Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Курсовой проект по курсу
«Операционные системы»

Студент: Друхольский А.К.
Группа: М8О-207Б-21
Вариант: 20
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Содержание

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

Репозиторий

https://github.com/ssForz/OS-labs

Постановка задачи

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

Общие сведения о программе

Списки свободных блоков

Fballoc.h, Fballoc.cpp

Алгоритм Мак-Кьюзи-Кэрелса

MKKalloc.h, MKKalloc.cpp

В main.cpp представлен код тестирования этих реализаций

Общий метод и алгоритм решения

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

Списки свободных блоков

На странице памяти определённого размера имеется список свободных блоков, между которыми и разбивается память страницы. Под блоки мы можем выделять различную память в рамках той, что доступна на странице. Информацию о блоках мы храним в формате одностороннего списка, но при этом храним указатель на первый блок данной страницы.
При выделении памяти по запросу fbmalloc наш алгоритм находит такой свободный блок (если не была осуществлена дефрагментация), который наиболее подходит по размеру под запрос пользователя. Это нужно для того, чтобы избежать хранения большого объёма незадействованной памяти в блоке, если это возможно.
Дефрагментация объединяет пустые блоки в более крупные. Если не вся память задействована, то всегда будет как минимум один свободный блок.

Алгоритм Мак-Кьюзи-Кэрелса

На странице расположены блоки, объём памяти которых равен степени двойки. Величина всех блоков одинакова. Блоки между собой не связаны, а обращаться к ним мы можем с помощью указателя на страницу памяти. В случае, если страниц несколько, то массив для управления страницами либо хранит число, равное размеру блоков на страницу, либо указатель на следующую страницу (при объединении нескольких страниц в один буфер, у нас будет храниться указатель на первую страницу буфера).
В случае данного алгоритма дефрагментация возможна, только когда все блоки страницы освобождены.
Для удобства в моей реализации (для одной страницы) я хранил вектор значений {1;0}, который указывает на то, занят блок или свободен. При вызове mkmalloc страница сразу разбивается на несколько блоков, размеров наиболее приближенно к тому, которое задано пользователем(округлено в большую сторону до степени двойки)

Исходный код

Fballoc.h

```
#pragma once
#include<iostream>
#include <cinttypes>

using namespace std;

class FBalloc
{
private:
   struct Block
   {
      Block* next_block_ptr;
      size_t availible_block_size;
   };

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

public:
   FBalloc();
```

```
    virtual ~FBalloc();
    void* fbmalloc(const size_t size);
    void fbfree(const void* ptr, const size_t size);
    void fb_print_info();
    void fbdefragment();

};
```

FBalloc.cpp

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

using namespace std;

FBalloc::FBalloc()
{
    //initialize method
    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 = (Block*)mem_page_ptr;
    first_block_ptr->next_block_ptr = nullptr;
    first_block_ptr->availible_block_size = mem_page_size;

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

```
void* FBalloc::fbmalloc(const size_t size)
{
   if (size == 0) {
      return nullptr;
   }
   //user cant request allocation of size less than size of Block struct
   const size_t req_size = max(size, sizeof(Block));
   size_t fit_min = 1e8;

   Block* block_ptr = first_block_ptr;

   while(block_ptr != nullptr) {
      //find more perfect fit of availible memory
      if (block_ptr->availible_block_size >= req_size + sizeof(Block)) {
         if (block_ptr->availible_block_size - (req_size + sizeof(Block)) <= fit_min) {
            fit_min = block_ptr->availible_block_size - (req_size + sizeof(Block));

         }
      }

      block_ptr = block_ptr->next_block_ptr;

   }



   //error if there are no fits
   if (fit_min == 1e8) {
      throw runtime_error("Error: Can't allocate requested memory");
   }
   block_ptr = first_block_ptr;
   while(block_ptr != nullptr) {

      //take this space in block
      if (block_ptr->availible_block_size - (req_size + sizeof(Block)) == fit_min) {
         uint8_t* take_block_space_ptr = (uint8_t*)block_ptr + block_ptr->availible_block_size - req_size;
         block_ptr->availible_block_size = block_ptr->availible_block_size - req_size;
```

```
            return take_block_space_ptr;
        }
        block_ptr = block_ptr->next_block_ptr;
    }
    throw runtime_error("Error: Can't allocate requested memory");
}


void FBalloc::fbfree(const void* ptr, const size_t size)
{
    if (ptr == nullptr) {
        return;
    }

    const size_t freed_space = max(size, sizeof(Block));

    Block* freed_block_ptr = (Block*)ptr;
    freed_block_ptr->next_block_ptr = nullptr;
    freed_block_ptr->availible_block_size = freed_space;

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

