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

КУРСОВОЙ ПРОЕКТ

по курсу

«Операционные системы»

Студент: Шумилова Александра
Группа: М8О-207Б-21
Вариант: 21
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Содержание

Репозиторий

https://github.com/tonsoleils/OS

Постановка задачи

Цель работы

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

Задание

Необходимо спроектировать и реализовать программный прототип в соответствии с выбранным вариантом. Произвести анализ и сделать вывод на основании данных, полученных при работе программного прототипа.
Вариант 21: Необходимо сравнить два алгоритма аллокации: списки свободных блоков (наиболее подходящее) и алгоритм двойников.

Общие сведения о программе

Программы компилируются из файлов main.cpp, node.cpp с использованием заголовочных файлов topology.h и zmq_f.h с помощью cmake и CMakeFile. Также используется заголовочные файлы: unistd.h, iostream, vector, zmq.hpp, sstream, string, map.

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. При очередном запросе на выделение памяти администратор кучи подбирает 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. Кроме того, со временем этот алгоритм имеет тенденцию к созданию большого количества свободных блоков малого размера, которые не смогут удовлетворить ни один запрос на выделение памяти.
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.

В файле Allocator.h находится реализация базы для аллокатора на списке свободных блоков. В файле BlockAllocator.h находится сама реализация аллокатора на списках свободных блоков. В файле BuddyAllocator.h находится реализация аллокатора на алгоритме близнецов. В файле main.cpp находится пример использования аллокаторов.

Тестировать аллокаторы будем на время аллокации и освобождения, варьируя количество запрашиваемой памяти. Результаты тестирования приведены ниже:

1. Malloc

	Список свободных блоков, t, мкс	Алгоритм двойников, t, мкс
v.size() = 100	44	20
v.size() = 1000	288	196
v.size() = 10000	1229	1164
v.size() = 100000	12312	12005

2. Free

	Список свободных блоков, t, мкс	Алгоритм двойников, t, мкс
v.size() = 100	22	3
v.size() = 1000	254	31
v.size() = 10000	1872	288
v.size() = 100000	10002	1962

Исходя из приведенных выше результатов можно сделать вывод, что malloc у обоих аллокаторов практически одинаково эффективен, но, в следствие многократного проведения эксперимента результат времени у аллокатора на списке свободных блоков был довольно изменчив (колебался от n до 2 * n). Free же у аллокатора на алгоритме двойников оказался намного эффективнее (получилось ускорение примерно в 10 раз), чем у аллокатора на списке свободных блоков. Скорее всего, это связано с особенностями алгоритмов и их реализациями.

Что касаемо использования, то аллокатор на свободных блоках лучше использовать там, где нам не страшна фрагментация (или в совокупности с дефрагментацией памяти). У алгоритма двойников же фрагментация минимальна.

Исходный код

Allocator.h

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

int main() {
    BlockAllocator all = BlockAllocator(1024);
    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));
    }


    auto t1_end = std::chrono::high_resolution_clock::now();
    auto t1_duration = std::chrono::duration_cast<std::chrono::microseconds>(t1_end -
t1).count();
    std::cout << "Алгоритм двойников malloc: " << t1_duration << std::endl;


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


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


    auto t2_end = std::chrono::high_resolution_clock::now();
    auto t2_duration = std::chrono::duration_cast<std::chrono::microseconds>( t2_end - t2
).count();
    std::cout << "Алгоритм двойников free: " << t2_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));
    }


    t1_end = std::chrono::high_resolution_clock::now();
    t1_duration = std::chrono::duration_cast<std::chrono::microseconds>(t1_end - t1).count();
    std::cout << "Список свободных блоков malloc: " << t1_duration << std::endl;


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

```
    for (auto & i : v) {
        all.deallocate(i);
    }
    t2_end = std::chrono::high_resolution_clock::now();
    t2_duration = std::chrono::duration_cast<std::chrono::microseconds>( t2_end - t2 ).count();
    std::cout << "Список свободных блоков free: " << t2_duration << std::endl;

    return 0;
}
```

BlockAllocator.h

```
#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->blockSize);
    }

    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->blockSize <=
bestBlock->blockSize)) {
                bestBlock = block;
            }

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

        if (block->isFree && size <= block->blockSize && (bestBlock == nullptr || block-
>blockSize <= bestBlock->blockSize))  {
            bestBlock = block;
        }
        if (buddy->isFree && size <= buddy->blockSize && (bestBlock == nullptr || buddy-
>blockSize < bestBlock->blockSize)) {
            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->blockSize) {
                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" <<
std::endl;
        }
    }

    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(BuddyBlock));
        }

        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) - sizeof(BuddyBlock));
        block->isFree = true;

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

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

main.cpp

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

```
int main() {
    BlockAllocator all = BlockAllocator(1024);
    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));
    }


    auto t1_end = std::chrono::high_resolution_clock::now();
    auto t1_duration = std::chrono::duration_cast<std::chrono::microseconds>(t1_end -
t1).count();
    std::cout << "Алгоритм двойников malloc: " << t1_duration << std::endl;


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


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


    auto t2_end = std::chrono::high_resolution_clock::now();
    auto t2_duration = std::chrono::duration_cast<std::chrono::microseconds>( t2_end - t2
).count();
    std::cout << "Алгоритм двойников free: " << t2_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));
    }
```

```
    t1_end = std::chrono::high_resolution_clock::now();
    t1_duration = std::chrono::duration_cast<std::chrono::microseconds>(t1_end - t1).count();
    std::cout << "Список свободных блоков malloc: " << t1_duration << std::endl;

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

    for (auto & i : v) {
        all.deallocate(i);
    }
    t2_end = std::chrono::high_resolution_clock::now();
    t2_duration = std::chrono::duration_cast<std::chrono::microseconds>( t2_end - t2 ).count();
    std::cout << "Список свободных блоков free: " << t2_duration << std::endl;

    return 0;
}
```

Демонстрация работы программы

tonsoleils@LAPTOP-31GE9NQM:/mnt/d/op_sys/os/coursework$ ./coursework

Алгоритм двойников malloc: 12005

Алгоритм двойников free: 2040

Список свободных блоков malloc: 12312

Список свободных блоков free: 14159

Выводы

Проделав данную работу, я спроектировала и реализовала программный прототип в соответствии с выбранным вариантом, сравнила по времени аллокации и освобождении памяти два алгоритма аллокации: списки свободных блоков (наиболее подходящее) и алгоритм двойников. Получила примерно равную эффективность по времени аллокации у обоих алгоритмов и более хорошие результаты по времени освобождения у аллокатора на алгоритме двойников.