Московский Авиационный Институт

(Национальный Исследовательский институт)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовая работа по курсу

«Операционные системы»

«Аллокаторы памяти»

Студент:Литовченко Анна Александровна
Группа: М8О-207Б-21
Вариант: 13
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Содержание

Репозиторий

https://github.com/Annalitov/OS/kp

Постановка задачи

Приобретение практических навыков в использовании знаний, полученных в течение курса, проведения исследования в выбранной предметной области.

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

Алгоритмы для сравнения:

- списки свободных блоков(первое подходящее)
- блоки по 2 в степени N

Сведения о программе

Программа написана на C++, операционная система MacOs. Состоит из двух пар cpp/h файлов для каждого аллокатора и тестирующего исполняемого файла main.cpp.

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. Для данной курсовой работы я рассматривала два алгоритма аллокации:

- Алгоритм с выбором первого подходящего участка памяти, основанный на списках
- Простые списки, основанные на степени двойки.

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.

Наиболее подходящее — 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 и менее расточительно, чем «наиболее подходящее». Также, он несколько производительнее, чем « следующие подходящее». 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

Списки, основанные на степени двойки

Списки, основанные на степени числа 2, чаще всего применяются для реализации процедур выделения и освобождения памяти в библиотеке С прикладного уровня. Эта методика использует набор списков свободной памяти, в каждом из которых хранятся буферы определенного размера. Размер буфера всегда кратен степени двойки

При этом каждый свободный буфер хранит в себе либо указатель на следующий свободный буфер, либо размер буфера, либо указатель на список, которому принадлежит буфер(в моей реализации буфер содержит свой размер).

При поступлении запроса 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 будем извлекать первый). Размер памяти был записан в буфер при инициализации, так что достаточно будет увеличить указатель на начало буфера на число байт, необходимое для хранения размера и вернуть его из функции. При операции освобождения, буфер освобождается только целиком, нет возможности частично освободить буфер

Данный алгоритм является весьма простым и быстрым. Этот аллокатор предоставляет понятный программный интерфейс, важнейшим преимуществом которого является процедура освобождения, а именно то, что в нее не нужно передавать размер буфера. Однако алгоритм имеет несколько значимых недостатков. Округление количества памяти вверх до ближайшей степени двойки приводит к неэкономному распределению памяти. Так же проблемой являются запросы, равные или очень близкие к очередной степени двойки. Необходимость хранения заголовка буфера влечет к перерасходу выделенной памяти в 2 раза. Также не поддерживается слияние смежных буферов.

Исходный код

RMAllocator.h

```
#pragma once
#include <cinttypes>
using namespace std;
class RMAllocator
{
private:
	struct RMNode
	{
		RMNode* next_block_ptr;
		size_t current_block_size;
	};

	const size_t mem_page_size = 1024;
	uint8_t* mem_page_ptr;
	RMNode* first_block_ptr;

public:
	RMAllocator();
	virtual ~RMAllocator();

	void* malloc(const size_t size);
	void free(const void* ptr, const size_t size);
	void print();
	void defragment();

};
```

