Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Курсовой проект по курсу
«Операционные системы»

Тема работы
“Аллокаторы памяти”

Студент: Зубко Дмитрий Валерьевич
Группа: М8О-208Б-20
Вариант: 13
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021

Содержание

Репозиторий

https://github.com/usernameMAI/OS

Постановка задачи

Исследование 2 аллокаторов памяти: 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.
13. Необходимо сравнить два алгоритма аллокации: списки свободных блоков (первое подходящее) и блоки по 2 в степени n.

Общие сведения о программе

Программа состоит из трёх файлов:

main.cpp

PoolAllocator.cpp

Degree2Allocator.cpp

Содержит makefile:

```
all:
	g++ main.cpp PoolAllocator.cpp Degree2Allocator.cpp -o main -fsanitize=address
clean:
	rm -rf main
```

Общий метод и алгоритм решения

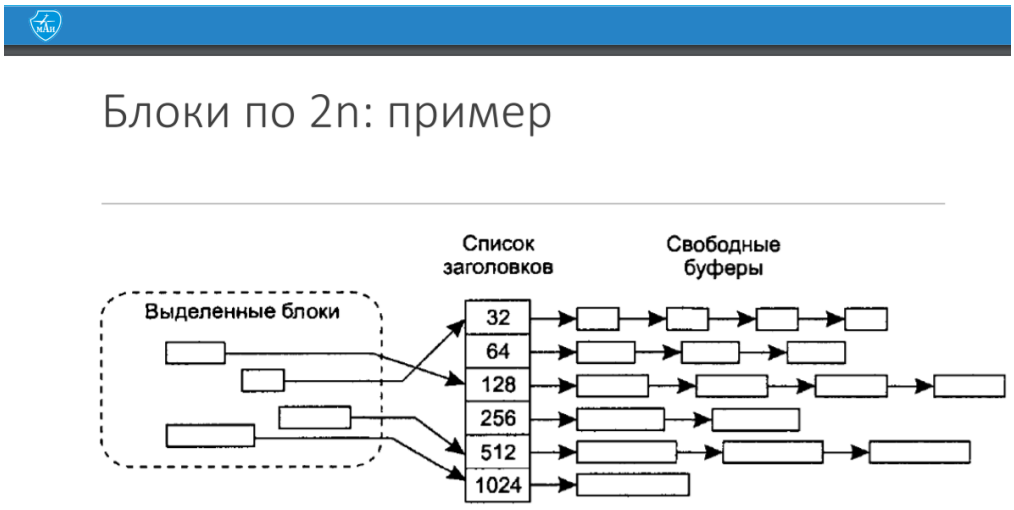
Аллокатор – менеджер памяти, который обрабатывает запросы на выделение и освобождение памяти.

1. Алгоритм аллокации через списки свободных блоков (первое подходящее).

Этот алгоритм отслеживает память с помощью связных списков распределенных и свободных сегментов памяти, где сегмент содержит либо свободную, либо занятую память. Каждый элемент списка хранит внутри своё обозначение – свободен ли он, размер участка памяти и указатель на его начало.

Алгоритм первого подходящего заключается в том, что мы проходим по списку сегментов, пока не будет найден блок памяти подходящего размера. После этого найденный блок разбивается на два: один подходящего размера, другой пустой.

2. Алгоритм аллокации через блоки по 2 в степени n.



В данном алгоритме размер буфера всегда кратен 2. Каждый свободный буфер хранит указатель на следующий свободный буфер, либо размер буфера, либо указатель на список, которому принадлежит буфер. Я реализовал так, что буфер содержит свой размер.

Когда поступает запрос на аллокацию памяти, к запрошенному размеру прибавляется размер памяти, необходимый для хранения размера буфера. Из списка, содержащего минимальные по размеру буферы, удовлетворяющие запросу, извлекается и удаляется любой элемент (для простоты это первый). Так как размер памяти был записан в буфер при инициализации, достаточно будет увеличить указатель на начало буфера на число байт, необходимое для хранения размера и вернуть его из функции.

