Московский авиационный институт
(Национальный исследовательский университет)
Факультет "Информационные технологии и прикладная математика"
Кафедра "Вычислительная математика и программирование"

Курсовой проект по курсу “Операционные системы”

Тема работы “Аллокаторы памяти ”

Студент: Сибирцев Роман Денисович

Группа: М8О-208Б-22

Преподаватель: Миронов Евгений Сергеевич

Вариант: 20

Оценка: _____

Дата: _____

Подпись: _____

Москва, 2023

Содержание

1 Репозиторий

https://github.com/RomanSibirtsev/MAI_OS_labs

2 Цель работы

- Приобретение практических навыков в использовании знаний, полученных в течении курса
- Проведение исследования в выбранной предметной области

3 Задание

Необходимо сравнить два алгоритма аллокации: списки свободных блоков (наиболее подходящее) и алгоритм Мак-Кьюзи-Кэрелса

4 Описание работы программы

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

Алгоритм Мак-Кьюзика-Кэрелса:

Алгоритм подразумевает, что память разбита на набор последовательных страниц, и все буферы, относящиеся к одной странице, должны иметь одинаковый размер (являющийся некоторой степенью числа 2). Каждая страница может находиться в одном из трёх перечисленных состояний.

- Быть свободной.
- Быть разбитой на буферы определённого размера.
- Являться частью буфера, объединяющего сразу несколько страниц.

Список свободных блоков:

Свободные блоки организуем в список. Блок хранит размер и ссылку на следующий свободный блок

Наиболее подходящий участок. Выделение памяти из наиболее подходящей свободной области, имеющей достаточный для удовлетворения запроса объём. Это самый выгодный по памяти алгоритм из всех трёх (первый подходящий участок, наиболее походящий участок, наименее подходящий участок), но он не самый быстрый.

5 Исходный код

MCKAllocator.hpp

```
#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>

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;


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

class MCKAllocator final{
    private:
    void* _memory;
    Page* _free_pages_list;
    size_t _memory_size;
    size_t _page_size;

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

    virtual ~MCKAllocator();

    void_pointer alloc(size_type);
    void free(void_pointer);
};
```

MCKAllocator.cpp

```
    #include "MCKAllocator.hpp"

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

    MCKAllocator::~MCKAllocator()
    {
```

```
13          Page* cur_page = this->_free_pages_list;
14
15          while (cur_page) {
16              Page* to_delete = cur_page;
17              cur_page = cur_page->_next;
18              munmap(to_delete, _page_size);
19              to_delete = nullptr;
20          }
21          _free_pages_list = nullptr;
22      }
23
24      void_pointer MCKAllocator::alloc(size_type new_block_size)
25      {
26          if (_memory_size < new_block_size) return nullptr;
27
28          size_t rounded_block_size = 1;
29          while (rounded_block_size < new_block_size) {
30              rounded_block_size *= 2;
31          }
32
33          Page* prev_page = nullptr;
34          Page* cur_page = _free_pages_list;
35
36          while (cur_page) {
37              if (!cur_page->_is_large && cur_page->_block_size ==
    rounded_block_size) {
38                  void_pointer block = reinterpret_cast<void_pointer
    >(cur_page);
39                  _free_pages_list = cur_page->_next;
40                  _memory_size -= new_block_size;
41
42                  return block;
43              }
44
45              prev_page = cur_page;
46              cur_page = cur_page->_next;
47          }
48
49          if (_memory_size < _page_size) return nullptr;
50
51          Page* new_page = reinterpret_cast<Page*>(mmap(NULL,
    _page_size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS
    , -1, 0));
52
53          if (new_page == MAP_FAILED) {
54              throw std::bad_alloc();
55          }
56          new_page->_is_large = false;
57          new_page->_block_size = rounded_block_size;
58          new_page->_next = nullptr;
59
60          size_t num_blocks = _page_size / rounded_block_size;
61          for (size_t i = 0; i != num_blocks; ++i) {
62              Page* block_page = reinterpret_cast<Page*>(
    reinterpret_cast<int8_t*>(new_page) + i * rounded_block_size);
63              block_page->_is_large = false;
64              block_page->_block_size = rounded_block_size;
65              block_page->_next = this->_free_pages_list;
66              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 MCKAllocator::free(void_pointer block)
    {
        if (block == nullptr) return;

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

FBLAllocator.hpp

```
    #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>

    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;

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

    class FBLAllocator final{
      private:
        BlockHeader* _free_blocks_list;

