Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовая работа по курсу

«Операционные системы»

Группа: М8О-209Б-22

Студент: Концебалов О.С.

Преподаватель: Пономарев Н.В.

Оценка: _____

Дата: 25.12.2023

Москва, 2023.

Содержание

Постановка задачи

Необходимо спроектировать и реализовать программный прототип в соответствии с выбранным вариантом. Провести анализ, сделать вывод на основании данных, полученных при работе программного прототипа.

Вариант курсового проекта: Аллокаторы памяти

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.

В отчете необходимо отобразить следующее:

- Подробное описание каждого из исследуемых алгоритмов
- Процесс тестирования
- Обоснование подхода тестирования
- Результаты тестирования
- Заключение по проведенной работе

Сравнение алгоритмов аллокаторов памяти (детальное описание задания описано выше). Каждый аллокатор должен обладать следующим интерфейсом (могут быть отличия в зависимости от особенностей алгоритма):

- Allocator* createMemoryAllocator(void *realMemory, size_t memory_size) (создание аллокатора памяти размера memory_size)
- void* alloc(Allocator * allocator, size_t block_size) (выделение памяти при помощи аллокатора размера block_size)
- void* free(Allocator * allocator, void * block) (возвращает выделенную память аллокатору)

Задание варианта: Необходимо сравнить два алгоритма аллокации: списки свободных блоков (первое подходящее) и алгоритм Мак-Кьюзи-Кэрелса.

Общие сведения о программе

Программа состоит из двух папок include/allocator и src. В папке include/allocator находятся три заголовочных файла: IAllocator.h – интерфейсный класс Allocator, который является чисто виртуальным, и от которого будут отнаследованы два других класса Аллокаторов, каждый из которых использует опеределенный алгоритм аллокации; ListAllocator.h – класс аллокатора, который использует алгоритм списка свободных блоков (первое подходящее); MacKuseyCarelsAllocator.h – класс аллокатора, который использует алгоритм Мак-Кьюзи-Кэрелса. В папке src находятся две папки: allocator, в которой находятся два файла ListAllocator.cpp и MacKuseyCarelsAllocator.cpp с реализацией соответствующих классов, и папки benchmark, в которой находится файл benchmark.cpp для сравнения двух алгоритмов аллокации. Также в главной директории проекта находится файл run.cpp для запуска кода, Makefile и CMakeLists.txt для удобной сборки и запуска проекта.

Для реализации класса ListAllocator.h использовался алгоритм «Списки свободных блоков (первое подходящее)».

Алгоритм аллокации "Списки свободных блоков (первое подходящее)" используется для эффективной работы с динамической памятью. Основная идея этого алгоритма заключается в том, чтобы иметь список свободных блоков памяти и выбирать первый блок, который удовлетворяет требованиям размера запрашиваемой памяти.

1. Инициализация памяти: изначально, имеется единый большой блок памяти, который целиком доступен для распределения. Этот блок инициализируется как "свободный".

2. Структура данных: для хранения списка свободных блоков, часто используется связный список. Каждый элемент списка содержит информацию о размере блока и указатель на следующий свободный блок.

3. Аллокация памяти: при запросе аллокации памяти определенного размера, алгоритм просматривает список свободных блоков, начиная с самого начала списка. Если находится блок, размер которого достаточно большой, чтобы удовлетворить запрос, то этот блок используется для аллокации. Для этого блоку просто присваивается статус "занят", и происходит выдача указателя на его начало. Если блок недостаточно большой, алгоритм продолжает поиск следующего блока. Если список свободных блоков заканчивается и подходящий блок не найден, может быть выполнено управление остатком памяти.

4. Освобождение памяти: при освобождении блока памяти, он помечается как "свободный". Затем алгоритм проверяет соседние свободные блоки. Если они существуют, то происходит их объединение в один большой блок. Это позволяет

эффективно использовать фрагментированную память, устраняя разрывы между свободными блоками.

Преимущества алгоритма "Списки свободных блоков (первое подходящее)":

- Простота реализации и понимания.

- Эффективное использование памяти за счет объединения свободных блоков.

- Быстрый доступ к первому подходящему блоку.

