МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ (НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

КУРСОВОЙ ПРОЕКТ

по курсу операционные системы I семестр, 2021/22 уч. год

Студент *Сикорский Александр Александрович, группа М8О-208Б-20*
Преподаватель *Миронов Евгений Сергеевич*
Вариант *21*
Оценка
Дата
Подпись

Содержание

Репозиторий

`https://github.com/sikorskii/os/tree/master/coursework`

Постановка задачи

Задание: Вариант 21:
Аллокаторы памяти: список свободных блоков и алгоритм двойников.

Общие сведения о программе

Исходный код лежит в 4 файлах, еще один - CMakeList:

1. src/main.cpp
2. Allocator.h- пытался сделать абстрактный класс для обоих аллокаторов, но не получилось.
3. BlockAllocator.h - реализация на списке свободных блоков.
4. BuddyAllocator.h - реализация алгоритма двойников.

Программа собирается с помощью Cmake. Нужно получить один исполняемый файл.

Общий метод и алгоритм решения

Вообще аллокаторы нужны для того, чтобы иметь возможность упростить и оптимизировать выделение, перемещение и освобождение памяти. У стандартных средств работы с памятью есть несколько недостатков. Тот же malloc - довольно тяжелый вызов. И что для тяжелого объекта, что для легкого он будет рабоать схожим образом. Кроме этого мы обращаемся к операционной системе, не имеем гарантий, как быстро отработает выделение памяти, а сам этот процесс довольно затратен по времени. Кроме этого мы сталкиваемся с проблемой, что куча, выделенная приложению оказывается фрагментирована через некоторое время и дефрагментировать её никто особо не собирается. Очень мала вероятность, но мы можем столкнуться с тем, что не найдется последовательный кусок памяти требуемой длины. Аллокатор на списках свободных блоков работает так,

что в ходе работы изначально выделенный кусок памяти разбивается на блоки по требованию. Так мы получаем список блоков, которые могут быть свободны или заняты. У блока должен быть заголовок, по которому можно узнать его размер, статус, кроме этого на основании адреса блока и его размера можно узнать адрес его соседа справа. При деаллокации необходимо пометить блок как свободный. К сожалению, такой аллокатор тоже подвержен фрагментаци. Аллокатор на алгоритме двойников подразумевает, что

блоки памяти являются блоками размера степени двойки. Когда мы просим память, аллокатор ищет блок подходящего размера. Если мы просим память размера меньше половины блока, блок нужно разбить пополам. Так и получается блок и его двойник. Блок можно разбивать и дальше, пока мы не получим наименьший подходящий кусок памяти. Свободные блоки можно сливать с двойниками. Для этого его двойник должен тоже быть свободен. Каждый блок содержит размер, статус свободен-занят, адрес следующего блока же можно узнать на основании этих данных. При поиске блока мы разбиваем наш единственный блок, если вся доступная аллокатору память пока что - единый участок или проходимся по нашему неявному списку в поисках подходящего блока. Сливание происходит в отдельной функции, которая работает над списком, пока не останется подходящих для объединения блоков.

Исходный код

main.cpp

```cpp
#include <iostream>
#include <unistd.h>
#include "BlockAllocator.h"
#include "BuddyAllocator.h"
#include <vector>
#include <chrono>

int main() {
    BlockAllocator all = BlockAllocator(1024);
    void* mem = all.allocate(10000000);
    BuddyAllocator allocator = BuddyAllocator(1024);

    std::vector<void*> v(100000);

    auto t1 = std::chrono::high_resolution_clock::now();

    for (int i = 0; i < v.size(); i++) {
        v[i] = allocator.malloc(i + (i / 2 % 5 + 1));
    }


    for (auto & i : v) {
        allocator.free(i);
    }

    auto t2 = std::chrono::high_resolution_clock::now();
```

```
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>( t2 - t1 ).cou
    std::cout << duration << std::endl;

    t1 = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < v.size(); i++) {
        v[i] = all.allocate(i + (i / 2 % 5 + 1));
    }
    for (auto & i : v) {
        all.deallocate(i);
    }
    t2 = std::chrono::high_resolution_clock::now();
    duration = std::chrono::duration_cast<std::chrono::microseconds>( t2 - t1 ).count();
    std::cout << duration;
}
```

