Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

Курсовая работа по предмету «Операционные системы»

Студент: Пирязев М.А.

Преподаватель: Миронов Е.С.

Группа: М8О-207Б-22

Дата: 18.12.2023

Оценка:

Подпись:

Москва 2023 г.

Оглавление

Постановка задачи

Задание: Необходимо реализовать два алгоритма аллокации памяти и сравнить их. Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям free() и malloc(). Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра. В отчёте необходимо отобразить следующее:

- Подробное описание каждого из исследуемых алгоритмов.
- Процесс тестирования.
- Обоснование подхода тестирования.
- Результат тестирования.
- Заключение по проведённой работе.

Каждый аллокатор должен обладать следующим интерфейсом:

- Allocator* createMemoryAllocator(void* realMemory, size_t memory_size) - создание аллокатора памяти размера memory_size.
- void* alloc(Allocator* allocator, size_t block_size) - выделение памяти при помощи аллокатора размера block_size.
- void* free(Allocator* allocator, void* block) - возвращает выделенную память аллокатором.

Вариант 18: Необходимо сравнить два алгоритма аллокации: блоки по 2 в степени n и алгоритм двойников.

Общие сведения о теме работы

Аллокаторы – это важная часть работы операционных систем и языков программирования. Они отвечают за распределение и управление памятью, чтобы программы могли эффективно использовать ресурсы компьютера.

Аллокаторы памяти делят память на блоки разного размера и выделяют их по запросу программ. Когда программа больше не использует блоки памяти, аллокатор может освободить их для повторного использования. Это позволяет оптимизировать использование памяти и предотвратить утечки памяти.

Существуют различные алгоритмы аллокации памяти, такие как линейный аллокатор, сегментный аллокатор и аллокатор со свободным списком. Каждый из них имеет свои преимущества и недостатки, поэтому выбор аллокатора зависит от конкретных требований исходной программы. В данной курсовой работе представлена реализация алгоритмов аллокации памяти алгоритм двойников и блоки по 2 в степени n.

Важно, чтобы аллокаторы были эффективными и надежными. Они должны быстро выделять и освобождать память, минимизировать фрагментацию и предотвращать утечки памяти. Правильный выбор и оптимизация аллокатора могут значительно повысить производительность программы.

Алгоритм двойников

Принцип работы алгоритма двойной аллокации основан на следующих шагах:

1. Исходная область памяти, разбивается на блоки разных размеров. Каждый блок имеет заголовок, который содержит информацию о размере блока и его статусе (свободен или занят).

2. При запросе на выделение памяти алгоритм проверяет доступные блоки памяти и выбирает блок подходящего размера.

3. Если найден блок, который полностью соответствует запрошенному размеру, он отмечается как занятый, а указатель на начало блока возвращается как результат.

4. Если найден блок, который больше запрошенного размера, то блок разделяется на две части: одна используется для выделения памяти, а оставшаяся часть становится свободным блоком.

5. Если нет подходящего блока, алгоритм может использовать методы компактации или сборки мусора для сжатия свободных блоков, чтобы получить один большой свободный блок. Этот пункт алгоритма зависит от конкретной реализации аллокатора.

6. При освобождении выделенного блока памяти алгоритм отмечает его как свободный и проверяет смежные блоки для объединения соседних свободных блоков. Это позволяет уменьшить фрагментацию памяти и создать большие свободные блоки.