Исходный код

PoolAllocator.cpp

```
#include <list>
#include <iostream>
#include <algorithm>

class PoolAllocator {

    struct Node {
        char *start;
        size_t capacity;
        bool availability;

        void Print() const {
            std::cout << "Node: capacity " << capacity << ", type " <<
(availability ? "Free" : "Busy") << std::endl;
        }

    };

public:

    PoolAllocator(size_t size) {
        data = (char *) malloc(size);
        Node node = {data, size, true};
        blocks.push_front(node);
    }

    ~PoolAllocator() {
        free(data);
    }

    void *alloc(size_t size) {

        if (size <= 0)
            return nullptr;
```

```
        size_t size_of_node = 0;
        auto needed_node = blocks.end();

        for (auto it = blocks.begin(); it != blocks.end(); ++it) {
            if (it->availability && it->capacity >= size && (size_of_node ==
0 || it->capacity < size_of_node)) {
                size_of_node = it->capacity;
                needed_node = it;
                break;
            }
        }

        if (size_of_node == 0) {
            std::cout << "no alloc" << std::endl;
            exit(1);
        }

        if (size == size_of_node) {
            needed_node->availability = false;
        } else {
            Node new_node = {needed_node->start + size, needed_node->capacity
- size, true};
            needed_node->capacity = size;
            needed_node->availability = false;
            blocks.insert(std::next(needed_node), new_node);
        }

        return (void *) (needed_node->start);

    }

    void dealloc(void *ptr) {

        auto it = std::find_if(blocks.begin(), blocks.end(), [ptr](const Node
&node) {
            return node.start == (char *) ptr && !node.availability;
        });

        if (it == blocks.end()) {
            std::cout << "no alloc" << std::endl;
            exit(1);
        }

        it->availability = true;

        if (it != blocks.begin() && std::prev(it)->availability) {
            auto prev_it = std::prev(it);
            prev_it->capacity += it->capacity;
            blocks.erase(it);
            it = prev_it;
        }

        if (std::next(it) != blocks.end() && std::next(it)->availability) {
            auto next_it = std::next(it);
            it->capacity += next_it->capacity;
            blocks.erase(next_it);
        }

    }

    void PrintStatus() const {
        int occ_sum = 0;
        int free_sum = 0;
```

```
        for (const Node &bl: blocks) {
            bl.Print();
            if (bl.availability) {
                free_sum += bl.capacity;
            } else {
                occ_sum += bl.capacity;
            }
        }

        std::cout << "Occupied memory " << occ_sum << std::endl;
        std::cout << "Free memory " << free_sum << std::endl << std::endl;

    }

private:
    char *data;
    std::list<Node> blocks;
};
```

Degree2Allocator.cpp

```
#include <vector>
#include <list>
#include <iostream>
#include <algorithm>

struct Degrees2{

    unsigned int bl_16 = 0;
    unsigned int bl_32 = 0;
    unsigned int bl_64 = 0;
    unsigned int bl_128 = 0;
    unsigned int bl_256 = 0;
    unsigned int bl_512 = 0;
};

class Degree2Allocator{

public:

    Degree2Allocator(const Degrees2& init_data):lists((index_to_size.size()))
{

        std::vector<unsigned int> mem_sizes = {
                init_data.bl_16, init_data.bl_32, init_data.bl_64,
init_data.bl_128, init_data.bl_256, init_data.bl_512
        };

        unsigned int sum = 0;
        for(int i = 0; i < mem_sizes.size(); ++i){
            sum += mem_sizes[i] * index_to_size[i];
        }

        data = (char*)malloc(sum);
        char* copy_data = data;

        for(int i = 0; i < mem_sizes.size(); ++i){
            for(int j = 0; j < mem_sizes[i]; ++j){
                lists[i].push_back(copy_data);
                *((int*)copy_data) = (int)index_to_size[i];
```

```
                copy_data += index_to_size[i];
            }
        }

        mem_size = sum;

    }

    ~Degree2Allocator(){
        free(data);
    };


    void* alloc(size_t mem_size){

        if(mem_size <= 0)
            return nullptr;

        mem_size += sizeof(int);
        int idx = -1;

        for(size_t i = 0; i < lists.size(); ++i){
            if(index_to_size[i] >= mem_size && !lists[i].empty()){
                idx = i;
                break;
            }
        }

        if(idx == -1){
            std::cout << "Error" << std::endl;
        }

        char* to_return = lists[idx].front();
        lists[idx].pop_front();
        return (void*)(to_return + sizeof(int));
    }

    void dealloc(void* ptr){
        char* c_ptr = (char*)(ptr);
        c_ptr = c_ptr - sizeof(int);
        int block_size = *((int*)c_ptr);
        int idx = std::lower_bound(index_to_size.begin(),
index_to_size.end(), block_size) - index_to_size.begin();

        if(idx == index_to_size.size()){
            std::cout << "Error alloc" << std::endl;
        }

        lists[idx].push_back(c_ptr);
    }

    void PrintStatus(){
        int free_sum = 0;

        for(size_t i = 0; i < lists.size(); ++i){
            std::cout << "List with" << index_to_size[i] << " byte blocks,
size: " << lists[i].size() << std::endl;
            free_sum += lists[i].size() * index_to_size[i];
        }

        int occ_sum = mem_size - free_sum;

        std::cout << "Occupied memory " << occ_sum << std::endl;
```

```
        std::cout << "Free memory " << free_sum << std::endl << std::endl;

    }

private:
    const std::vector<int> index_to_size = {16,32,64,128,256,512,1024};
    std::vector<std::list<char*>> lists;
    char* data;
    int mem_size;
};
```

