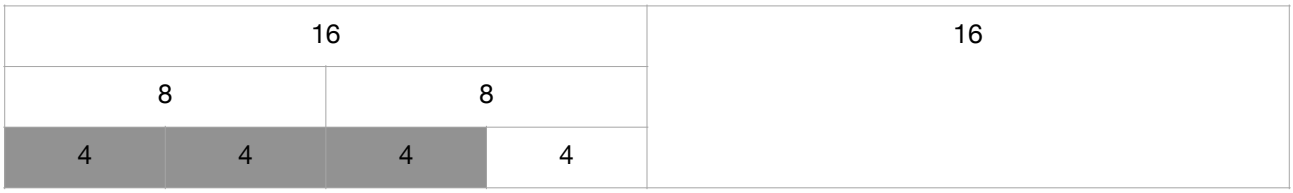
Сравнение фрагментации

Алгоритм двойников обладает меньшей фрагментацией, чем Алгоритм Слияния. Так как в первом алгоритме в любом случае производится попытка слияния, то возможность выделить память выделить память определенного размера увеличивается. Рассмотрим и сравним работу этих алгоритмов на примере:

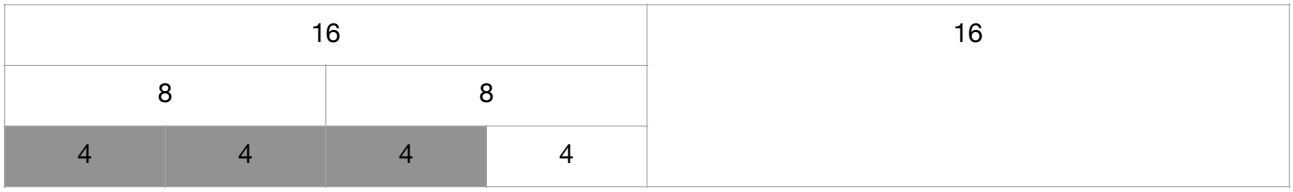
Для удобства пусть размер страницы будет 32 байт.

1. Сначала выделим 3 раза по 4 байта:

Алгоритм двойников:

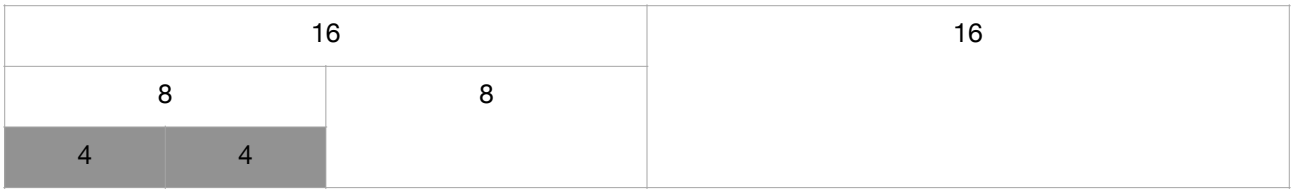


Отложенное слияние:

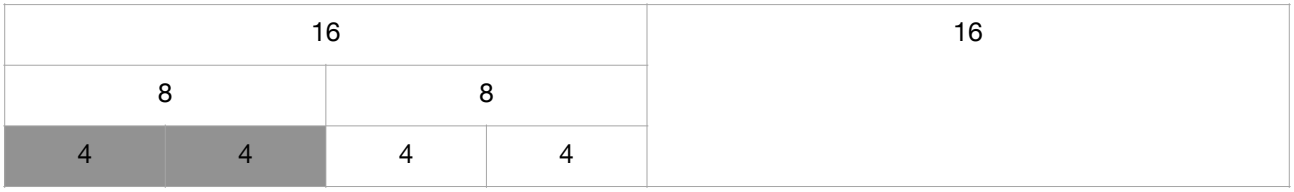


2. Освободим блок самый крайний блок

Алгоритм двойников:



