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

Курсовой проект по курсу
«Операционные системы»

Студент: Мусаелян Ярослав
Группа: М8О-207Б-21
Вариант: 19
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Содержание

Репозиторий

https://github.com/YMusaelyan/os

Постановка задачи

Цель работы

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

Задание

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

19. Необходимо сравнить два алгоритма аллокации: списки свободных блоков (наиболее подходящее) и блоки по 2 в степени n

Подробное описание каждого из исследуемых алгоритмов

Аллокатор или распределитель памяти в языке программирования C++ — специализированный класс, реализующий и инкапсулирующий малозначимые (с прикладной точки зрения) детали распределения и освобождения ресурсов компьютерной памяти.

Концептуально выделяется пять основных операции, которые можно осуществить над аллокатором (хочется отметить, что не все аллокаторы могут явно соответствовать этому интерфейсу):

- *create* – создает аллокатор и отдает ему в распоряжение некоторый объем памяти;
- *allocate* – выделяет блок определенного размера из области памяти, которым распоряжается аллокатор;
- *deallocate* – освобождает определенный блок;
- *free* – освобождает все выделенные блоки из памяти аллокатора

(память, выделенная аллокатору, не освобождается);
- *destroy* – уничтожает аллокатор с последующим освобождением памяти, выделенной аллокатору.

Для выделения памяти в куче могут быть использованы алгоритмы, построенные на любой из рассмотренных ранее стратегий размещения, например на методе первого подходящего или следующего подходящего. Для этого требуется только, чтобы в куче был в том или ином виде реализован список свободных блоков.

Просматривая такой список свободных блоков, можно без труда выбрать подходящий, блок в соответствии с заданной стратегией размещения. Однако, основные трудности при работе с кучей связаны не с выделением памяти, а с ее освобождением. Каждый блок содержит заголовок. В заголовке представлена информация о размере этого блока, размере предыдущего блока и логическая переменная, показывающая свободен ли блок. При запросе на выделение памяти определённого размера ищется первый подходящий свободный блок, в соответствии с заданной стратегией размещения. При деаллокации ранее выделенной памяти сначала проверяется доступность переданного указателя, после просто в заголовке меняем информацию о том, что блок доступен для использования.

Главная проблема состоит в необходимости выполнять объединение примыкающих свободных блоков, если таковые появляются в ходе освобождения памяти.

Если возвращаемые в кучу блоки памяти не объединять, то достаточно скоро вся куча будет разбита на множество мелких свободных блоков, не пригодных для повторного использования.

Алгоритм “наиболее подходящего”. При очередном запросе на выделение памяти подбирается в списке свободных блоков наименьший блок, размер которого больше или равен запросу. Алгоритм “наиболее подходящего” обеспечивает сохранение более крупных свободных блоков, но может потребовать просмотра всего списка свободных блоков.

Алгоритм на блоках 2 в степени n. При очередном запросе на выделение памяти размер блоков выравнивается под ближайщую большую степень двойки.

Процесс тестирования

Для процесса тестирования будем засекать время с помощью функции time() для аллокации и деаллокации памяти с помощью реализованных алгоритмов. Проводить тестирование будем используя вектор большого размера, где каждый элемент будет хранить указатель на выделенную память.

В результате работы получили:

Time allocate BinaryAllocator:8085

Time deallocate BinaryAllocator:83

Time allocate FreeBlocksAllocator:3286

Time deallocate FreeBlocksAllocator:87

Таким образом, можем сделать вывод, что освобождение память происходит почти за одно и тоже время, так как функции освобождения идентичные, но вот алгоритм на блоках 2 в степени n почти в три раза дольше выделяет память, чем алгоритм на свободных блоках наиболее подходящего.