```
}

void FBalloc::fbdefragment()
{
   Block* block_ptr = first_block_ptr;
   while(block_ptr != nullptr) {

      while (block_ptr != nullptr) {
         uint8_t* end_ptr_of_block = (uint8_t*)block_ptr + block_ptr->availible_block_size;

         if (end_ptr_of_block != (uint8_t*)block_ptr->next_block_ptr) {
            break;
         }

         block_ptr->availible_block_size += block_ptr->next_block_ptr->availible_block_size;
         block_ptr->next_block_ptr = block_ptr->next_block_ptr->next_block_ptr;

      }


      block_ptr = block_ptr->next_block_ptr;
   }
}

void FBalloc::fb_print_info()
{
   int block_counter = 0;
   Block* block_ptr = first_block_ptr;
   cout<<"Free blocks allocator info:"<<endl;
   cout<<"Size of memory page: "<<mem_page_size<<endl;
   cout<<"Pointer of memory page: "<<(void*)mem_page_ptr<<endl;
   cout<<"Availible_blocks:"<<endl;
   cout<<"|№ | block pointer | block size |"<<endl;
   while(block_ptr != nullptr) {
```

```
        cout<<"|"<<block_counter<<" | "<< (void*)block_ptr<<"|    "<<block_ptr->availible_block_size<<"
|"<<endl;
        block_counter++;
        block_ptr = block_ptr->next_block_ptr;
    }
    cout<<endl;

}
```

MKKalloc.h

```
#pragma once
#include<iostream>
#include <cinttypes>
#include <vector>
using namespace std;

class MKalloc
{
private:

    size_t mem_page_size = 1024;
    uint8_t* page_ptr;

    size_t unit_size;
    size_t unit_count;
    size_t free_units_count;
    vector<int> frees;

public:
    MKalloc();
    virtual ~MKalloc();

    void* mkmalloc(const size_t size);
    void mkfree(const void* ptr);
    void mk_print_info();
    void mkdefragment();
```

```
};
```

MKKalloc.cpp

```
#include "MKKalloc.h"
#include <iostream>
#include <cinttypes>
#include<vector>

using namespace std;

MKalloc::MKalloc()
{
   //initialize method
   page_ptr = (uint8_t*)malloc(mem_page_size);
   if (page_ptr == nullptr) {
      throw runtime_error("Error: Cannot allocate a memory page");
   }
   unit_size = mem_page_size;
   unit_count = 1;
   free_units_count = 1;
   frees.push_back(1);

}
MKalloc::~MKalloc()
{
   if (page_ptr != nullptr) {
      ::free(page_ptr);
   }
}

void* MKalloc::mkmalloc(const size_t size)
{
   if (size == 0) {
      return nullptr;
   }
   if (size > mem_page_size) {
```