Демонстрация работы программы

Проведём тестирование и сравнение двух аллокаторов:

Время выделения 4096 байт:

```
steady_clock::time_point list_allocator_init_start = steady_clock::now();
PoolAllocator list_allocator(4096);
steady_clock::time_point list_allocator_init_end = steady_clock::now();
std::cout << "List allocator initialization with one page of memory :"
          << std::chrono::duration_cast<std::chrono::nanoseconds>(
                  list_allocator_init_end -
list_allocator_init_start).count()
          << " ns" << std::endl;


steady_clock::time_point d2_allocator_init_start = steady_clock::now();
Degrees2 d = {64, 32, 16, 4, 2};
Degree2Allocator d2_allocator(d); // 1 страница
steady_clock::time_point d2_allocator_init_end = steady_clock::now();
std::cout << "D2 allocator initialization with one page of memory :"
          << std::chrono::duration_cast<std::chrono::nanoseconds>(
                  d2_allocator_init_end - d2_allocator_init_start).count()
          << " ns" << std::endl;

std::cout << "\n";
```

Вывод:

List allocator initialization with one page of memory :19800 ns

D2 allocator initialization with one page of memory :43000 ns

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

```
std::vector<char *> pointers(15, 0);
Degrees2 d = {0, 0, 32, 20, 20, 10};
Degree2Allocator allocator(d);
steady_clock::time_point n2_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 n2_test1_end = steady_clock::now();
std::cout << "D2 allocator first test:"
          <<
std::chrono::duration_cast<std::chrono::microseconds>(n2_test1_end -
n2_test1_start).count()
          << " microseconds" << std::endl;
allocator.PrintStatus();
for (int i = 0; i < 15; ++i) {
    allocator.dealloc(pointers[i]);
}
```

```
PoolAllocator allocator(4096);
std::vector<char *> pointers(1000, 0);
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();
std::cout << "List allocator first test:"
          << std::chrono::duration_cast<std::chrono::microseconds>(test1_end
- test1_start).count()
          << " microseconds" << std::endl;
allocator.PrintStatus();
for (int i = 0; i < 15; ++i) {
    allocator.dealloc(pointers[i]);
}
```

Вывод:

D2 allocator first test:3 microseconds
List with16 byte blocks, size: 0
List with32 byte blocks, size: 0
List with64 byte blocks, size: 32
List with128 byte blocks, size: 20
List with256 byte blocks, size: 10
List with512 byte blocks, size: 5
List with1024 byte blocks, size: 0
Occupied memory 5120
Free memory 9728

List allocator first test:11 microseconds
Node: capacity 256, type Busy
Node: capacity 256, type Busy
Node: capacity 256, type Busy
Node: capacity 256, type Busy
Node: capacity 256, type Busy
Node: capacity 128, type Busy
Node: capacity 128, type Busy