Недостатки алгоритма "Списки свободных блоков (первое подходящее)":

- Возможна фрагментация памяти при неправильном распределении блоков.

- Поиск подходящего блока может занимать больше времени, если он находится в конце списка свободных блоков.

- Требуется дополнительное пространство для хранения указателей на следующий блок в списке свободных блоков.

В целом, алгоритм аллокации "Списки свободных блоков (первое подходящее)" является простым и эффективным способом управления памятью во время выполнения программы. Он широко применяется в различных системах и является одним из основных алгоритмов аллокации памяти.

Для реализации класса MacKuseyCarelsAllocator.h использовался алгоритм аллокации «Мак-Кьюзи-Кэрелса».

Алгоритм аллокации Мак-Кьюзи-Кэрелса (МСС) является одним из способов управления динамической памятью и используется в системах, где имеется ограниченное количество свободного пространства и требуется эффективное использование памяти.

1. Инициализация памяти: при запуске алгоритма имеется единый большой блок памяти, который целиком доступен для распределения. Этот блок инициализируется как "свободный".

2. Разбиение блоков: изначально все свободное пространство представлено в виде одного блока. При аллокации памяти блок может быть разделен на два более маленьких блока: один используется для запрошенного объема памяти, а второй остается свободным.

3. Структура данных: для хранения информации о блоках памяти используются двусвязные списки. Каждый блок содержит информацию о размере и состоянии (занят или свободен), а также указатели на предыдущий и следующий блоки.

4. Аллокация памяти: при запросе аллокации памяти определенного размера, алгоритм просматривает связанный список свободных блоков, начиная с самого начала списка. Если находится свободный блок, размер которого достаточно большой, чтобы удовлетворить запрос, то этот блок используется для аллокации. Для этого блок помечается как "занятый", а указатель на начало блока возвращается как результат аллокации. Если блок недостаточно большой, алгоритм продолжает поиск в списке свободных блоков. Если блоки заканчиваются и подходящий блок не найден, происходит обработка ошибки.

5. Освобождение памяти: при освобождении блока памяти, он помечается как "свободный". Затем алгоритм проверяет блоки-соседи. Если соседние блоки также являются свободными, происходит их объединение в один большой блок. Это позволяет эффективно использовать фрагментированную память, устраняя разрывы между свободными блоками.

Преимущества алгоритма Мак-Кьюзи-Кэрелса:

- Устранение фрагментации памяти путем объединения свободных блоков.

- Поддержка размещения памяти переменного размера.

- Более эффективное использование памяти по сравнению с другими алгоритмами.

Недостатки алгоритма Мак-Кьюзи-Кэрелса:

- Дополнительное пространство требуется для хранения информации о каждом блоке.

- Поиск подходящего блока может занимать больше времени, особенно если список свободных блоков длинный.

- Нет возможности выделения памяти, которая является частью свободного блока без разделения его на несколько блоков.

В целом, алгоритм аллокации Мак-Кьюзи-Кэрелса представляет собой эффективный способ управления динамической памятью, особенно в условиях ограниченного пространства. Он широко применяется в различных программных системах, где требуется эффективное использование памяти и управление динамическим выделением памяти.

Общий метод и алгоритм решения

Алгоритмы аллокации, а также суть их реализации описаны выше. Для сравнения двух алгоритмов аллокации мной был написан специальный бенчмарк. Сравнивал алгоритмы я по самому важному критерию – затраченному времени на аллокацию. Для этой цели использовалась библиотека chrono, с ее широким функционалом для замеров времени. Я делал очень большое количество запросов на выделение памяти и на освобождение и смотрел на поведение программы и затраченное время.

Код был написан в парадигме ООП с использованием наследования и полиморфизма.

