

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

**Приобретение практических навыков в использовании
знаний, полученных в течении курса.**

Студент: Рылов Александр Дмитриевич

Группа: М8О-207Б-21

Вариант: 16

Преподаватель: Миронов Евгений Сергеевич

Оценка: _____

Дата: _____

Подпись: _____

Москва, 2022

Содержание

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Демонстрация работы программы
7. Выводы

Репозиторий

<https://github.com/Brokiloene/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

Вариант 16: Необходимо сравнить два алгоритма аллокации: алгоритм Мак-Кьюзи-Кэрелса и блоки по 2 в степени n .

Общие сведения о программе

Файлы `row_of_two.hpp` и `mckk.hpp` содержат интерфейсы классов аллокаторов, файлы `row_of_two.cpp` и `mckk.cpp` — их реализацию. Файл `main.cpp` содержит использование аллокаторов и демонстрацию их работы, сборка осуществляется с помощью утилиты `make`.

Общий метод и алгоритм решения

Алгоритм аллокации блоков степени 2 заключается в хранении списков указателей на свободные блоки памяти одного размера, размер блока хранится в самом блоке. Алгоритм Мак-Кьюзи-Кэрелса является улучшенной версией предыдущего, в нём свободные блоки так же хранятся в списках, но размер блока хранится на уровне самого аллокатора в специальном массиве.

Исходный код

main.cpp

```
#include <iostream>
```

```

#include <chrono>

#include "pow_of_two_allocator.hpp"
#include "mckk.hpp"

using namespace std::chrono;

int main(){
    std::vector<int> blocks_amount = { 64, 32, 16, 4, 20, 10, 0};

    steady_clock::time_point pow2_init_start = steady_clock::now();
    pow_two_allocator pow_two_allocator(blocks_amount);
    steady_clock::time_point pow2_init_end = steady_clock::now();

    std::cout << "powers of two init time: " <<
    std::chrono::duration_cast<std::chrono::nanoseconds>(pow2_init_end -
pow2_init_start).count() << " nanosec\n";

    int pages_cnt = 10;
    std::vector<int> pages_fragments = { 32, 128, 256, 1024, 512, 256, 256, 1024,
16, 256};

    steady_clock::time_point mckk_init_start = steady_clock::now();
    mckk_allocator mckk_allocator(pages_cnt, pages_fragments);
    steady_clock::time_point mckk_init_end = steady_clock::now();

    std::cout << "mckk init time: " <<
    std::chrono::duration_cast<std::chrono::nanoseconds>(mckk_init_end -
mckk_init_start).count() << " nanosec\n\n";

```

```

std::cout << "test: allocate 10 char[256] + deallocate + allocate 10 char[128] +
deallocate:\n";

std::vector<char *> pointers1(10, 0);
steady_clock::time_point pow2_test_start = steady_clock::now();
for (int i = 0; i < 10; ++i) {
    pointers1[i] = (char *)pow_two_allocator.allocate(256);
}
for (int i = 0; i < 10; ++i) {
    pow_two_allocator.deallocate(pointers1[i]);
}
for (int i = 5; i < 10; ++i) {
    pointers1[i] = (char *)pow_two_allocator.allocate(128);
}
for (int i = 0; i < 10; ++i) {
    pow_two_allocator.deallocate(pointers1[i]);
}
steady_clock::time_point pow_two_test_end = steady_clock::now();
std::cerr << "powers of two: " <<
std::chrono::duration_cast<std::chrono::microseconds>(pow_two_test_end -
pow2_test_start).count() << " microsec\n";

```

```

std::vector<char *> pointers2(10, 0);
steady_clock::time_point mckk_test_start = steady_clock::now();
for (int i = 0; i < 10; ++i) {
    pointers2[i] = (char *)mckk_allocator.allocate(256);
}
for (int i = 5; i < 10; ++i) {
    mckk_allocator.deallocate(pointers2[i]);
}

```

```

    for (int i = 5; i < 10; ++i) {
        pointers2[i] = (char *)mckk_allocator.allocate(128);
    }
    for (int i = 0; i < 10; ++i) {
        mckk_allocator.deallocate(pointers2[i]);
    }
    steady_clock::time_point mckk_test_end = steady_clock::now();
    std::cerr << "mckk: " <<
std::chrono::duration_cast<std::chrono::microseconds>(mckk_test_end -
mckk_test_start).count() << " microsec\n";
}

```

mckk.cpp

```

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

#define PAGE_SIZE 1024

struct Page {
    int block_size;
    char *start;
    char *end;
};

class mckk_allocator {
private:
    std::vector<int> pows_of_two = { 16,32,64,128,256,512,1024};
6

```

```

std::vector<std::list<char *>> free_blocks_lists;
std::vector<Page> block_mem_info;
char *data;

public:
    mckk_allocator(int pages_cnt, std::vector<int>& pages_fragments); //
    pages_fragments is a vector with sizes of blocks
                                                    // on which page is splitted

    void *allocate(int bytes_amount);
    void deallocate(void *ptr);

    ~mckk_allocator();

};

