Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу

«Операционные системы»

Тема

«Сравнение алгоритмов аллокации памяти»

Студент: Эсмедляев Федор Романович
Группа: М8О–212Б–22
Вариант: 22
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2023.

Постановка задачи

Задание

1. Приобретение практических навыков в использовании знаний, полученных в течении курса
2. Проведение исследования в выбранной предметной области

b. Возможные варианты курсового проекта:

b.i. 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

b.ii. Виртуальная память

Необходимо создать рабочую модель виртуальной памяти. Требуется взять алгоритм аллокации памяти и использовать его совместно с алгоритмом замещения страниц. При разработке предусмотреть следующий интерфейс использования:

- CreateMemory
- Malloc
- Free

Параметры функций могут отличаться в зависимости от выбранных алгоритмов. Исследовать полученную модель виртуальной памяти – ее плюсы и минусы. Возможно сделать данные алгоритмы, переопределив стандартные механизмы аллокации языка C++. Также проработать модель тестирования полученного программного решения.

Общие сведения о программе

Основные файлы — файлы аллокации, представлены в виде .h и .cpp. В них реализованы 2 алгоритма аллокации списки свободных блоков (первое подходящее) и алгоритм Мак-Кьюзи-Кэрелса. Все тесты происходят при совместном использовании run.cpp и benchmark.cpp.

Алгоритм решения

Списки свободных блоков(первое подходящее):

1)Выделение

а)Находим нужный блок памяти

б)Уменьшаем его размер на N

в)Если есть свободный блок длинной N+1 нельзя выделить из него блок длиной N

2)Освобождение

а)Находим место для вставки освобожденного блока

б)При необходимости производим слияние соседних блоков.

в)С односвязным списком сложность O(n)

Мак-Кьюзи-Кэрелся:

1)Выделение

а)При запросе памяти округляем запрос до степени 2

б)При выделении блока в блоке не требуется хранить дополнительную информацию

в)Если нет блока нужного размера, то выделяем новую страницу, которую разбиваем на блоки данного размера

2)Освобождение

а)Для освобождения не требуется указывать размер блока

Основные файлы программы

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>

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;
};
```

ListAllocator.h

```
#pragma once

#include "IAllocator.h"

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;
};
```

MacKuseyCaretsAllocator.h

```
#pragma once

#include "IAllocator.h"

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;
};
```

ListAllocator.cpp

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

```
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(BlockHeader)) {
                BlockHeader* new_block =
reinterpret_cast<BlockHeader*>(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<BlockHeader*>(static_cast<int8_t*>(blo
ck) - sizeof(BlockHeader));
    header->_next = this->_free_blocks_list;
    this->_free_blocks_list = header;
}
```

MacKuseyCarestAllocator.cpp

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

MacKuseyCarelsAllocator::MacKuseyCarelsAllocator(void_p
ointer real_memory, size_type memory_size)
{
    _memory =
reinterpret_cast<void*>(reinterpret_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::alloc(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 =
reinterpret_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*>(reinterpret_cast<int8_t*>(new_p
age) + 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;
}
```

benchmark.cpp

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

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

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::milliseconds>(e
nd_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::milliseconds>(e
nd_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::milliseconds>(e
nd_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::milliseconds>(e
nd_time - start_time).count() <<
```

```
                    " milliseconds" << std::endl;
}
```

Тесты

```
void* block1 = list_alloc.alloc(10000);
void* block2 = list_alloc.alloc(10);
void* block3 = list_alloc.alloc(1);

void* block4 = MKC_alloc.alloc(10000);
void* block5 = MKC_alloc.alloc(10);
void* block6 = MKC_alloc.alloc(1);
```

вывод

```
Block 1: 0x5647a2c5e020
Block 2: 0x5647a2c60740
Block 3: 0x5647a2c6075a
Block 4: 0x7f459fc04000
Block 5: 0x7f459fbc0000
Block 6: 0x7f459fbbf000
Comparing ListAllocator and MacKuseyCarelsAllocator
Block allocation rate
Time of alloc ListAllocator: 10 milliseconds
Time of alloc MacKuseyCarelsAllocator: 523 milliseconds
Block free rate
0x5647a2c68020
0x5647a2c6803a
0x5647a2c68055
0x5647a2c68071
0x5647a2c6808e
0x5647a2c680ac
0x5647a2c680cb
0x5647a2c680eb
0x5647a2c6810c
```

0x5647a2c6812e

0x5647a2c68151

0x5647a2c68175

0x5647a2c6819a

0x5647a2c681c0

0x5647a2c681e7

0x5647a2c6820f

0x5647a2c68238

0x5647a2c68262

0x5647a2c6828d

0x5647a2c682b9

Time of free ListAllocator: 1 milliseconds

0x7f459fbbe000

0x7f459fbbd000

0x7f459fbbc000

0x7f459fbbb000

0x7f459fbba000

0x7f459faad000

0x7f459faac000

0x7f459faab000

0x7f459faaa000

0x7f459faa9000

0x7f459faa8000

0x7f459faa7000

0x7f459faa6000

0x7f459faa5000

0x7f459faa4000

0x7f459faa3000

0x7f459faa2000

0x7f459faa1000

0x7f459faa0000

0x7f459fa9f000

Time of free MacKuseyCarelsAllocator: 2 milliseconds

```
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);
```

Вывод

```
Block 1: 0x55d7b0979020
Block 2: 0x55d7b097b740
Block 3: (nil)
Block 4: 0x7f567b74e000
Block 5: 0x7f567b70a000
Block 6: (nil)
Comparing ListAllocator and MacKuseyCarelsAllocator
Block allocation rate
Time of alloc ListAllocator: 12 milliseconds
Time of alloc MacKuseyCarelsAllocator: 515 milliseconds
```

Вывод

Проделав работу, я стало понятно, что алгоритм на списке быстрее работает, если количество блоков небольшое, поскольку дерево поддерживать и проходить по элементам гараздо дольше, однако на большом количество аллокаций, время алгоритмов почти одинаковое.