Для разработки классов Аллокаторов использовал cppreference и статью allocatortraits. Очень помогли

Код программы

./include/allocator/IAllocator.h

```
#pragma once

#include <exception>
#include <iostream>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <stdbool.h>
#include <sys/mman.h>

namespace allocator {

class Allocator {
  public:
    using void_pointer = void*;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;
    using propagate_on_container_move_assignment = std::true_type;
    using is_always_equal = std::true_type;

  protected:
    Allocator() = default;

  public:
    virtual ~Allocator() = default;

    virtual void_pointer alloc(const size_type) = 0;
    virtual void free(void_pointer) = 0;
};

}; // namespace allocator
```

./include/allocator/ListAllocator.h

```
#pragma once

#include "IAllocator.h"

namespace allocator {
```

```
struct BlockHeader {
    size_t _size;
    BlockHeader* _next;
};

class ListAllocator final : public Allocator {
  private:
    BlockHeader* _free_blocks_list;

  public:
    ListAllocator() = delete;
    ListAllocator(void_pointer, size_type);

    virtual ~ListAllocator();

    virtual void_pointer alloc(size_type) override;
    virtual void free(void_pointer) override;
};

}; // namespace allocator
```

./include/allocator/ MacKuseyCarelsAllocator.h

```
#pragma once

#include "IAllocator.h"

namespace allocator {

struct Page {
    Page* _next;
    bool _is_large;
    size_t _block_size;
};

class MacKuseyCarelsAllocator final : public Allocator {
  private:
    void* _memory;
    Page* _free_pages_list;
    size_t _memory_size;
    size_t _page_size;
```

```
  public:
    MacKuseyCarelsAllocator() = delete;
    MacKuseyCarelsAllocator(void_pointer, size_type);

    virtual ~MacKuseyCarelsAllocator();

    virtual void_pointer alloc(size_type) override;
    virtual void free(void_pointer) override;
};

}; // namespace allocator
```

./src/allocator/ListAllocator.cpp

```
#include "../../include/allocator/ListAllocator.h"

using namespace allocator;

ListAllocator::ListAllocator(void_pointer real_memory, size_type
memory_size)
{
    _free_blocks_list = reinterpret_cast<BlockHeader*>(real_memory +
sizeof(ListAllocator));
    _free_blocks_list->_size = memory_size - sizeof(ListAllocator) -
sizeof(BlockHeader);
    _free_blocks_list->_next = nullptr;
}

ListAllocator::~ListAllocator()
{
    BlockHeader* cur_block = this->_free_blocks_list;

    while (cur_block) {
        BlockHeader* to_delete = cur_block;
        cur_block = cur_block->_next;
        to_delete = nullptr;
    }

    this->_free_blocks_list = nullptr;
}
```

```
typename Allocator::void_pointer ListAllocator::alloc(size_type
new_block_size)
{
    BlockHeader* prev_block = nullptr;
    BlockHeader* cur_block = this->_free_blocks_list;

    size_type adjusted_size = new_block_size + sizeof(BlockHeader);

    while (cur_block) {
        if (cur_block->_size >= adjusted_size) {
            if (cur_block->_size >= adjusted_size + sizeof(Block-
Header)) {
                BlockHeader* new_block = reinterpret_cast<Block-
Header*>(reinterpret_cast<int8_t*>(cur_block) + adjusted_size);

                new_block->_size = cur_block->_size - adjusted_size -
sizeof(BlockHeader);
                new_block->_next = cur_block->_next;
                cur_block->_next = new_block;
                cur_block->_size = adjusted_size;
            }

            if (prev_block) {
                prev_block->_next = cur_block->_next;
            } else {
                this->_free_blocks_list = cur_block->_next;
            }

            return reinterpret_cast<int8_t*>(cur_block) +
sizeof(BlockHeader);
        }

        prev_block = cur_block;
        cur_block = cur_block->_next;
    }

    return nullptr;
}

void ListAllocator::free(void_pointer block)
{
    if (block == nullptr) return;
```

```
    BlockHeader* header = reinterpret_cast<Block-
Header*>(static_cast<int8_t*>(block) - sizeof(BlockHeader));
    header->_next = this->_free_blocks_list;
    this->_free_blocks_list = header;
}
```