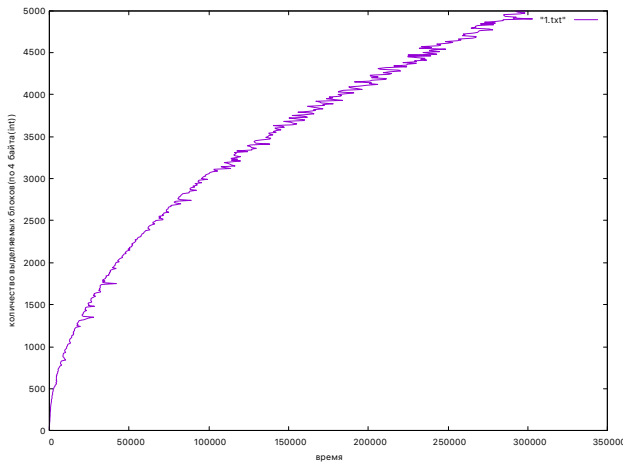
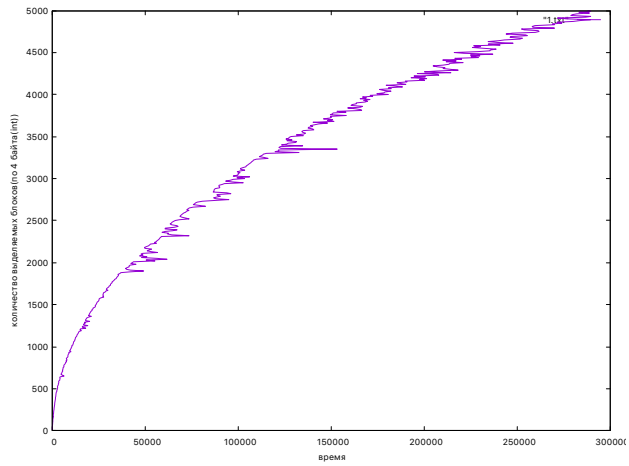
Отложенное слияние:



3. Теперь, если нам потребуется выделить 8 байт, то в первом случае мы выделим память после первых 8 байт, во втором случае, хоть у нас и свободны вторые 8 байт, но они разделены на блоки, следовательно мы не сможем туда записать, и нам придется делить следующий блок, а данные блоки останутся не использованы, что увеличивает фрагментацию. В случае, если система находится в равновесном состоянии, то с большей вероятностью фрагментация в первом алгоритме будет меньше, чем во втором, так как во втором будет больше не слитых блоков.