```

pow_of_two.cpp

```
#include "pow_of_two.hpp"
```

```

pow_two_allocator::pow_two_allocator(std::vector<int> &blocks_amount) {
    free_blocks_lists = std::vector<std::list<char *>>(pows_of_two.size());
    int bytes_sum = 0;
    for (int i = 0; i < blocks_amount.size(); ++i) {
        bytes_sum += blocks_amount[i] * pows_of_two[i];
    }
    data = (char *) malloc(bytes_sum);
    char *data_copy = data;
    for (int i = 0; i < blocks_amount.size(); ++i) {
        for (int j = 0; j < blocks_amount[i]; ++j) {
            free_blocks_lists[i].push_back(data_copy);
            *((int *)data_copy) = pows_of_two[i];

```

```

        data_copy += pows_of_two[i];
        //std::cout << pows_of_two[i] << '\n';
    }
}
}

void *pow_two_allocator::allocate(int bytes_amount) {
    if (bytes_amount == 0) {
        return nullptr;
    }
    bytes_amount += sizeof(int);
    //std::cout << "free_blocks_lists.size = " << free_blocks_lists.size() << std::endl;
    /*for (auto el : free_blocks_lists){
        std::cout << el.size() << std::endl;
    }*/
    int ind = -1;
    for (int i = 0; i < free_blocks_lists.size(); ++i) {
        if (bytes_amount <= pows_of_two[i] && !free_blocks_lists[i].empty()) { // if
requested amount of bytes is fit and such block exists
            ind = i;
            break;
        }
    }
    if (ind == -1) {
        std::cout << "There isn't memory\n";
    }

    char *memory = free_blocks_lists[ind].front();
    free_blocks_lists[ind].pop_front();
    return (void *)(memory + sizeof(int));
}

```



```
}
```

```
void pow_two_allocator::deallocate(void *ptr) {  
    char *char_ptr = (char *)ptr;  
    char_ptr = char_ptr - sizeof(int);  
    int block_size = *((int *)char_ptr);  
    int ind = -1;  
    for(int i = 0; i < pows_of_two.size(); ++i) {  
        if(pows_of_two[i] == block_size) {  
            ind = i;  
        }  
    }  
  
    free_blocks_lists[ind].push_back(char_ptr);  
}
```

```
pow_two_allocator::~~pow_two_allocator() {  
    free(data);  
}
```

pow_of_two.hpp

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

```
class pow_two_allocator {  
private:  
    std::vector<std::list<char *>> free_blocks_lists;  
    std::vector<int> pows_of_two = { 16,32,64,128,256,512,1024};  
    char *data;
```

public:

```
pow_two_allocator(std::vector<int> &blocksAmount);
```

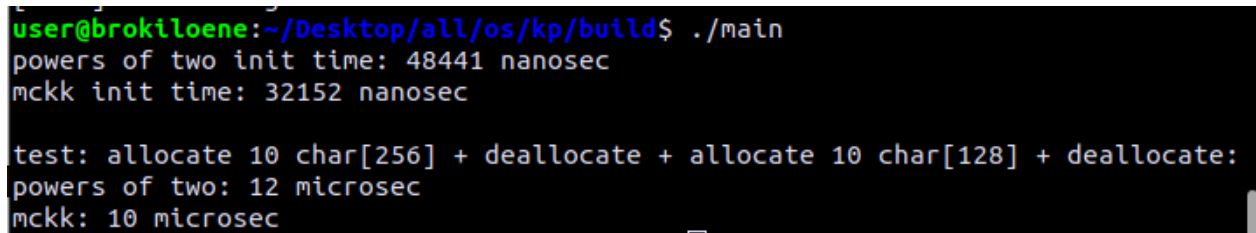
```
void *allocate(int bytesAmount);
```

```
void deallocate(void *ptr);
```

```
~pow_two_allocator();
```

```
};
```

Демонстрация работы программы

A screenshot of a terminal window with a black background and green and white text. The prompt is 'user@brokiloene:~/Desktop/all/os/kp/build\$'. The command executed is './main'. The output shows initialization times for 'powers of two' (48441 nanosec) and 'mckk' (32152 nanosec). A test case is shown: 'test: allocate 10 char[256] + deallocate + allocate 10 char[128] + deallocate:', with results 'powers of two: 12 microsec' and 'mckk: 10 microsec'.

```
user@brokiloene:~/Desktop/all/os/kp/build$ ./main
powers of two init time: 48441 nanosec
mckk init time: 32152 nanosec

test: allocate 10 char[256] + deallocate + allocate 10 char[128] + deallocate:
powers of two: 12 microsec
mckk: 10 microsec
```

Выводы

В ходе выполнения курсового проекта я закрепил навыки работы с аллокаторами. Из результатов работы программы видно, что инициализация и аллокация алгоритмом Мак-Кьюзи-Кэрелса происходит немного быстрее и поэтому является более предпочтительным алгоритмом аллокации памяти.