./srcallocator/ MacKuseyCarelsAllocator.cpp

```
#include "../../include/allocator/MacKuseyCarelsAllocator.h"

using namespace allocator;

MacKuseyCarelsAllocator::MacKuseyCarelsAllocator(void_pointer
real_memory, size_type memory_size)
{
    _memory = reinterpret_cast<void*>(reinter-
pret_cast<int8_t*>(real_memory) + sizeof(MacKuseyCarelsAllocator));
    _free_pages_list = nullptr;
    _memory_size = memory_size - sizeof(MacKuseyCarelsAllocator);
    _page_size = getpagesize();
}

MacKuseyCarelsAllocator::~MacKuseyCarelsAllocator()
{
    Page* cur_page = this->_free_pages_list;

    while (cur_page) {
        Page* to_delete = cur_page;
        cur_page = cur_page->_next;
        munmap(to_delete, _page_size);
        to_delete = nullptr;
    }
    _free_pages_list = nullptr;
}

typename Allocator::void_pointer MacKuseyCarelsAllocator::al-
loc(size_type new_block_size)
{
    if (_memory_size < new_block_size) return nullptr;

    size_t rounded_block_size = 1;
```

```
    while (rounded_block_size < new_block_size) {
        rounded_block_size *= 2;
    }

    Page* prev_page = nullptr;
    Page* cur_page = _free_pages_list;

    while (cur_page) {
        if (!cur_page->_is_large && cur_page->_block_size ==
rounded_block_size) {
            void_pointer block = reinter-
pret_cast<void_pointer>(cur_page);
            _free_pages_list = cur_page->_next;
            _memory_size -= new_block_size;

            return block;
        }

        prev_page = cur_page;
        cur_page = cur_page->_next;
    }

    if (_memory_size < _page_size) return nullptr;

    Page* new_page = reinterpret_cast<Page*>(mmap(NULL, _page_size,
                                                  PROT_READ |
PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS,
                                                  -1, 0));

    if (new_page == MAP_FAILED) {
        throw std::bad_alloc();
    }
    new_page->_is_large = false;
    new_page->_block_size = rounded_block_size;
    new_page->_next = nullptr;

    size_t num_blocks = _page_size / rounded_block_size;
    for (size_t i = 0; i != num_blocks; ++i) {
        Page* block_page = reinterpret_cast<Page*>(reinter-
pret_cast<int8_t*>(new_page) + i * rounded_block_size);
        block_page->_is_large = false;
        block_page->_block_size = rounded_block_size;
```

```
        block_page->_next = this->_free_pages_list;
        this->_free_pages_list = block_page;
    }

    void_pointer block = reinterpret_cast<void_pointer>(new_page);
    this->_free_pages_list = new_page->_next;

    return block;
}

void MacKuseyCarelsAllocator::free(void_pointer block)
{
    if (block == nullptr) return;

    Page* page = reinterpret_cast<Page*>(block);
    page->_next = _free_pages_list;
    _free_pages_list = page;
}
```

./src/benchmark/benchmark.cpp

```
#include <chrono>
#include <cstdlib>
#include <vector>

#include "../include/allocator/ListAllocator.h"
#include "../include/allocator/MacKuseyCarelsAllocator.h"

using namespace allocator;

size_t page_size = sysconf(_SC_PAGESIZE);

void benchmark() {
    void* list_memory = sbrk(10000 * page_size);
    void* MKC_memory = mmap(NULL, 1000 * page_size, PROT_READ |
PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    ListAllocator list_alloc(list_memory, 10000 * page_size);
    MacKuseyCarelsAllocator MKC_alloc(MKC_memory, 1000 * page_size);
    std::vector<void*> list_blocks;
    std::vector<void*> MKC_blocks;
```

```
    std::cout << "Comparing ListAllocator and MacKuseyCarelsAllocator"
<< std::endl;

    std::cout << "Block allocation rate" << std::endl;
    auto start_time = std::chrono::steady_clock::now();
    for (size_t i = 0; i != 100000; ++i) {
        void* block = list_alloc.alloc(i % 50 + 10);
        list_blocks.push_back(block);
    }
    auto end_time = std::chrono::steady_clock::now();
    std::cout << "Time of alloc ListAllocator: " <<
                 std::chrono::duration_cast<std::chrono::millisec-
onds>(end_time - start_time).count() <<
                 " milliseconds" << std::endl;

    start_time = std::chrono::steady_clock::now();
    for (size_t i = 0; i != 100000; ++i) {
        void* block = MKC_alloc.alloc(i % 50 + 10);
        MKC_blocks.push_back(block);
    }
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of alloc MacKuseyCarelsAllocator: " <<
                 std::chrono::duration_cast<std::chrono::millisec-
onds>(end_time - start_time).count() <<
                 " milliseconds" << std::endl;

    std::cout << "Block free rate" << std::endl;
    start_time = std::chrono::steady_clock::now();
    for (size_t i = 0; i != list_blocks.size(); ++i) {
        list_alloc.free(list_blocks[i]);
        if (i < 20) {
            std::cout << list_blocks[i] << std::endl;
        }
    }
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of free ListAllocator: " <<
                 std::chrono::duration_cast<std::chrono::millisec-
onds>(end_time - start_time).count() <<
                 " milliseconds" << std::endl;

    start_time = std::chrono::steady_clock::now();
```

```
    for (size_t i = 0; i != MKC_blocks.size(); ++i) {
        MKC_alloc.free(MKC_blocks[i]);
        if (i < 20) {
            std::cout << MKC_blocks[i] << std::endl;
        }
    }
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of free MacKuseyCarelsAllocator: " <<
                 std::chrono::duration_cast<std::chrono::millisec-
onds>(end_time - start_time).count() <<
                 " milliseconds" << std::endl;
}
```