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

        virtual ~FBLAllocator();

        void_pointer alloc(size_type);
        void free(void_pointer);
    };
```

FBLAllocator.cpp

```
    #include "FBLAllocator.hpp"

```

```
3     FBLAllocator::FBLAllocator(void_pointer real_memory, size_type
      memory_size)
4     {
5         _free_blocks_list = reinterpret_cast<BlockHeader*>(
      real_memory + sizeof(FBLAllocator));
6         _free_blocks_list->_size = memory_size - sizeof(
      FBLAllocator) - sizeof(BlockHeader);
7         _free_blocks_list->_next = nullptr;
8     }
9
10    FBLAllocator::~FBLAllocator()
11    {
12        BlockHeader* cur_block = this->_free_blocks_list;
13
14        while (cur_block) {
15            BlockHeader* to_delete = cur_block;
16            cur_block = cur_block->_next;
17            to_delete = nullptr;
18        }
19
20        this->_free_blocks_list = nullptr;
21    }
22
23    void_pointer FBLAllocator::alloc(size_type new_block_size)
24    {
25        BlockHeader* prev_block = nullptr;
26        BlockHeader* cur_block = this->_free_blocks_list;
27
28        size_type adjusted_size = new_block_size + sizeof(
      BlockHeader);
29
30        int diff = new_block_size * 100;
31
32        while (cur_block) {
33            if (cur_block->_size >= adjusted_size && cur_block->
      _size - adjusted_size < diff) {
34                diff = cur_block->_size;
35            }
36            prev_block = cur_block;
37            cur_block = cur_block->_next;
38        }
39
40        prev_block = nullptr;
41        cur_block = this->_free_blocks_list;
42
43        while (cur_block) {
44            if (cur_block->_size >= adjusted_size) {
45                if (cur_block->_size >= adjusted_size + sizeof(
      BlockHeader)) {
46                    BlockHeader* new_block = reinterpret_cast<
      BlockHeader*>(reinterpret_cast<int8_t*>(cur_block) +
      adjusted_size);
47
48                    new_block->_size = cur_block->_size -
      adjusted_size - sizeof(BlockHeader);
49                    new_block->_next = cur_block->_next;
50                    cur_block->_next = new_block;
51                    cur_block->_size = adjusted_size;
52                }
```

```
53
54                  if (prev_block) {
55                      prev_block->_next = cur_block->_next;
56                  } else {
57                      this->_free_blocks_list = cur_block->_next;
58                  }
59
60                  return reinterpret_cast<int8_t*>(cur_block) +
    sizeof(BlockHeader);
61              }
62
63              prev_block = cur_block;
64              cur_block = cur_block->_next;
65          }
66
67          return nullptr;
68      }
69
70      void FBLAllocator::free(void_pointer block)
71      {
72          if (block == nullptr) return;
73
74          BlockHeader* header = reinterpret_cast<BlockHeader*>(
    static_cast<int8_t*>(block) - sizeof(BlockHeader));
75          header->_next = this->_free_blocks_list;
76          this->_free_blocks_list = header;
77      }
```

main.cpp

```
1   #include <chrono>
2   #include <cstdlib>
3   #include <vector>
4
5   #include "MCKAllocator.hpp"
6   #include "FBLAllocator.hpp"
7
8   size_t page_size = sysconf(_SC_PAGESIZE);
9
10  int main() {
11
12      void* list_memory = sbrk(10000 * page_size * 100);
13      void* MKC_memory = sbrk(10000 * page_size * 100);
14
15      FBLAllocator list_alloc(list_memory, 10000 * page_size);
16      MCKAllocator MKC_alloc(MKC_memory, 1000 * page_size);
17      std::vector<void*> list_blocks;
18      std::vector<void*> MKC_blocks;
19
20      std::cout << "Comparing FBLAllocator and MCKAllocator" <<
    std::endl;
21
22      std::cout << "Block allocation rate" << std::endl;
23      auto start_time = std::chrono::steady_clock::now();
24      for (size_t i = 0; i != 100000; ++i) {
25          void* block = list_alloc.alloc(i % 1000 + 100);
26          list_blocks.push_back(block);
27      }
28      auto end_time = std::chrono::steady_clock::now();
29      std::cout << "Time of alloc FBLAllocator: " <<
```