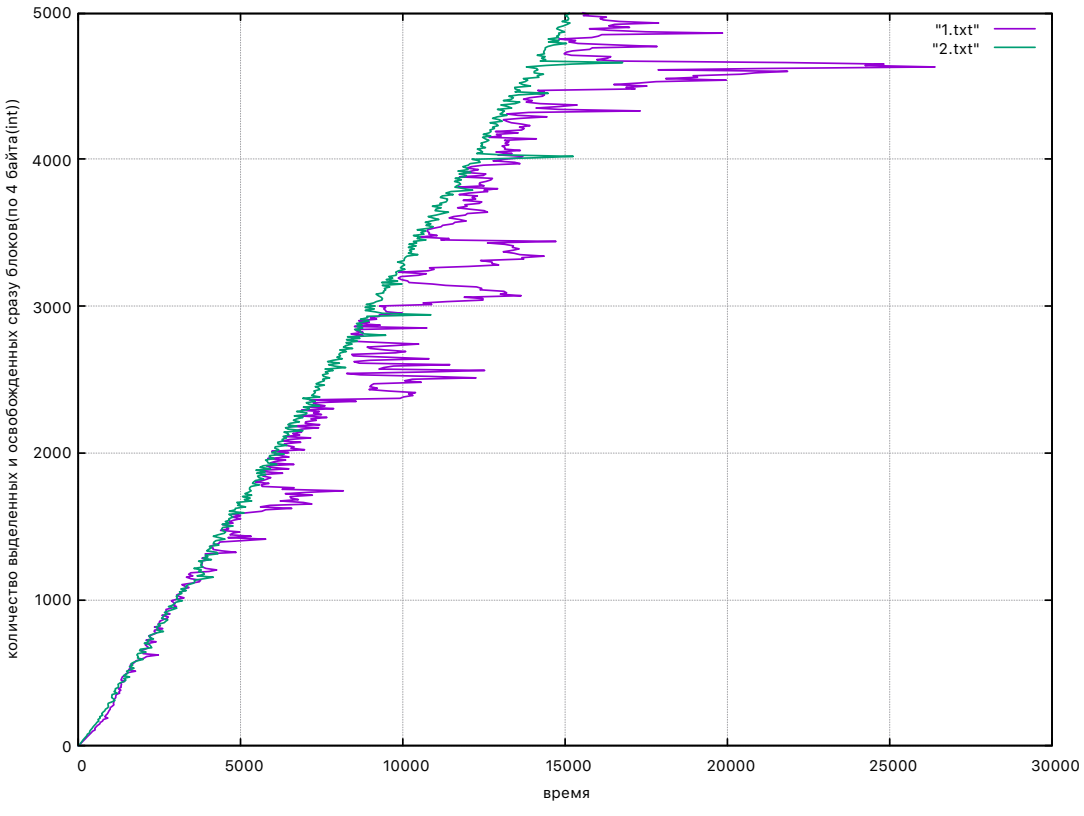
Сравнение выделения/освобождение памяти

Рассмотрим зависимость времени выделения блоков памяти разного размера. Размер страницы -1024 байт.

Алгоритм Двойников	Отложенное слияние
	

Как можно увидеть графики схожи, при сравнении графиков можно увидеть, что времени на выделения одинакового количества блоков в Оптимальном слиянии требуется немного больше, чем в алгоритме Двойников. Такое происходит так как при Отложенном слиянии, при вставке мы производим дополнительные вычисления для определения глобально свободных блоков и локально свободных блоков.

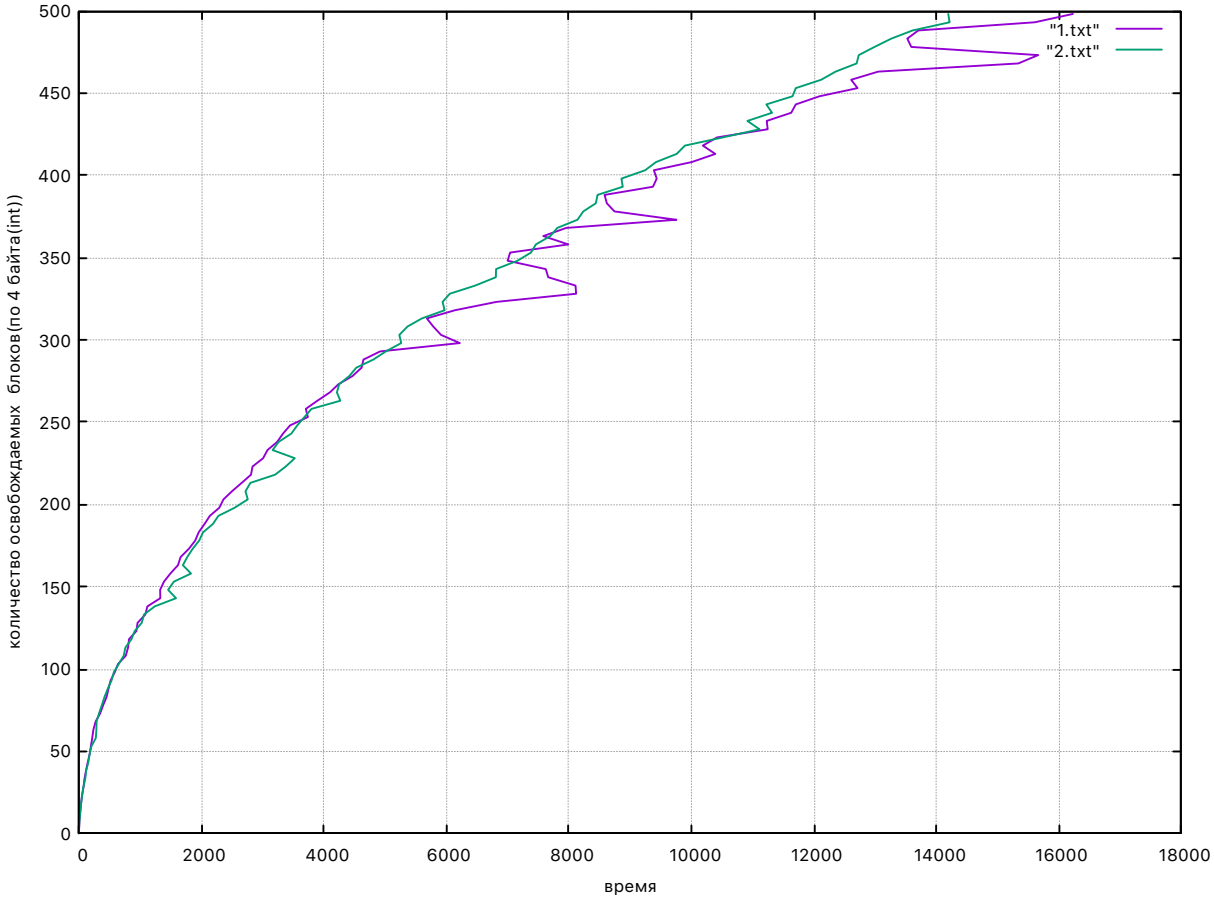
Теперь рассмотрим график который показывает как выделение и освобождение зависит от времени. Будем рассматривать освобождение сразу после выделения памяти. Размер страницы -1024 байт.