./run.cpp

```
#include "include/allocator/ListAllocator.h"
#include "include/allocator/MacKuseyCarelsAllocator.h"
#include "src/benchmarks/benchmark.cpp"

using namespace allocator;

int main() {
    size_t page_size = sysconf(_SC_PAGESIZE);
    void* list_memory = sbrk(10 * page_size);

    ListAllocator list_alloc(list_memory, 10 * page_size);

    void* MKC_memory = mmap(NULL, 10 * page_size, PROT_READ |
PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    MacKuseyCarelsAllocator MKC_alloc(MKC_memory, 10 * page_size);

    void* block1 = list_alloc.alloc(10000);
    void* block2 = list_alloc.alloc(10);
    void* block3 = list_alloc.alloc(44651047871);

    void* block4 = MKC_alloc.alloc(1024);
    void* block5 = MKC_alloc.alloc(2049);
    void* block6 = MKC_alloc.alloc(144420166);

    printf("Block 1: %p\n", block1);
    printf("Block 2: %p\n", block2);
```

```
    printf("Block 3: %p\n", block3);

    printf("Block 4: %p\n", block4);
    printf("Block 5: %p\n", block5);
    printf("Block 6: %p\n", block6);

    benchmark();

    return 0;
}
```

./CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(KP)

add_compile_options(-Wall -g -O0)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

set(INCLUDE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/include)
set(SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/src)

include_directories(${INCLUDE_DIR})

file(GLOB_RECURSE SOURCES LIST_DIRECTORIES false ${SOURCE_DIR}/*.cpp)
file(GLOB_RECURSE INCLUDES LIST_DIRECTORIES false ${IN-
CLUDE_DIR}/*.hpp)

add_library(
            ${CMAKE_PROJECT_NAME}_lib
            ${SOURCES}
            ${INCLUDES}
            ${ENUMS}
           )

add_executable(
               ${CMAKE_PROJECT_NAME}_exe
               ${CMAKE_CURRENT_SOURCE_DIR}/run.cpp
              )
```

```
set_target_properties(
                      ${PROJECT_NAME}_exe PROPERTIES
                      CXX_STANDARD 20
                      CXX_STANDARD_REQUIRED YES
                      CXX_EXTENSIONS NO
                     )

add_dependencies(${CMAKE_PROJECT_NAME}_exe ${CMAKE_PROJECT_NAME}_lib)
target_link_libraries(${CMAKE_PROJECT_NAME}_exe ${CMAKE_PRO-
JECT_NAME}_lib)
```

./Makefile