Node: capacity 128, type Busy
Node: capacity 128, type Busy
Node: capacity 128, type Busy
Node: capacity 128, type Busy
Node: capacity 128, type Busy
Node: capacity 128, type Busy
Node: capacity 128, type Busy
Node: capacity 128, type Busy
Node: capacity 1536, type Free
Occupied memory 2560
Free memory 1536

Second test: Allocate and free 750 20 bytes arrays:

```
Degrees2 d = {0, 400, 400};
Degree2Allocator allocator(d);
std::vector<char *> pointers(750, 0);
steady_clock::time_point alloc_start = steady_clock::now();
for (int i = 0; i < 750; ++i) {
    pointers[i] = (char *) allocator.alloc(20);
}
steady_clock::time_point alloc_end = steady_clock::now();
for (int i = 0; i < 750; ++i) {
    allocator.dealloc(pointers[i]);
}
steady_clock::time_point test_end = steady_clock::now();
std::cout << "D2 allocator second test:\n"
          << "Allocation :" <<
duration_cast<std::chrono::microseconds>(alloc_end - alloc_start).count()
          << " microseconds" << "\n"
          << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end - alloc_end).count()
          << " microseconds" << "\n";
```

```
PoolAllocator allocator(16000);
std::vector<char *> pointers(750, 0);
steady_clock::time_point alloc_start = steady_clock::now();
for (int i = 0; i < 750; ++i) {
    pointers[i] = (char *) allocator.alloc(20);
}
steady_clock::time_point alloc_end = steady_clock::now();
for (int i = 0; i < 750; ++i) {
    allocator.dealloc(pointers[i]);
}
steady_clock::time_point test_end = steady_clock::now();
std::cout << "List allocator second test:\n"
          << "Allocation :" <<
duration_cast<std::chrono::microseconds>(alloc_end - alloc_start).count()
          << " microseconds" << "\n"
          << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end - alloc_end).count()
          << " microseconds" << "\n";
```

Вывод:

D2 allocator second test:

Allocation :62 microseconds
Deallocation :101 microseconds
List allocator second test:
Allocation :3060 microseconds
Deallocation :132 microseconds

Third test: Allocate 500 20 bytes arrays, deallocate every second, allocate 250 12 bytes:

```
Degrees2 d = {400, 700};
Degree2Allocator allocator(d);
std::vector<char *> pointers(750, 0);
steady_clock::time_point test_start = steady_clock::now();
for (int i = 0; i < 500; ++i) {
    pointers[i] = (char *) allocator.alloc(20);
}
for (int i = 0; i < 250; ++i) {
    allocator.dealloc(pointers[i * 2]);
}
for (int i = 500; i < 750; ++i) {
    pointers[i] = (char *) allocator.alloc(12);
}
steady_clock::time_point test_end = steady_clock::now();
std::cout << "D2 allocator third test:"
          << std::chrono::duration_cast<std::chrono::microseconds>(test_end -
test_start).count()
          << " microseconds" << std::endl;
allocator.PrintStatus();
for (int i = 0; i < 250; ++i) {
    allocator.dealloc(pointers[i * 2 + 1]);
}
for (int i = 500; i < 750; ++i) {
    allocator.dealloc(pointers[i]);
}
```

```
PoolAllocator allocator(16000);
std::vector<char *> pointers(750, 0);
steady_clock::time_point test_start = steady_clock::now();
for (int i = 0; i < 500; ++i) {
    pointers[i] = (char *) allocator.alloc(20);
}
for (int i = 0; i < 250; ++i) {
    allocator.dealloc(pointers[i * 2]);
}
for (int i = 500; i < 750; ++i) {
    pointers[i] = (char *) allocator.alloc(12);
}
steady_clock::time_point test_end = steady_clock::now();
std::cout << "\nList allocator third test:"
          << std::chrono::duration_cast<std::chrono::microseconds>(test_end -
test_start).count()
          << " microseconds" << std::endl;
allocator.PrintStatus();
for (int i = 0; i < 250; ++i) {
    allocator.dealloc(pointers[i * 2 + 1]);
}
for (int i = 500; i < 750; ++i) {
```

```
    allocator.dealloc(pointers[i]);
}
```