Allocator.h

```
//
// Created by aldes on 25.12.2021.
//

#ifndef COURSEWORK_ALLOCATOR_H
#define COURSEWORK_ALLOCATOR_H


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

    Allocator() = default;

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

    virtual pointer allocate(size_type size) = 0;

    virtual void deallocate(pointer ptr) = 0;

    void free() {
        auto* header = static_cast<Header*>(startPointer);
```

```
        header->isAvailable = true;
        header->size = (totalSize - headerSize);
        usedSize = headerSize;
    }

protected:

    struct Header {
    public:
        size_type size;
        size_type previousSize;
        bool isAvailable;

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

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

    const size_type headerSize = sizeof(Header);
    const size_type blockAlignment = 4;
    pointer startPointer = nullptr;
    pointer endPointer = nullptr;
    size_type totalSize = 0;
    size_type usedSize = 0;

    Header* find(size_type size) {
        auto* header = static_cast<Header*>(startPointer);
        while (!header->isAvailable || header->size < size) {
            header = header->next();
            if (header >= endPointer) {
                return nullptr;
            }
        }
        return header;
    }

    void splitBlock(Header* header, size_type chunk) {
        size_type blockSize = header->size;
```

```
        header->size = chunk;
        header->isAvailable = false;
        if (blockSize - chunk >= headerSize) {
            auto *next = header->next();
            next->previousSize = chunk;
            next->size = blockSize - chunk - headerSize;
            next->isAvailable = true;
            usedSize += chunk + headerSize;
            auto *followed = next->next();
            if (followed < endPointer) {
                followed->previousSize = next->size;
            }
        } else {
            header->size = blockSize;
            usedSize += blockSize;
        }
    }

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


#endif //COURSEWORK_ALLOCATOR_H
```

BlockAllocator.h

```
//
// Created by aldes on 25.12.2021.
//

#ifndef COURSEWORK_BLOCKALLOCATOR_H
#define COURSEWORK_BLOCKALLOCATOR_H
#include "Allocator.h"
```

```
class BlockAllocator : public Allocator{
public:
    explicit BlockAllocator(size_type size) {
        if ((startPointer = malloc(size)) == nullptr) {
            std::cout << "cant allocate such memory" << std::endl;
            return;
        }
        totalSize = size;
        endPointer = static_cast<void*>(static_cast<char*>(startPointer) + totalSize);
        auto* header = (Header*) startPointer;
        header->isAvailable = true;
        header->size = (totalSize - headerSize);
        header->previousSize = 0;
        usedSize = headerSize;
    }

    pointer allocate(size_type size) override {
        if (size <= 0) {
            std::cout << "blockSize must be > 0" << std::endl;
            return nullptr;
        }
        if (size > totalSize - usedSize) {
            //throw std::bad_alloc();
        }
        auto* header = find(size);
        if (header == nullptr) {
            //throw std::bad_alloc();
            return nullptr;
        }
        splitBlock(header, size);
        return header + 1;
    }

    void deallocate(pointer ptr) override {
        if (!validateAddress(ptr)) {
            return;
        }
        auto* header = static_cast<Header*>(ptr) - 1;
        header->isAvailable = true;
        usedSize -= header->size;
    }
```

```
private:
    bool isPrevious(Header* header) {
        auto* previous = header->previous();
        return header != startPointer && previous->isAvailable;
    }

    bool isNextFree(Header* header) {
        auto* next = header->next();
        return header != endPointer && next->isAvailable;
    }
};


#endif //COURSEWORK_BLOCKALLOCATOR_H
```