```
.PHONY: build run clean strace

build: clean
    mkdir build
    cd ./build; cmake ..; make all

run:
    ./build/*_exe

clean:
    rm -rf ./build/

strace:
    strace -f ./build/*_exe
```

Использование утилиты strace

Без бенчмарка, потому что слишком много системных вызовов на выделение и очищение памяти

execve("./build/KP_exe", ["./build/KP_exe"], 0x7ffc96f74f28 /* 60 vars */) = 0

brk(NULL) = 0x5622f08d3000

arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd746cef80) = -1 EINVAL (Invalid argument)

mmap(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7c4b098000

access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)

openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=67103, ...}, AT_EMPTY_PATH) = 0

mmap(NULL, 67103, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f7c4b087000

close(3) = 0

openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libstdc++.so.6",
O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832

newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=2260296, ...}, AT_EMPTY_PATH)
= 0

mmap(NULL, 2275520, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f7c4ae00000

mprotect(0x7f7c4ae9a000, 1576960, PROT_NONE) = 0

mmap(0x7f7c4ae9a000, 1118208, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x9a000) = 0x7f7c4ae9a000

mmap(0x7f7c4afab000, 454656, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ab000) = 0x7f7c4afab000

mmap(0x7f7c4b01b000, 57344, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x21a000) = 0x7f7c4b01b000

mmap(0x7f7c4b029000, 10432, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f7c4b029000

close(3) = 0

openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libgcc_s.so.1", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0"..., 832) = 832

newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=125488, ...}, AT_EMPTY_PATH) = 0

mmap(NULL, 127720, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7c4b067000

mmap(0x7f7c4b06a000, 94208, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x3000) = 0x7f7c4b06a000

mmap(0x7f7c4b081000, 16384, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1a000) = 0x7f7c4b081000

mmap(0x7f7c4b085000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1d000) = 0x7f7c4b085000

close(3) = 0

openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"..., 832) = 832

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0"..., 48, 848) = 48

pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0 =\340\2563\265?\356\25x\261\27\313A#\350"..., 68, 896) = 68

newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2216304, ...}, AT_EMPTY_PATH) = 0

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

mmap(NULL, 2260560, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7c4aa00000

mmap(0x7f7c4aa28000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f7c4aa28000

mmap(0x7f7c4abbd000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7f7c4abbd000

mmap(0x7f7c4ac15000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x214000) = 0x7f7c4ac15000

mmap(0x7f7c4ac1b000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f7c4ac1b000

close(3) = 0

openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libm.so.6", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832

newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=940560, ...}, AT_EMPTY_PATH) = 0

mmap(NULL, 942344, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7c4ad19000

mmap(0x7f7c4ad27000, 507904, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xe000) = 0x7f7c4ad27000

mmap(0x7f7c4ada3000, 372736, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x8a000) = 0x7f7c4ada3000

mmap(0x7f7c4adfe000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xe4000) = 0x7f7c4adfe000

close(3) = 0

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7c4b065000

arch_prctl(ARCH_SET_FS, 0x7f7c4b0663c0) = 0

set_tid_address(0x7f7c4b066690) = 5489

set_robust_list(0x7f7c4b0666a0, 24) = 0

rseq(0x7f7c4b066d60, 0x20, 0, 0x53053053) = 0

mprotect(0x7f7c4ac15000, 16384, PROT_READ) = 0

mprotect(0x7f7c4adfe000, 4096, PROT_READ) = 0

mprotect(0x7f7c4b085000, 4096, PROT_READ) = 0

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7c4b063000

```
mprotect(0x7f7c4b01b000, 45056, PROT_READ) = 0

mprotect(0x5622eeef8000, 4096, PROT_READ) = 0

mprotect(0x7f7c4b0d2000, 8192, PROT_READ) = 0

prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0

munmap(0x7f7c4b087000, 67103)           = 0

getrandom("\xb9\xae\x29\x78\xef\x33\xb2\x70", 8, GRND_NONBLOCK) = 8

brk(NULL)                               = 0x5622f08d3000

brk(0x5622f08f4000)                     = 0x5622f08f4000

futex(0x7f7c4b02977c, FUTEX_WAKE_PRIVATE, 2147483647) = 0

brk(0x5622f08fe000)                     = 0x5622f08fe000

mmap(NULL, 40960, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7c4b08e000

mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7c4b0d1000

mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7c4b08d000

newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...},
AT_EMPTY_PATH) = 0

write(1, "Block 1: 0x5622f08f4020\n", 24Block 1: 0x5622f08f4020

) = 24

write(1, "Block 2: 0x5622f08f6740\n", 24Block 2: 0x5622f08f6740

) = 24

write(1, "Block 3: (nil)\n", 15Block 3: (nil)