Исходный код

main.cpp

```
#include <iostream>
#include <ctime>
#include <vector>
#include "binary_allocator.h"
#include "freeblock_allocator.h"

using namespace std;

int main() {
	int start, end, size;
	auto allocator1 = BinaryAllocator(1024000);
	vector<void *> pointer(10000);
	start = clock();
	for(int i = 0; i < 10000; i++) {
		pointer[i] = allocator1.allocate(i + 1);
	}
	end = clock();
	cout << "Time allocate BinaryAllocator:" << end - start <<"\n"<< endl;

	start = clock();
	for(int i = 0; i < 10000; i++) {
		allocator1.deallocate(pointer[i]);
	}
	end = clock();
	cout << "Time deallocate BinaryAllocator:" << end - start <<"\n"<< endl;

	auto allocator2 = FreeBlocksAllocator(1024000);
	start = clock();
	for(int i = 0; i < 10000; i++) {
		pointer[i] = allocator2.allocate(i + 1);
	}
	end = clock();
	cout << "Time allocate FreeBlocksAllocator:" << end - start <<"\n"<< endl;

	start = clock();
	for(int i = 0; i < 10000; i++) {
		allocator2.deallocate(pointer[i]);
	}
	end = clock();
	cout << "Time deallocate FreeBlocksAllocator:" << end - start <<"\n"<< endl;

	return 0;
```

```
}
```

allocator.h

```
#ifndef ALLOCATOR_H
#define ALLOCATOR_H

#include <iostream>

class Allocator {
public:
   typedef void value_type;
   typedef value_type *pointer;

   Allocator() = default;

   ~Allocator() {
      ::free(pointer_start);
   }

   void free() {
      auto *header = static_cast<Header *>(pointer_start);
      header->size = (size_total - size_header);
      header->available = true;
      size_used = size_header;
   };

   virtual pointer allocate(size_t size) = 0;
   virtual void deallocate(pointer ptr) = 0;

protected:

   struct Header {
   public:
      size_t size;
      size_t size_previous;
      bool available;

      inline Header *next() {
         return (Header *) ((char *) (this + 1) + size);
      }

      inline Header *previous() {
         return (Header *) ((char *) this - size_previous) - 1;
      }

   };

   const size_t size_header = sizeof(Header);
   pointer pointer_start = nullptr;
   pointer pointer_end = nullptr;
   size_t size_total = 0;
   size_t size_used = 0;

   Header* find(size_t size) {
      auto *header = static_cast<Header *>(pointer_start);
      while (!header->available || header->size < size) {
         header = header->next();
         if (header >= pointer_end) {
```

```
            return nullptr;
        }
    }
    return header;
}

void block_separation(Header *header, size_t chunk) {
    size_t size_block = header->size;
    header->size = chunk;
    header->available = false;
    if (size_block - chunk >= size_header) {
        auto *next = header->next();
        next->size_previous = chunk;
        next->size = size_block - chunk - size_header;
        next->available = true;
        size_used += chunk + size_header;
        auto *followed = next->next();
        if (followed < pointer_end) {
            followed->size_previous = next->size;
        }
    } else {
        header->size = size_block;
        size_used += size_block;
    }
}

bool check_address(void *ptr) {
    auto *header = static_cast<Header *>(pointer_start);
    while (header < pointer_end) {
        if (header + 1 == ptr){
            return true;
        }
        header = header->next();
    }
    return false;
}

};

#endif //ALLOCATOR_H
```