```
                    std::chrono::duration_cast<std::chrono::milliseconds>(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 % 1000 + 100);
        MKC_blocks.push_back(block);
    }
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of alloc MCKAllocator: " <<
                    std::chrono::duration_cast<std::chrono::milliseconds>(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]);
    }
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of free FBLAllocator: " <<
                    std::chrono::duration_cast<std::chrono::milliseconds>(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]);
    }
    end_time = std::chrono::steady_clock::now();

    std::cout << "Time of free MCKAllocator: " <<
                    std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time).count() <<
                    " milliseconds" << std::endl;


 //--------

    std::vector<void*> list_blocks2;
    std::vector<void*> MKC_blocks2;

    list_memory = sbrk(10000 * page_size * 100);
    MKC_memory = sbrk(10000 * page_size * 100);

    FBLAllocator list_alloc2(list_memory, 10000 * page_size);
    MCKAllocator MKC_alloc2(MKC_memory, 1000 * page_size);
    std::cout << "Block allocation rate of 1000 bytes" << std::endl;
    start_time = std::chrono::steady_clock::now();
    for (size_t i = 0; i != 100000; ++i) {
        void* block = list_alloc2.alloc(1000);
        list_blocks2.push_back(block);
    }
    end_time = std::chrono::steady_clock::now();
    std::cout << "Time of alloc FBLAllocator: " <<
                    std::chrono::duration_cast<std::chrono::milliseconds>(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_alloc2.alloc(1000);
            MKC_blocks2.push_back(block);
        }
        end_time = std::chrono::steady_clock::now();
        std::cout << "Time of alloc MCKAllocator: " <<
                    std::chrono::duration_cast<std::chrono::
milliseconds>(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_alloc2.free(list_blocks2[i]);
        }
        end_time = std::chrono::steady_clock::now();
        std::cout << "Time of free FBLAllocator: " <<
                    std::chrono::duration_cast<std::chrono::
milliseconds>(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_alloc2.free(MKC_blocks2[i]);
        }
        end_time = std::chrono::steady_clock::now();

        std::cout << "Time of free MCKAllocator: " <<
                    std::chrono::duration_cast<std::chrono::
milliseconds>(end_time - start_time).count() <<
                    " milliseconds" << std::endl;

     //----------------------------

        std::vector<void*> list_blocks3;
        std::vector<void*> MKC_blocks3;

        list_memory = sbrk(10000 * page_size * 100);
        MKC_memory = sbrk(10000 * page_size * 100);

        FBLAllocator list_alloc3(list_memory, 10000 * page_size);
        MCKAllocator MKC_alloc3(MKC_memory, 1000 * page_size);
        std::cout << "Block allocation rate of 3000 bytes" << std
::endl;
        start_time = std::chrono::steady_clock::now();
        for (size_t i = 0; i != 100000; ++i) {
            void* block = list_alloc3.alloc(3000);
            list_blocks3.push_back(block);
        }
        end_time = std::chrono::steady_clock::now();
        std::cout << "Time of alloc FBLAllocator: " <<
                    std::chrono::duration_cast<std::chrono::
milliseconds>(end_time - start_time).count() <<
                    " milliseconds" << std::endl;