buddy.cpp

```
 #include "buddy.h"
#include "consts.h"

unsigned int buddy_allocator::get_order(unsigned int mem_size){
    unsigned int i = 0;
    while (std::pow(2,i)<(mem_size/sizeof(char))) {
        i++;
    }
    return i;
}

buddy_allocator::buddy_allocator(){
    memory_ptr = new char[SIZE];
    memory.push_back(new block(get_order(SIZE),memory_ptr));
}

char *buddy_allocator::malloc(unsigned int size){
```

```
    unsigned int order_of_request = get_order(size);
    bool found_block_of_correct_order = false;

    if(((1<<order_of_request)+total_allocated)>SIZE){
        throw std::runtime_error("not enough memory");
    }
    bool blocks_available = false;
    total_allocated+=(1<<order_of_request);
    while (!found_block_of_correct_order) {
        for(auto b: memory){
            if(b->taken) continue;
            if(b->order == order_of_request){
                found_block_of_correct_order = true;
                b->taken = true;
                return b->ptr;
            }
        }
        for(auto b: memory){
            if(b->taken) continue;
            if(b->order > order_of_request){
                b->order--;
                memory.insert(std::find(memory.begin(),memory.end(),b)+1,new block(b-
>order,b->ptr+std::lround(std::pow(2,b->order))));
                blocks_available = true;
                break;
            }
        }
        if (!(blocks_available || found_block_of_correct_order)) throw std::runtime_er-
ror("no space left in allocator");
    }
    return nullptr;
}

void buddy_allocator::free(char *ptr) {
    for (auto it = memory.begin(); it != memory.end(); ++it) {
        if ((*it)->ptr == ptr && (*it)->taken) {
            total_allocated-=(1<<(*it)->order);
            (*it)->taken = false;
            auto buddy_allocator = find_buddy(it);
            while (buddy_allocator != memory.end() && !(*buddy_allocator)->taken) {
                merge(it, buddy_allocator);
                buddy_allocator = find_buddy(it);
            }
            break;
        }
    }

}
```

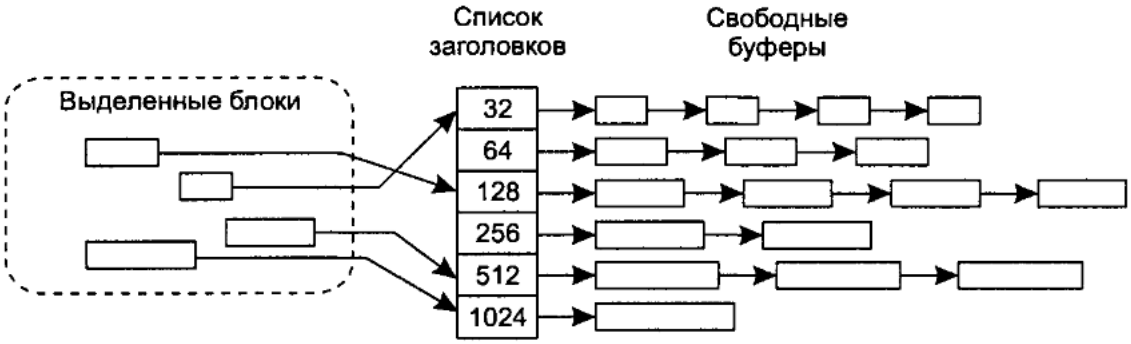
```
std::vector<block*>::iterator buddy_allocator::find_buddy(std::vector<block*>::iterator
it) {
    uintptr_t buddy_address = ((uintptr_t)(*it)->ptr ^ (1 << (*it)->order));
    for (auto buddy_allocator = memory.begin(); buddy_allocator != memory.end();
++buddy_allocator) {
        if ((uintptr_t)(*buddy_allocator)->ptr == buddy_address && (*buddy_allocator)-
>order == (*it)->order) {
            return buddy_allocator;
        }
    }
    return memory.end();
}

void buddy_allocator::merge(std::vector<block*>::iterator it, std::vector<block*>::iter-
ator buddy_allocator) {
    if ((*it)->ptr > (*buddy_allocator)->ptr) {
        std::swap(it, buddy_allocator);
    }
    (*it)->order++;
    delete *buddy_allocator;
    memory.erase(buddy_allocator);
}

buddy_allocator::~buddy_allocator() {
    delete[] memory_ptr;
    for(auto b: memory){
        delete b;
    }
    memory.clear();
}
```

Аллокатор на блоках размером 2^n

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.