Далее во всем таблицах зеленым показан график Отложенного слияния, а фиолетовым - Алгоритма Двойников.

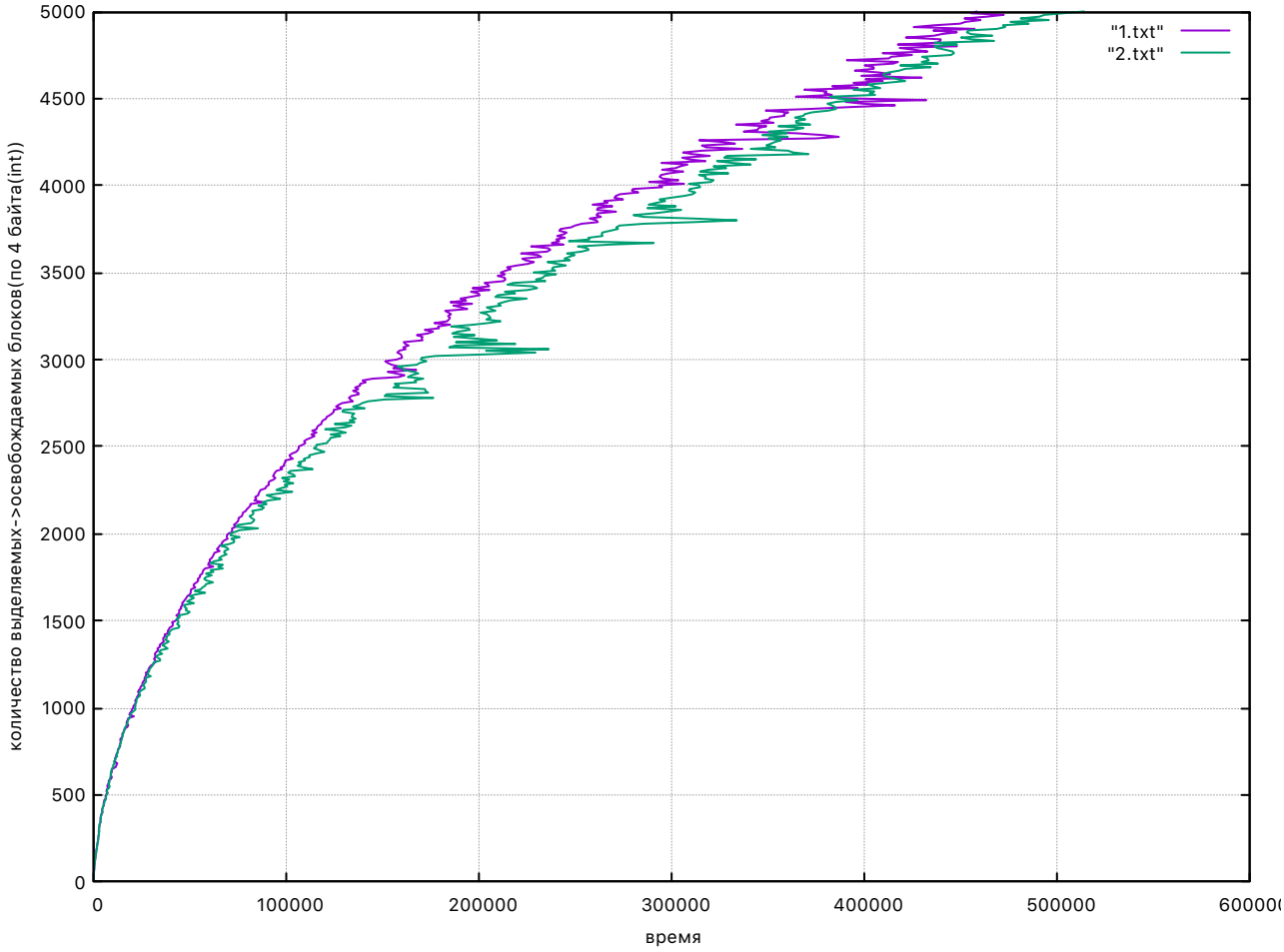
Данный график показывает, что времени на выделение и освобождение сразу элемента в Отложенном слиянии требуется меньше, чем в Алгоритме двойников. Это происходит потому, что при выделении и освобождении в Алгоритме двойников происходит очень много слияний и разделений блоков. Данный график может случить примером для оценки производительности при частом удалении и выделении памяти, так как мы наглядно видим, что Алгоритм Двойников делает "не необходимые" слияния, так которые затрачивается много времени.