Вывод:
D2 allocator third test:83 microseconds
List with16 byte blocks, size: 150
List with32 byte blocks, size: 450
List with64 byte blocks, size: 0
List with128 byte blocks, size: 0
List with256 byte blocks, size: 0
List with512 byte blocks, size: 0
List with1024 byte blocks, size: 0
Occupied memory 12000
Free memory 16800

List allocator third test:3077 microseconds
Node: capacity 12, type Busy
Node: capacity 8, type Free
….
Node: capacity 12, type Busy
Node: capacity 8, type Free
Node: capacity 8, type Free
Node: capacity 20, type Busy
Node: capacity 6000, type Free
Occupied memory 8000
Free memory 8000

Fourth test: Allocate and free 1500 20 bytes arrays:

```
Degrees2 d = {0, 800, 800};
Degree2Allocator allocator(d);
std::vector<char *> pointers(1500, 0);
steady_clock::time_point alloc_start = steady_clock::now();
for (int i = 0; i < 1500; ++i) {
    pointers[i] = (char *) allocator.alloc(20);
}
steady_clock::time_point alloc_end = steady_clock::now();
for (int i = 0; i < 1500; ++i) {
    allocator.dealloc(pointers[i]);
}
steady_clock::time_point test_end = steady_clock::now();
std::cout << "D2 allocator fourth test:\n"
          << "Allocation :" <<
duration_cast<std::chrono::microseconds>(alloc_end - alloc_start).count()
          << " microseconds" << "\n"
          << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end - alloc_end).count()
          << " microseconds" << "\n";
```

```
PoolAllocator allocator(32000);
std::vector<char *> pointers(1500, 0);
steady_clock::time_point alloc_start = steady_clock::now();
for (int i = 0; i < 1500; ++i) {
    pointers[i] = (char *) allocator.alloc(20);
}
steady_clock::time_point alloc_end = steady_clock::now();
for (int i = 0; i < 1500; ++i) {
    allocator.dealloc(pointers[i]);
}
steady_clock::time_point test_end = steady_clock::now();
std::cout << "List allocator fourth test:\n"
          << "Allocation :" <<
duration_cast<std::chrono::microseconds>(alloc_end - alloc_start).count()
          << " microseconds" << "\n"
          << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end - alloc_end).count()
          << " microseconds" << "\n";
```

Вывод:

D2 allocator fourth test:
Allocation :126 microseconds
Deallocation :238 microseconds
List allocator fourth test:
Allocation :11164 microseconds
Deallocation :248 microseconds

Время, требуемое для инициализации больше у алгоритма на списках степени 2. Это происходит потому что этому алгоритму требуется время для инициализации заголовков блоков.

1 тест:

Данный тест показывает, как неэффективно расходует память аллоктор, основанный на степенях 2. Ему потребовалось в 2 раза больше памяти. Причина – выделение этим аллокатором блоков фиксированного размера, без возможности разбить их на более мелкие.

2 тест:

Данный тест показывает, что аллокатор на степенях двойки справляется намного быстрее. Причина – аллокатору свободных блоков требуется пройтись по всему списку в поиске сегмента памяти.

3 тест:

Данный тест показывает, как быстро может произойти фрагментация в аллокаторе, основанном на поиске первого подходящего блока. Хотя во втором случае проблема фрагментации сильно также не наблюдается, но всё же не является эффективной.

4 тест:

Этот тест аналогичен второму, но он показывает, как ведут себя аллокаторы при увеличении количества входных данных в два раза. Время аллокации “списки степени 2” увеличилось примерно в 2 раза. Время аллокации алгоритма “первое подходящее” увеличилось примерно в 4 раза.

Выводы

Данный курсовой проект познакомил меня аллокаторами памяти, их видами. Я закрепил свои знания о представлении памяти. Научился исследовать их. Оказалось, что аллокатор на списке свободных блоков в целом проигрывает аллокаторы на блоках в степени n.