```
        throw runtime_error("Error: Cannot allocate a memory of size more than requested at start");
    }
    if (free_units_count == 0) {
        throw runtime_error("Error: Out of memory");
    }
    //convert request size to 2^n
    size_t ssize = size;
    for (ssize; ssize <= mem_page_size; ssize++) {
        if (ssize == 2 || ssize == 4 || ssize == 8 || ssize == 16 || ssize == 32 || ssize == 64 || ssize == 128 ||
ssize == 256 || ssize == 512 || ssize == 1024) {
            break;
        }
    }
    if (unit_count != 1) {
        if (unit_size != ssize) {
            throw runtime_error("Error: Cannot allocate a block of size different to one defined previously");
        }
    }
    if (unit_count == 1) {
        unit_size = ssize;
        unit_count = mem_page_size/ssize;
        free_units_count = unit_count;
        frees.resize(unit_count);
        for (int i = 0; i < unit_count; i++) {
            frees[i] = 1;
        }
    }
    uint8_t* unit_ptr;
    for (int i = 0; i < unit_count; i++) {
        if (frees[i] == 1) {
            unit_ptr = (uint8_t*)(page_ptr + mem_page_size - (unit_count-i)*unit_size);
            free_units_count--;
            frees[i] = 0;
            mem_page_size -= unit_size;
            return unit_ptr;
        }
```

```
    }
    throw runtime_error("Error: Cannot allocate");

}


void MKalloc::mkfree(const void* ptr)
{
    if (ptr == nullptr) {
        return;
    }
    for (int i = 0; i <= unit_count; i++) {
        if (ptr == page_ptr + i*unit_size) {
            mem_page_size += unit_size;
            free_units_count++;
            frees[i] = 1;
            return;
        }
    }



}

void MKalloc::mkdefragment()
{
    if (unit_count = free_units_count) {
        unit_count = 1;
        mem_page_size = 1024;
        unit_size = mem_page_size;
        free_units_count = unit_count;
        frees.resize(1);
        frees[0] = 1;
    }
}
```

```
void MKalloc::mk_print_info()
{
   int block_counter = unit_count;
   uint8_t* show_page_ptr = page_ptr;
   cout<<"MKK allocator info:"<<endl;
   cout<<"Availible size of memory page: "<<mem_page_size<<endl;
   cout<<"Pointer of memory page: "<<(void*)page_ptr<<endl;
   cout<<"Availible_blocks:"<<endl;
   cout<<"|№ | block pointer | block size |"<<endl;
   for (int i = 0; i < free_units_count; i++) {
      cout<<"|"<<i<<" | "<< (void*)(page_ptr + i*unit_size)<<"|    "<<unit_size<<"     |"<<endl;
   }
   cout<<endl;

}
```

main.cpp

```
#include <iostream>
#include "FBalloc.h"
#include "MKKalloc.h"
#include<chrono>
using namespace std;

int main()
{
   cout << "Memory allocators:" << endl;
   cout << "- Free blocks allocator (most fit policy)" << endl;
   cout << "- McKusick-Karels allocator" << endl;
   cout << endl<<endl;

   FBalloc a1;
   MKalloc a2;

   cout<<"Free blocks allocator (most fit policy):"<<endl<<endl;
   auto begin = chrono::steady_clock::now();
   a1.fb_print_info();
```

```
void* p1 = a1.fbmalloc(100);
cout << "p1 = " << p1 << endl << endl;
a1.fb_print_info();

void* p2 = a1.fbmalloc(100);
cout << "p2 = " << p2 << endl << endl;
a1.fb_print_info();

void* p3 = a1.fbmalloc(100);
cout << "p3 = " << p3 << endl << endl;
a1.fb_print_info();

a1.fbfree(p2, 100);
cout << "p2 freed" << endl << endl;
a1.fb_print_info();

a1.fbfree(p1, 100);
cout << "p1 freed" << endl << endl;
a1.fb_print_info();

a1.fbfree(p3, 100);
cout << "p3 freed" << endl << endl;
a1.fb_print_info();

a1.fbdefragment();
cout << "defragmented" << endl << endl;
a1.fb_print_info();
auto end = chrono::steady_clock::now();