Теперь сравним графики, которые показывают зависимость освобождения от времени, если мы будем удалять не сразу и будем удалять не все элементы, только их часть, после выделения. Размер страницы -1024 байт.



Данный график показывает, что Отложенное слияние по прежнему выигрывает у Алгоритма двойников при большом количестве удалений. При малом количестве удалений Алгоритм Двойников выигрывает так как в Отложенном слиянии производится дополнительный подсчет глобально и локально свободных блоков, при увеличения количество данных время затраченное на подсчет становится меньше времени, затрачиваемого на обьединение блоков.

Теперь рассмотрим график, который показывает зависимость выделения и освобождения блоков от времени, при этом сначала происходит выделение всех блоков, а потом из освобождение. Размер страницы -1024 байт.



Данный график показывает, что Алгоритм Двойников при больших значениях выигрывает у Отложенного слияния. Это происходит из за последовательного выделения и удаления. Так как при последовательном освобождении при удалении блока подсчет глобально и локально свободных блоков в моей реализации будет производится чаще, чем при не последовательном слиянии. Следовательно обработка вектора страницы может не привести к критическому состоянию для слияния, и время потраченное на обработку будет влиять на время удаления, так как только после нескольких таких обработок будет происходить слияние, в итоге все равно приводящее к освобождению всего блока.

Выводы:

При выполнения данного курсового проекта я научилась реализовывать два алгоритма аллокации и сравнила их характеристики. В данной работе приведены графики для сравнения, листинг программы и их примеры

использования. При реализации данного курсового проекта я хранила данные в векторе, в Отложенном слиянии при удалении производятся частные обработки вектора, что влияет на производительность программы в целом, если использовать бинарное дерево для хранения блоков, то можно заметно улучшить производительность. Мне было очень интересно выполнять курсовой проект и производить анализ алгоритмов аллокации.

Литература:

1. Таненбаум Э., Бос Х. *Современные операционные системы. — 4-е изд.* — СПб.: Издательский дом «Питер», 2018.

2. Поисковик Google [электронный ресурс] URL: https://google.com/ (дата обращения: 24.12.2020)

3. Ю. Вахалия Unix изнутри — Издательский дом «Питер», 2003.