Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Курсовой проект по курсу**

**«Операционные системы»**

Студент: Муратов А.А.

Группа: М8О-212Б-22

Вариант: 19

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2023

**Постановка задачи**

### Аллигаторы 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 на блоках размером 2n**

Методика использует набор списков свободной памяти. В каждом

списке хранятся буферы определенного размера. Размер буфера всегда кратен степени числа 2. На рисунке ниже показан пример шести списков, содержащих

буферы размером 32, 64, 128, 256, 512 и 1024 байта соответственно.

Принцип работы аллокатора таков, что при запросе какого – то количества памяти сначала вычисляется какого размера потребуется буфер, затем в списке буферов ищется свободный буфер, после чего память выделяется пользователю.

**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 (MKK алгоритм чуть медленнее в данных условиях).

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