BuddyAllocator.h

```
#include <iostream>
#include <cstring>
#include <cmath>

#define DIV_ROUNDUP(A, B) \
({ \
    typeof(A) _a_ = A; \
    typeof(B) _b_ = B; \
    (_a_ + (_b_ - 1)) / _b_; \
})

#define ALIGN_UP(A, B) \
({ \
    typeof(A) _a__ = A; \
    typeof(B) _b__ = B; \
    DIV_ROUNDUP(_a__, _b__) * _b__; \
})


struct BuddyBlock {
    size_t blockSize;
    bool isFree;
};

class BuddyAllocator {
private:
```

```
BuddyBlock *head = nullptr;
BuddyBlock *tail = nullptr;
void *data = nullptr;

bool expanded = false;

BuddyBlock* next(BuddyBlock *block) {
    return reinterpret_cast<BuddyBlock*>(reinterpret_cast<uint8_t*>(block) + block->
}

BuddyBlock* split(BuddyBlock *block, size_t size) {
    if (block != nullptr && size != 0) {
        while (size < block->blockSize) {
            size_t sz = block->blockSize >> 1;
            block->blockSize = sz;
            block = this->next(block);
            block->blockSize = sz;
            block->isFree = true;
        }
        if (size <= block->blockSize) return block;
    }
    return nullptr;
}

BuddyBlock* findBest(size_t size) {
    if (size == 0) return nullptr;

    BuddyBlock *bestBlock = nullptr;
    BuddyBlock *block = this->head;
    BuddyBlock *buddy = this->next(block);

    if (buddy == this->tail && block->isFree) {
        return this->split(block, size);
    }

    while (block < this->tail && buddy < this->tail) {
        if (block->isFree && buddy->isFree && block->blockSize == buddy->blockSize)
            block->blockSize <<= 1;
            if (size <= block->blockSize && (bestBlock == nullptr || block->blockSiz
                bestBlock = block;
            }
```

```
            block = this->next(buddy);
            if (block < this->tail)  {
                buddy = this->next(block);
            }
            continue;
        }

        if (block->isFree && size <= block->blockSize && (bestBlock == nullptr || bl
            bestBlock = block;
        }
        if (buddy->isFree && size <= buddy->blockSize && (bestBlock == nullptr || bu
            bestBlock = buddy;
        }

        if (block->blockSize <= buddy->blockSize) {
            block = this->next(buddy);
            if (block < this->tail) {
                buddy = this->next(block);
            }
        }
        else {
            block = buddy;
            buddy = this->next(buddy);
        }
    }

    if (bestBlock != nullptr) {
        return this->split(bestBlock, size);
    }

    return nullptr;
}

size_t requiredSize(size_t size) {
    size_t actual_size = sizeof(BuddyBlock);

    size += sizeof(BuddyBlock);
    size = ALIGN_UP(size, sizeof(BuddyBlock));

    while (size > actual_size) {
        actual_size <<= 1;
    }
```

```
        return actual_size;
    }

    void coalescence() {
        while (true) {
            BuddyBlock *block = this->head;
            BuddyBlock *buddy = this->next(block);

            bool noCoalescence = true;
            while (block < this->tail && buddy < this->tail) {
                if (block->isFree && buddy->isFree && block->blockSize == buddy->blockSi
                    block->blockSize <<= 1;
                    block = this->next(block);
                    if (block < this->tail) {
                        buddy = this->next(block);
                        noCoalescence = false;
                    }
                }
                else if (block->blockSize < buddy->blockSize) {
                    block = buddy;
                    buddy = this->next(buddy);
                }
                else {
                    block = this->next(buddy);
                    if (block < this->tail) {
                        buddy = this->next(block);
                    }
                }
            }

            if (noCoalescence) {
                return;
            }
        }
    }

public:
    bool debug = false;

    BuddyAllocator(size_t size) {
        this->expand(size);
```

```
}

~BuddyAllocator() {
    this->head = nullptr;
    this->tail = nullptr;
    std::free(this->data);
}

void expand(size_t size) {

    if (this->head) {
        size += this->head->blockSize;
    }
    size = pow(2, ceil(log(size) / log(2)));


    this->data = std::realloc(this->data, size);

    this->head = static_cast<BuddyBlock*>(data);
    this->head->blockSize = size;
    this->head->isFree = true;

    this->tail = next(head);

    if (this->debug) {
        std::cout << "Expanded the heap. Current blockSize: " << size << " bytes" <<
    }
}

void setsize(size_t size) {
    size -= this->head->blockSize;
    this->expand(size);
}

void *malloc(size_t size) {
    if (size == 0) return nullptr;

    size_t actualSize = this->requiredSize(size);

    BuddyBlock *found = this->findBest(actualSize);
    if (found == nullptr) {
        this->coalescence();
```

```
            found = this->findBest(actualSize);
        }

        if (found != nullptr) {
            found->isFree = false;
            this->expanded = false;
            return reinterpret_cast<void*>(reinterpret_cast<char*>(found) + sizeof(Buddy
        }

        if (this->expanded) {
            this->expanded = false;
            return nullptr;
        }
        this->expanded = true;
        this->expand(size);
        return this->malloc(size);
    }


    void free(void *ptr) {
        if (ptr == nullptr) {
            return;
        }

        BuddyBlock *block = reinterpret_cast<BuddyBlock*>(reinterpret_cast<char*>(ptr) -
        block->isFree = true;

        if (this->debug) {
            std::cout << "Freed " << block->blockSize - sizeof(BuddyBlock) << " bytes" <
        }

        this->coalescence();
    }
};
```

Демонстрация работы программы

Ниже приведен пример работы программы.

test.txt

```
/home/aldes/dev/MAI-labs/OS/coursework/cmake-build-debug/coursework
150
```

Выводы

В ходе работы я познакомился с тем, как реализуются аллокаторы памяти. Аллокатор на алгоритме двойников - довольно неплох и при этом его не так уж сложно реализовать. Сложность освобождения памяти у него O(1) если мы просто делаем free. Хотя при этом можно провести слияние блоков, чтобы выиграть по времени при других действиях. Алгоритм двойников работает быстрее списка свободных блоков, кроме этого он меньше подвержен фрагментации, а структура в виде блоков размером степеней двойки удобна и проста для представления. В случае списка блоков нам нужно еще тратить время на операции, связанные с содержанием блоков в списке. Еще можно реализовать этот аллокатор не на списке свободных блоков, а на красно-черном дереве. С одной стороны, это очень сильно усложняет реализацию. С моей точки зрения код такого дерева сложнее, чем код самого аллокатора. С другой стороны мы продолжаем сохранять довольно низкую пространственную сложность при выигрыше во времени. Древовидное представление позволяет прийти к логарифмической временной сложности. Кроме этого, следствием такого представления будет то, что будет проще находить наиболее подходящие для выделения блоки памяти.