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

```
137         for (size_t i = 0; i != 100000; ++i) {
138             void* block = MKC_alloc3.alloc(3000);
139             MKC_blocks3.push_back(block);
140         }
141         end_time = std::chrono::steady_clock::now();
142         std::cout << "Time of alloc MCKAllocator: " <<
143                         std::chrono::duration_cast<std::chrono::
    milliseconds>(end_time - start_time).count() <<
144                         " milliseconds" << std::endl;
145 
146         std::cout << "Block free rate" << std::endl;
147         start_time = std::chrono::steady_clock::now();
148         for (size_t i = 0; i != list_blocks.size(); ++i) {
149             list_alloc3.free(list_blocks3[i]);
150         }
151         end_time = std::chrono::steady_clock::now();
152         std::cout << "Time of free FBLAllocator: " <<
153                         std::chrono::duration_cast<std::chrono::
    milliseconds>(end_time - start_time).count() <<
154                         " milliseconds" << std::endl;
155 
156         start_time = std::chrono::steady_clock::now();
157         for (size_t i = 0; i != MKC_blocks.size(); ++i) {
158             MKC_alloc3.free(MKC_blocks3[i]);
159         }
160         end_time = std::chrono::steady_clock::now();
161 
162         std::cout << "Time of free MCKAllocator: " <<
163                         std::chrono::duration_cast<std::chrono::
    milliseconds>(end_time - start_time).count() <<
164                         " milliseconds" << std::endl;
165      //-------------------------------------------------------
166         std::vector<void*> list_blocks4;
167         std::vector<void*> MKC_blocks4;
168 
169         list_memory = sbrk(10000 * page_size * 100);
170         MKC_memory = sbrk(10000 * page_size * 100);
171 
172         FBLAllocator list_alloc4(list_memory, 10000 * page_size);
173         MCKAllocator MKC_alloc4(MKC_memory, 10000 * page_size);
174         std::cout << "Block allocation rate of 1000 bytes" << std
    ::endl;
175         start_time = std::chrono::steady_clock::now();
176         for (size_t i = 0; i != 10000000; ++i) {
177             void* block = list_alloc4.alloc(200);
178             list_blocks4.push_back(block);
179         }
180         end_time = std::chrono::steady_clock::now();
181         std::cout << "Time of alloc FBLAllocator: " <<
182                         std::chrono::duration_cast<std::chrono::
    milliseconds>(end_time - start_time).count() <<
183                         " milliseconds" << std::endl;
184 
185         start_time = std::chrono::steady_clock::now();
186         for (size_t i = 0; i != 10000000; ++i) {
187             void* block = MKC_alloc4.alloc(200);
188             MKC_blocks4.push_back(block);
189         }
190         end_time = std::chrono::steady_clock::now();
```

```
191         std::cout << "Time of alloc MCKAllocator: " <<
192                         std::chrono::duration_cast<std::chrono::
    milliseconds>(end_time - start_time).count() <<
193                         " milliseconds" << std::endl;
194 
195         std::cout << "Block free rate" << std::endl;
196         start_time = std::chrono::steady_clock::now();
197         for (size_t i = 0; i != list_blocks.size(); ++i) {
198             list_alloc4.free(list_blocks4[i]);
199         }
200         end_time = std::chrono::steady_clock::now();
201         std::cout << "Time of free FBLAllocator: " <<
202                         std::chrono::duration_cast<std::chrono::
    milliseconds>(end_time - start_time).count() <<
203                         " milliseconds" << std::endl;
204 
205         start_time = std::chrono::steady_clock::now();
206         for (size_t i = 0; i != MKC_blocks.size(); ++i) {
207             MKC_alloc4.free(MKC_blocks4[i]);
208         }
209         end_time = std::chrono::steady_clock::now();
210 
211         std::cout << "Time of free MCKAllocator: " <<
212                         std::chrono::duration_cast<std::chrono::
    milliseconds>(end_time - start_time).count() <<
213                         " milliseconds" << std::endl;
214 
215     }
```

6 Консоль

```
Comparing FBLAllocator and MCKAllocator
Block allocation rate
Time of alloc FBLAllocator: 6 milliseconds
Time of alloc MCKAllocator: 28 milliseconds
Block free rate
Time of free FBLAllocator: 0 milliseconds
Time of free MCKAllocator: 1 milliseconds
Block allocation rate of 1000 bytes
Time of alloc FBLAllocator: 8 milliseconds
Time of alloc MCKAllocator: 34 milliseconds
Block free rate
Time of free FBLAllocator: 0 milliseconds
Time of free MCKAllocator: 0 milliseconds
Block allocation rate of 3000 bytes
Time of alloc FBLAllocator: 8 milliseconds
Time of alloc MCKAllocator: 34 milliseconds
Block free rate
Time of free FBLAllocator: 0 milliseconds
Time of free MCKAllocator: 0 milliseconds
Block allocation rate of 1000 bytes
Time of alloc FBLAllocator: 171 milliseconds
Time of alloc MCKAllocator: 186 milliseconds
Block free rate
Time of free FBLAllocator: 1 milliseconds
Time of free MCKAllocator: 1 milliseconds
```

7 Выводы

В ходе данного курсового проекта я приобрёл практические навыки в использовании знаний, полученных в течении курса, а также провёл исследование 2 аллокаторов памяти: список свободных блоков (наиболее подходящее) и алгоритм Мак-Кьюзика-Кэрелса.

Сравнивая эти два способа аллокации памяти, основываясь на времени их работы и на самом принципе работы можно сделать вывод, что метод свободных блоков работает быстрее, чем алгоритм Мак-Кьюзи-Кэрелса, но с ростом количества выделяемых блоков разница в работе алгоритмов становится меньше. Также при выделении блоков разного размера методом свободных блоков эти блоки будут хаотично разбросаны по памяти, при работе же алгоритма Мак-Кьюзи-Кэрелса, блоки с одинаковым размером будут расположены друг за другом в области памяти.