)       = 15

write(1, "Block 4: 0x7f7c4b0d1000\n", 24Block 4: 0x7f7c4b0d1000

) = 24

write(1, "Block 5: 0x7f7c4b08d000\n", 24Block 5: 0x7f7c4b08d000

) = 24
```

```
write(1, "Block 6: (nil)\n", 15Block 6: (nil)
)       = 15
exit_group(0)                   = ?
+++ exited with 0 +++
```

Демонстрация работы программы

```
baronpipistron@BaronPIpistron:~/MAI_OS/KP$ make run
./build/*_exe
Block 1: 0x55b6ed9d0020
Block 2: 0x55b6ed9d2740
Block 3: (nil)
Block 4: 0x7f8d5db8b000
Block 5: 0x7f8d5db47000
Block 6: (nil)
Comparing ListAllocator and MacKuseyCarelsAllocator
Block allocation rate
Time of alloc ListAllocator: 6 milliseconds
Time of alloc MacKuseyCarelsAllocator: 477 milliseconds
Block free rate
0x55b6ed9da020
0x55b6ed9da03a
0x55b6ed9da055
0x55b6ed9da071
0x55b6ed9da08e
0x55b6ed9da0ac
0x55b6ed9da0cb
0x55b6ed9da0eb
0x55b6ed9da10c
0x55b6ed9da12e
0x55b6ed9da151
0x55b6ed9da175
0x55b6ed9da19a
0x55b6ed9da1c0
0x55b6ed9da1e7
```

0x55b6ed9da20f

0x55b6ed9da238

0x55b6ed9da262

0x55b6ed9da28d

0x55b6ed9da2b9

Time of free ListAllocator: 0 milliseconds

0x7f8d5db46000

0x7f8d5db45000

0x7f8d5db44000

0x7f8d5db43000

0x7f8d5db42000

0x7f8d5db41000

0x7f8d5da35000

0x7f8d5da34000

0x7f8d5da33000

0x7f8d5da32000

0x7f8d5da31000

0x7f8d5da30000

0x7f8d5da2f000

0x7f8d5da2e000

0x7f8d5da2d000

0x7f8d5da2c000

0x7f8d5d7ff000

0x7f8d5d7fe000

0x7f8d5d7fd000

0x7f8d5d7fc000

Time of free MacKuseyCarelsAllocator: 2 milliseconds

Вывод

В ходе выполнения данной работы, я сравнил два алгоритма аллокации «Списки свободных блоков (первое подходящее)» и «Мак-Кьюзи-Кэрелса». Работа оказалась очень интересной и выполнять ее было одно удовольствие. Пришлось погрузиться в глубины ОС Linux чтобы правильно написать оба алгоритма и понять как сама ОС взаимодействует с памятью на уровне ядра. Узнал очень много нового про саму ОС и аллокаторы, а также усовершенствовал свои навыки во владении C++.

Результат сравнения двух алгоритмов аллокации оказался довольно интересный. «Списки свободных блоков (первое подходящее)» оказался в разы быстрее «Мак-Кьюзи-Кэрелса». Я считаю, что это связано с тем, что второй алгоритм стучится в ядро, если не находит страницы с подходящим размером блоков, а вызовы в ядро очень долгие. Но с другой стороны «Мак-Кьюзи-Кэрелса» предлагает нам более надежное хранение наших данных и исключает фрагментацию, поиск свободных блоков не занимает слишком много времени, и в целом нахожу его более универсальным, в то время как поиск в первом алгоритме будет увеличиваться пропорционально тому, чем дальше в списке находится подходящий блок. Освобождают память оба алгоритма одинаково хорошо.

В целом работой остался доволен, выполнять было крайне интересно. Но буду пользоваться скорее всего все равно моим любимым стандартным std::allocator<T> и не знать никаких проблем. Несомненно, есть шанс, что понадобится написать свой аллокатор, но он крайне мал.

Работа на 8 из 10 – долго мучался со вторым алгоритмом.