RMAllocator.cpp

```
#include <iostream>
#include <cinttypes>
#include "RMAllocator.h"

using namespace std;
```

```
RMAllocator::RMAllocator()
{

	mem_page_ptr = (uint8_t*)::malloc(mem_page_size);

	if (mem_page_ptr == nullptr)
		throw runtime_error("Error: Cannot allocate a memory page");

	first_block_ptr = (RMNode*)mem_page_ptr;
	first_block_ptr->next_block_ptr = nullptr;
	first_block_ptr->current_block_size = mem_page_size;
}

RMAllocator::~RMAllocator()
{
	if (mem_page_ptr != nullptr)
		::free(mem_page_ptr);
}

void* RMAllocator::malloc(const size_t size)
{

	if (size == 0)
		return nullptr;

	const size_t requested_size = max(size, sizeof(RMNode));


	RMNode* block_ptr = first_block_ptr;


	while (block_ptr != nullptr)
	{
```

```
            if (block_ptr->current_block_size >= requested_size + sizeof(RMNode))
            {

                uint8_t* busy_block_ptr = (uint8_t*)block_ptr +
block_ptr->current_block_size - requested_size;

                block_ptr->current_block_size -= requested_size;
                return busy_block_ptr;
            }

            block_ptr = block_ptr->next_block_ptr;
        }


        throw runtime_error("Error: Cannot allocate memory of requested size");
}

void RMAllocator::free(const void* ptr, const size_t size)
{
        if (ptr == nullptr)
            return;


        const size_t freed_size = max(size, sizeof(RMNode));


        RMNode* freed_block_ptr = (RMNode*)ptr;
        freed_block_ptr->next_block_ptr = nullptr;
        freed_block_ptr->current_block_size = size;


        RMNode** previous_ptr_ptr = &first_block_ptr;
        RMNode* block_ptr = first_block_ptr;
```

```
        while (block_ptr != nullptr)
        {

                if (ptr < block_ptr)
                {
                        freed_block_ptr->next_block_ptr = block_ptr;
                        break;
                }


                previous_ptr_ptr = &block_ptr->next_block_ptr;
                block_ptr = block_ptr->next_block_ptr;
        }
        *previous_ptr_ptr = freed_block_ptr;

    defragment();
}

void RMAllocator::print()
{
        cout << "RMAllocator statistics:" << endl;
        cout << "memory page pointer = " << (void*)mem_page_ptr << endl;
        cout << "memory page size = " << mem_page_size << endl;

        int num = 0;
        RMNode* block_ptr = first_block_ptr;
        while (block_ptr != nullptr)
        {
                cout << "free block #" << num << " pointer = " << (void*)block_ptr <<
endl;
                cout << "free block #" << num << " size = " <<
block_ptr->current_block_size << endl;
                num++;
```

```
            block_ptr = block_ptr->next_block_ptr;
        }
        cout << "free blocks total = " << num << endl;
        cout << endl;
}

void RMAllocator::defragment()
{

        RMNode* block_ptr = first_block_ptr;

        while (block_ptr != nullptr)
        {
            while (block_ptr != nullptr)
            {
                uint8_t* end_of_block_ptr = (uint8_t*)block_ptr +
block_ptr->current_block_size;
                if (end_of_block_ptr != (uint8_t*)block_ptr->next_block_ptr)
                    break;

                block_ptr->current_block_size +=
block_ptr->next_block_ptr->current_block_size;
                block_ptr->next_block_ptr =
block_ptr->next_block_ptr->next_block_ptr;
            }

            block_ptr = block_ptr->next_block_ptr;
        }
}
```

N2Allocator.h

```
#pragma once

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

```
#include <algorithm>
#include <vector>

struct N2AllocatorInit {
    unsigned int block_16 = 0;
    unsigned int block_32 = 0;
    unsigned int block_64 = 0;
    unsigned int block_128 = 0;
    unsigned int block_256 = 0;
    unsigned int block_512 = 0;
    unsigned int block_1024 = 0;
};

class N2Allocator {
public:
    explicit N2Allocator(const N2AllocatorInit& init_data);

    ~N2Allocator();
    void* allocate(size_t mem_size);
    void deallocate(void *ptr);
    void PrintStatus(std::ostream& os) const;

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;
};
```

N2Allocator.cpp

```
#include "N2Allocator.h"

N2Allocator::N2Allocator(const N2AllocatorInit &init_data)
        : lists(index_to_size.size()) {
    std::vector<unsigned int> mem_sizes = {init_data.block_16,
```

```
                                    init_data.block_32,
                                    init_data.block_64,
                                    init_data.block_128,
                                    init_data.block_256,
                                    init_data.block_512,
                                    init_data.block_1024};
    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 *data_copy = data;
    for (int i = 0; i < mem_sizes.size(); ++i) {
        for (int j = 0; j < mem_sizes[i]; ++j) {
            lists[i].push_back(data_copy);
            *((int*)data_copy) = (int)index_to_size[i];
            data_copy += index_to_size[i];
        }
    }
    mem_size = sum;

}

N2Allocator::~N2Allocator() {
    free(data);
}

void *N2Allocator::allocate(size_t mem_size) {
    if (mem_size == 0) {
        return nullptr;
    }
    mem_size += sizeof(int);
    int index = -1;
    for (int i = 0; i < lists.size(); ++i) {
```