auto elapsed_ms = std::chrono::duration_cast<chrono::microseconds>(end - begin);
cout << "The time: " << elapsed_ms.count() << " ms\n";
cout<<"- McKusick-Karels allocator"<<endl<<endl;

begin = chrono::steady_clock::now();
a2.mk_print_info();
```

```
    void* p4 = a2.mkmalloc(256);
    cout << "p4 = " << p4 << endl << endl;
    a2.mk_print_info();

    void* p5 = a2.mkmalloc(256);
    cout << "p5 = " << p5 << endl << endl;
    a2.mk_print_info();

    void* p6 = a2.mkmalloc(250);
    cout << "p6 = " << p6 << endl << endl;
    a2.mk_print_info();

    void* p7 = a2.mkmalloc(256);
    cout << "p7 = " << p7 << endl << endl;
    a2.mk_print_info();

    a2.mkfree(p4);
    cout << "p4 freed" << endl << endl;
    a2.mk_print_info();

    a2.mkfree(p7);
    cout << "p7 freed" << endl << endl;
    a2.mk_print_info();

    a2.mkfree(p5);
    cout << "p5 freed" << endl << endl;
    a2.mk_print_info();

    a2.mkfree(p6);
    cout << "p6 freed" << endl << endl;
    a2.mk_print_info();

    a2.mkdefragment();
    cout << "defragmented" << endl << endl;
    a2.mk_print_info();
    end = std::chrono::steady_clock::now();
```

```
    elapsed_ms = chrono::duration_cast<chrono::microseconds>(end - begin);
    cout << "The time: " << elapsed_ms.count() << " ms\n";

}
```

Демонстрация работы программы

```
kpout (output):
Memory allocators:
- Free blocks allocator (most fit policy)
- McKusick-Karels allocator


Free blocks allocator (most fit policy):


Free blocks allocator info:
Size of memory page: 1024
Pointer of memory page: 0x5558640b02c0
Availible_blocks:
|№ | block pointer | block size |
|0 | 0x5558640b02c0|    1024     |

p1 = 0x5558640b065c

Free blocks allocator info:
Size of memory page: 1024
Pointer of memory page: 0x5558640b02c0
Availible_blocks:
|№ | block pointer | block size |
|0 | 0x5558640b02c0|    924     |