twon.cpp

```
#include "twon.h"
#include "consts.h"

twon::twon(){
    memory_ptr = new char[SIZE];
    int size_for_division = SIZE;
    std::vector<buffer> vector_for_1024;
    std::vector<buffer> vector_for_512;
    std::vector<buffer> vector_for_256;
    std::vector<buffer> vector_for_128;
    std::vector<buffer> vector_for_64;
    std::vector<buffer> vector_for_32;

    if (SIZE <= 32) {
        throw std::logic_error("Minimal size of a buffer can`t be less than 32 bytes");
    }
    unsigned int step {0};
    while (size_for_division >= 32){
        if (size_for_division >= 1024){
            vector_for_1024.push_back(buffer(memory_ptr + step));
            size_for_division -= 1024;
            step += 1024;
```

```
        }
        if (size_for_division >= 512){
            vector_for_512.push_back(buffer(memory_ptr + step));
            size_for_division -= 512;
            step += 512;
        }
        if (size_for_division >= 256){
            vector_for_256.push_back(buffer(memory_ptr + step));
            size_for_division -= 256;
            step += 256;
        }
        if (size_for_division >= 128){
            vector_for_128.push_back(buffer(memory_ptr + step));
            size_for_division -= 128;
            step += 128;
        }
        if (size_for_division >= 64){
            vector_for_64.push_back(buffer(memory_ptr + step));
            size_for_division -= 64;
            step += 64;
        }
        if (size_for_division >= 32){
            vector_for_32.push_back(buffer(memory_ptr + step));
            size_for_division -= 32;
            step += 32;
        }
    }

    memory.insert(std::make_pair(1024,vector_for_1024));
    memory.insert(std::make_pair(512,vector_for_512));
    memory.insert(std::make_pair(256,vector_for_256));
    memory.insert(std::make_pair(128,vector_for_128));
    memory.insert(std::make_pair(64,vector_for_64));
    memory.insert(std::make_pair(32,vector_for_32));

}

twon::~twon() {
    delete[] memory_ptr;
    for (auto& pair : memory) {
        pair.second.clear();
    }
    memory.clear();
}

unsigned int twon::get_order(unsigned int mem_size){
    unsigned int i = 0;
    while (std::pow(2,i)<(mem_size/sizeof(char))) {
        i++;
    }
```

```
    if (i < 5) {
        return 5;
    }
    return i;
}

char* twon::malloc(unsigned int size){
    unsigned int order_of_request = get_order(size);

    switch (order_of_request)
    {
    case 5:
        return search(std::round(pow(2, order_of_request)));
        break;
    case 6:
        return search(std::round(pow(2, order_of_request)));
        break;
    case 7:
        return search(std::round(pow(2, order_of_request)));
        break;
    case 8:
        return search(std::round(pow(2, order_of_request)));
        break;
    case 9:
        return search(std::round(pow(2, order_of_request)));
        break;
    case 10:
        return search(std::round(pow(2, order_of_request)));
        break;
    default:
        throw std::invalid_argument("Invalid order. Probably your query is too big");
        break;
    }

}

void twon::free(char* ptr) {
    for (auto& pair : memory) {
        for (auto& buf : pair.second) {
            if (buf.ptr == ptr) {
                if (buf.taken == false) {
                    throw std::logic_error("Double freeing of memory");
                }
                buf.taken = false;
                return;
            }
        }
    }
    throw std::invalid_argument("Invalid pointer");
}
```

```
inline char *twon::search(int key)
{
    while (key <= 1024){
        auto it = memory.find(key);
        if (it != memory.end() and !it->second.empty()){
            for (int i = 0; i < it->second.size(); i++){
                if (it->second[i].taken == false){
                    it->second[i].taken = true;
                    return it->second[i].ptr;
                }
            }
        }
        key *= 2;
    }
    throw std::runtime_error("Not enough free memory");
}
```