binary_allocator.h

```
#ifndef BINARYALLOCATOR_H
#define BINARYALLOCATOR_H

#include <cmath>
#include "allocator.h"

class BinaryAllocator : public Allocator {
public:

    size_t approximation(size_t size) {
        int i = 0;
        while (pow(2, i) < size){
            i++;
        }
        return (size_t) pow(2, i);
    };
```

```
    pointer allocate(size_t size) override {
        if (size <= 0) {
            std::cerr << "size must be bigger than 0\n";
            return nullptr;
        }
        size = approximation(size);
        if (size > size_total - size_used) {
            return nullptr;
        }
        auto *header = find(size);
        if (header == nullptr) {
            return nullptr;
        }
        block_separation(header, size);
        return header + 1;
    };

    void deallocate(pointer ptr) override {
        if (!check_address(ptr)) {
            return;
        }
        auto *header = static_cast<Header *>(ptr) - 1;
        header->available = true;
        size_used -= header->size;
        defragmentation(header);
    };

    explicit BinaryAllocator(size_t size) {
        size = approximation(size);
        if ((pointer_start = malloc(size)) == nullptr) {
            std::cerr << "failed to allocate memory\n";
            return;
        }
        size_total = size;
        pointer_end = static_cast<void *>(static_cast<char *>(pointer_start) + size_total);
        auto *header = (Header *) pointer_start;

        header->size = (size_total - size_header);
        header->size_previous = 0;
        size_used = size_header;
        header->available = true;
    };

private:

    void defragmentation(Header *header) {
        if (previous_free(header)) {
            auto *previous = header->previous();
            if (header->next() < pointer_end) {
                header->next()->size_previous += previous->size + size_header;
            }
            previous->size += header->size + size_header;
            size_used -= size_header;
            header = previous;
        }
        if (next_free(header)) {
            header->size += size_header + header->next()->size;
            size_used -= size_header;
```

```
            auto *next = header->next();
            if (next != pointer_end) { next->size_previous = header->size; }
        }
    }

    bool previous_free(Header *header) {
        auto *previous = header->previous();
        return header != pointer_start && previous->available;
    }

    bool next_free(Header *header) {
        auto *next = header->next();
        return header != pointer_end && next->available;
    }

};

#endif //BINARYALLOCATOR_H
```

freeblock_allocator.h

```
#ifndef FREEBLOCKSALLOCATOR_H
#define FREEBLOCKSALLOCATOR_H

#include "allocator.h"

class FreeBlocksAllocator : public Allocator {
public:

    pointer allocate(size_t size) override {
        if (size <= 0) {
            std::cerr << "size must be bigger than 0\n";
            return nullptr;
        }
        if (size > size_total - size_used) {
            return nullptr;
        }
        auto *header = find(size);
        if (header == nullptr) {
            return nullptr;
        }
        block_separation(header, size);
        return header + 1;
    };

    void deallocate(pointer ptr) override {
        if (!check_address(ptr)) {
            return;
        }
        auto *header = static_cast<Header *>(ptr) - 1;
        header->available = true;
        size_used -= header->size;
        defragmentation(header);
    };

    explicit FreeBlocksAllocator(size_t size) {
        if ((pointer_start = malloc(size)) == nullptr) {
            std::cerr << "failed to allocate memory\n";
            return;
```

```
        }
        size_total = size;
        pointer_end = static_cast<void *>(static_cast<char *>(pointer_start) + size_total);
        auto *header = (Header *) pointer_start;
        header->size = (size_total - size_header);
        header->size_previous = 0;
        header->available = true;
        size_used = size_header;
    };

private:

    void defragmentation(Header *header) {
        if (previous_free(header)) {
            auto *previous = header->previous();
            if (header->next() != pointer_end) {
                header->next()->size_previous += previous->size + size_header;
            }
            previous->size += header->size + size_header;
            size_used -= size_header;
            header = previous;
        }
        if (next_free(header)) {
            header->size += size_header + header->next()->size;
            size_used -= size_header;
            auto *next = header->next();
            if (next != pointer_end) { next->size_previous = header->size; }
        }
    }

    bool previous_free(Header *header) {
        auto *previous = header->previous();
        return header != pointer_start && previous->available;
    }

    bool next_free(Header *header) {
        auto *next = header->next();
        return header != pointer_end && next->available;
    }

};

#endif //FREEBLOCKSALLOCATOR_H
```

Выводы

В ходе проделанной работы я приобретение практические навыкы в использовании знаний, полученных в течении курса, провел исследование в области аллокаторов, сравнил два алгоритма аллокации памяти: списки свободных блоков (наиболее подходящее) и блоки по 2 в степени n.