```
        if (index_to_size[i] >= mem_size && !lists[i].empty()) {
            index = i;
            break;
        }
    }
    if (index == -1) {
        throw std::bad_alloc();
    }
    char *to_return = lists[index].front();
    lists[index].pop_front();
    return (void*)(to_return + sizeof(int));
}

void N2Allocator::deallocate(void *ptr) {
    char *c_ptr = (char *) (ptr);
    c_ptr = c_ptr - sizeof(int);
    int block_size = *((int*)c_ptr);
    int index = std::lower_bound(index_to_size.begin(), index_to_size.end(),
block_size) - index_to_size.begin();
    if (index == index_to_size.size()) {
        throw std::logic_error("this pointer wasnt allocated by this allocator");
    }
    lists[index].push_back(c_ptr);
}

void N2Allocator::PrintStatus(std::ostream& os) const {
    int free_sum = 0;
    for (int i = 0; i < lists.size(); ++i) {
        os << "List with " << index_to_size[i] << " byte blocks, size: " << lists[i].size()
<< "\n";
        free_sum += lists[i].size() * index_to_size[i];
    }
    int occ_sum = mem_size - free_sum;
```

```
    os << "Occupied memory " << occ_sum << "\n";
    os << "Free memory " << free_sum << "\n\n";
}
```

main.cpp

```
#include <iostream>
#include <chrono>
#include <vector>
#include "RMAllocator.h"
#include "N2Allocator.h"


using namespace std;

int main()
{

        RMAllocator a1;
        vector<void *> pointer(10);

        std::chrono::steady_clock::time_point alloc_start1 =
std::chrono::steady_clock::now();
        for(int i = 0; i < 10; i++) {
                pointer[i] = a1.malloc(10);
        }
        std::chrono::steady_clock::time_point alloc_end1 =
std::chrono::steady_clock::now();

        for(int i = 0; i < 10; i++) {
                a1.free(pointer[i], 10);
        }
        a1.defragment();
        std::chrono::steady_clock::time_point test_end1 =
std::chrono::steady_clock::now();
        cerr << "RMAllocator first test:\n"
```

```
            << "Allocation :" <<
duration_cast<std::chrono::microseconds>(alloc_end1 - alloc_start1).count() << "
microseconds" << "\n"
            << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end1 - alloc_end1).count() << "
microseconds" << "\n\n";

      N2Allocator allocator({.block_16 = 2, .block_32 = 800, .block_64 = 800});
    vector<char *> pointers(10, 0);
    std::chrono::steady_clock::time_point alloc_start =
std::chrono::steady_clock::now();
    for (int i = 0; i < 10; ++i) {
       pointers[i] = (char *) allocator.allocate(10);
    }
    std::chrono::steady_clock::time_point alloc_end =
std::chrono::steady_clock::now();
    for (int i = 0; i < 10; ++i) {
       allocator.deallocate(pointers[i]);
    }
    std::chrono::steady_clock::time_point test_end =
std::chrono::steady_clock::now();
    cout << "N2 allocator first test:" << endl;
            cout << "Allocation :" <<
duration_cast<std::chrono::microseconds>(alloc_end - alloc_start).count() << "
microseconds" << endl;
            cout << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end - alloc_end).count() << "
microseconds" << endl << endl;

RMAllocator a2;
      vector<void *> pointer2(30);

      std::chrono::steady_clock::time_point alloc_start2 =
std::chrono::steady_clock::now();
      for(int i = 0; i < 30; i++) {
```

```
            pointer2[i] = a2.malloc(10);
        }
        std::chrono::steady_clock::time_point alloc_end2 =
std::chrono::steady_clock::now();

        for(int i = 0; i < 30; i++) {
            a2.free(pointer2[i], 10);
        }
        a2.defragment();
        std::chrono::steady_clock::time_point test_end2 =
std::chrono::steady_clock::now();
        cerr << "RMAllocator second test:\n"
             << "Allocation :" <<
duration_cast<std::chrono::microseconds>(alloc_end2 - alloc_start2).count() << "
microseconds" << "\n"
             << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end2 - alloc_end2).count() << "
microseconds" << "\n\n";