p2 = 0x5558640b05f8

Free blocks allocator info:
Size of memory page: 1024
```

Pointer of memory page: 0x5558640b02c0

Availible_blocks:

|№ | block pointer | block size |

|0 | 0x5558640b02c0| 824 |

p3 = 0x5558640b0594

Free blocks allocator info:

Size of memory page: 1024

Pointer of memory page: 0x5558640b02c0

Availible_blocks:

|№ | block pointer | block size |

|0 | 0x5558640b02c0| 724 |

p2 freed

Free blocks allocator info:

Size of memory page: 1024

Pointer of memory page: 0x5558640b02c0

Availible_blocks:

|№ | block pointer | block size |

|0 | 0x5558640b02c0| 724 |

|1 | 0x5558640b05f8| 100 |

p1 freed

Free blocks allocator info:

Size of memory page: 1024

Pointer of memory page: 0x5558640b02c0

Availible_blocks:

|№ | block pointer | block size |

|0 | 0x5558640b02c0| 724 |

|1 | 0x5558640b05f8| 100 |

|2 | 0x5558640b065c| 100 |

p3 freed

Free blocks allocator info:

Size of memory page: 1024

Pointer of memory page: 0x5558640b02c0

Availible_blocks:

|№ | block pointer | block size |

|0 | 0x5558640b02c0| 724 |

|1 | 0x5558640b0594| 100 |

|2 | 0x5558640b05f8| 100 |

|3 | 0x5558640b065c| 100 |

defragmented

Free blocks allocator info:

Size of memory page: 1024

Pointer of memory page: 0x5558640b02c0

Availible_blocks:

|№ | block pointer | block size |

|0 | 0x5558640b02c0| 1024 |

The time: 306 ms

- McKusick-Karels allocator

MKK allocator info:

Availible size of memory page: 1024

Pointer of memory page: 0x5558640b06d0

Availible_blocks:

|№ | block pointer | block size |

|0 | 0x5558640b06d0| 1024 |

p4 = 0x5558640b06d0

MKK allocator info:

Availible size of memory page: 768

Pointer of memory page: 0x5558640b06d0

Availible_blocks:

|№ | block pointer | block size |

|0 | 0x5558640b06d0| 256 |

|1 | 0x5558640b07d0| 256 |

|2 | 0x5558640b08d0| 256 |

p5 = 0x5558640b06d0

MKK allocator info:

Availible size of memory page: 512

Pointer of memory page: 0x5558640b06d0

Availible_blocks:

|№ | block pointer | block size |

|0 | 0x5558640b06d0| 256 |

|1 | 0x5558640b07d0| 256 |

p6 = 0x5558640b06d0

MKK allocator info:

Availible size of memory page: 256

Pointer of memory page: 0x5558640b06d0

Availible_blocks:

|№ | block pointer | block size |

|0 | 0x5558640b06d0| 256 |

p7 = 0x5558640b06d0

MKK allocator info:

Availible size of memory page: 0

Pointer of memory page: 0x5558640b06d0

Availible_blocks:

|№ | block pointer | block size |

p4 freed

MKK allocator info:
Availible size of memory page: 256
Pointer of memory page: 0x5558640b06d0
Availible_blocks:
|№ | block pointer | block size |
|0 | 0x5558640b06d0| 256 |

p7 freed

MKK allocator info:
Availible size of memory page: 512
Pointer of memory page: 0x5558640b06d0
Availible_blocks:
№	block pointer	block size
0	0x5558640b06d0	256
1	0x5558640b07d0	256

p5 freed

MKK allocator info:
Availible size of memory page: 768
Pointer of memory page: 0x5558640b06d0
Availible_blocks:
№	block pointer	block size
0	0x5558640b06d0	256
1	0x5558640b07d0	256
2	0x5558640b08d0	256

p6 freed

MKK allocator info:

Availible size of memory page: 1024

Pointer of memory page: 0x5558640b06d0

Availible_blocks:

|№ | block pointer | block size |

|0 | 0x5558640b06d0| 256 |

|1 | 0x5558640b07d0| 256 |

|2 | 0x5558640b08d0| 256 |

|3 | 0x5558640b09d0| 256 |

defragmented

MKK allocator info:

Availible size of memory page: 1024

Pointer of memory page: 0x5558640b06d0

Availible_blocks:

|№ | block pointer | block size |

|0 | 0x5558640b06d0| 1024 |

The time: 356 ms

Выводы

В ходе выполнения курсового проекта я узнал как работают аллокаторы и познакомился с некоторыми алгоритмами аллокации.

Два представленных алгоритма можно сравнить.

Алгоритм со свободными блоками удобен в использовании, но не совсем грамотно использует память. Хоть поиск наиболее подходящего блока и призван минимализировать потери, они возможны. Так же сам алгоритм аллокации чуть более сложный из-за множества циклов перехода от первого блока к последнему. С другой стороны, у нас есть возможность выделять ровно столько памяти, сколько нам нужно.

Алгоритм Мак-Кьюзика-Кэрелса не позволяет нам свободно манипулировать памятью, поэтому не всегда найдёт применение, но в ситуациях, когда мы используем память численно равную степени двойки, мы получаем надёжный инструмент аллокации. Перераспределение памяти же в алгоритме подчинено более простому алгоритму, так как мы обращаемся к блокам используя только их индекс и указатель на начало страницы.

По времени алгоритмы сильно не отличаются в условиях проведенного выше теста (МКК алгоритм чуть медленнее в данных условиях).

Итак, каждый алгоритм находит применение и имеет свои достоинства и недостатки.