Сравнение алгоритмов аллокации

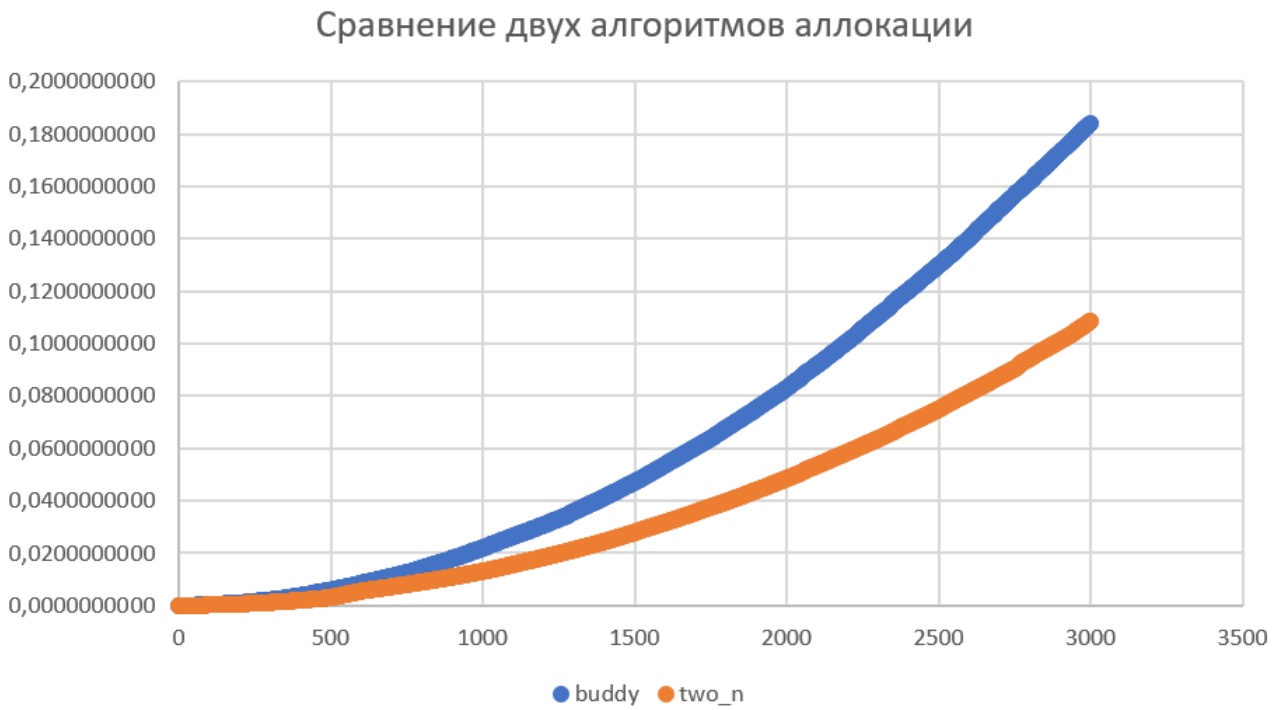
Размер блоков: В алгоритме двойников блоки памяти разделены на разные размеры, в то время как в алгоритме аллокации на блоках по 2 в степени n используются блоки, размер которых представляет собой степень двойки. Этот аспект важен при планировании и оптимизации использования памяти, так как использование блоков фиксированного размера может быть более эффективно.

Фрагментация памяти: Оба алгоритма позволяют решить проблему фрагментации памяти. В алгоритме двойников блоки памяти могут быть динамически объединены или разделены, чтобы уменьшить фрагментацию. В алгоритме аллокации на блоках по 2 в степени n блоки фиксированного размера не подвержены фрагментации на мелкие части, но могут возникать проблемы с неиспользованными частями блоков.

Эффективность выделения и освобождения памяти: Оба алгоритма обеспечивают эффективное выделение и освобождение блоков памяти. В алгоритме двойников можно выделить блок памяти нужного размера, а если блок больше, чем нужно, он разделится на две части. В алгоритме аллокации на блоках по 2 в степени 2n можно выделить только блок с фиксированным размером. Однако в силу того, что алгоритм не делает причудливых манипуляций с размерами блоков, он работает быстрее.

Сложность алгоритмов: Алгоритм двойников более гибкий и может быть сложнее в реализации и оптимизации, особенно при наличии большого числа разных размеров блоков. Алгоритм аллокации на блоках по 2 в степени n более простой и имеет фиксированный набор размеров блоков, что облегчает его реализацию. По данным тестов алгоритм выделения памяти по блокам 2 в степени n в большинстве случаев работает в 2 раза быстрее алгоритма двойников.

Вывод



На данной диаграмме представлена зависимость количества выделяемых элементов от времени для алгоритма двойников и блоков памяти по 2 в ствпени n. По данным диаграммы алгортм двойников работает в среднем в 2 раза медленнее, чем аллокатор на блоках по 2 в степени n. Для этого эксперимента у аллокаторов были запрошены по 3000 блоков памяти размером 32 бита. С точки зрения реализации алгоритм двойников более сложен, однако показывает себя лучше в тех ситуациях, когда пользователь запросил чуть больше памяти, чем минимальный размер свободного блока, у алгоритма 2 в степени n нет возможности уменьшить потери по памяти, так как нет возможности менять размеры своих блоков. Функция очистки памяти у алгоритма блоков по 2 в степени n также работает быстрее, потому что в отличие от алгоритма двойников ей нет необходимости искать соседа и соединять фрагменты памяти после освобождения близлежайших блоков.