        N2Allocator allocator2({.block_16 = 100, .block_32 = 800, .block_64 = 800});
      vector<char *> pointers2(30, 0);
      std::chrono::steady_clock::time_point alloc_start3 =
std::chrono::steady_clock::now();
      for (int i = 0; i < 30; ++i) {
        pointers2[i] = (char *) allocator2.allocate(10);
      }
      std::chrono::steady_clock::time_point alloc_end3 =
std::chrono::steady_clock::now();
      for (int i = 0; i < 30; ++i) {
        allocator2.deallocate(pointers2[i]);
      }
      std::chrono::steady_clock::time_point test_end3 =
std::chrono::steady_clock::now();
      cout << "N2 allocator second test:" << endl;
```

```
        cout << "Allocation :" <<
duration_cast<std::chrono::microseconds>(alloc_end3 - alloc_start3).count() << "
microseconds" << endl;
        cout << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end3 - alloc_end3).count() << "
microseconds" << endl << endl;

RMAllocator a3;
    vector<void *> pointer3(60);

    std::chrono::steady_clock::time_point alloc_start5 =
std::chrono::steady_clock::now();
    for(int i = 0; i < 60; i++) {
        pointer3[i] = a3.malloc(10);
    }
    std::chrono::steady_clock::time_point alloc_end5 =
std::chrono::steady_clock::now();

    for(int i = 0; i < 60; i++) {
        a3.free(pointer3[i], 10);
    }
    a3.defragment();
    std::chrono::steady_clock::time_point test_end5 =
std::chrono::steady_clock::now();
    cerr << "RMAllocator third test:\n"
        << "Allocation :" <<
duration_cast<std::chrono::microseconds>(alloc_end5 - alloc_start5).count() << "
microseconds" << "\n"
        << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end5 - alloc_end5).count() << "
microseconds" << "\n\n";

    N2Allocator allocator3({.block_16 = 100, .block_32 = 800, .block_64 = 800});
   vector<char *> pointers3(60, 0);
```

```
    std::chrono::steady_clock::time_point alloc_start6 =
std::chrono::steady_clock::now();
    for (int i = 0; i < 60; ++i) {
        pointers3[i] = (char *) allocator3.allocate(10);
    }
    std::chrono::steady_clock::time_point alloc_end6 =
std::chrono::steady_clock::now();
    for (int i = 0; i < 60; ++i) {
        allocator3.deallocate(pointers3[i]);
    }
    std::chrono::steady_clock::time_point test_end6 =
std::chrono::steady_clock::now();
    cout << "N2 allocator third test:" << endl;
            cout << "Allocation :" <<
duration_cast<std::chrono::microseconds>(alloc_end6 - alloc_start6).count() << "
microseconds" << endl;
            cout << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end6 - alloc_end6).count() << "
microseconds" << endl << endl;

}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.15)
project(kursach)
set(CMAKE_CXX_STANDARD 14)
add_executable(os_kp
    main.cpp
    N2Allocator.cpp
    N2Allocator.h
    RMAllocator.cpp
    RMAllocator.h)
```

Тестирование

Для тестирования будем засекать время с помощью библиотеки chrono. Будем сравнивать сколько времени понадобилось для аллокации и деаллокации памяти у данных двух алгоритмов.

RMAllocator first test:
Allocation :0 microseconds
Deallocation :1 microseconds

N2 allocator first test:
Allocation :2 microseconds
Deallocation :6 microseconds

RMAllocator second test:
Allocation :1 microseconds
Deallocation :5 microseconds

N2 allocator second test:
Allocation :5 microseconds
Deallocation :17 microseconds

RMAllocator third test:
Allocation :2 microseconds
Deallocation :18 microseconds

N2 allocator third test:
Allocation :9 microseconds
Deallocation :32 microseconds

Результаты тестов

> Таким образом можно сделать вывод, что процессы аллокации и деаллокации памяти у алгоритма «списки свободных блоков(первое подходящее)» проходят быстрее, чем у алгоритма основанного на степенях двойки.

Заключение

При выполнении курсовой работы я познакомилась с несколькими видами аллокаторов, а так же более подробно исследовал два из них.

Благодаря этой работе я чуть лучше поняла принципы и особенности работы UNIX систем.