Московский Авиационный Институт

(Национальный Исследовательский институт)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовая работа по курсу

«Операционные системы»

«Аллокаторы памяти»

Группа: М8О-206Б-18
Студент: Семенов И.М.
Преподаватель: Миронов Е.С.
Оценка:_____
Дата:_____

Москва, 2019

Содержание

Постановка задачи

Необходимо реализовать два алгоритма аллокации памяти и провести их тщательное сравнение по различным характеристикам

Сведения о программе

Программа написана на C++, в среде Ubuntu 18.04.3, сборка произведена посредством CMake. Программа использует библиотеку STL, в частности контейнеры(vector, list), алгоритмы(find_if, lower_bound). Для тестирования применяются функции из заголовочного файла chrono. Программа состоит из двух пар cpp/h файлов для каждого аллокатора и тестирующего исполняемого файла.

Аллокаторы памяти

Операционная система управляет всей доступной физической памятью машины и производит ее выделение для остальных подсистем ядра и прикладных задач. Данной процедурой управляет ядро, оно же и освобождает память, когда это требуется.

Менеджером памяти(аллокатором) называется часть ОС, непосредственно обрабатывающая запросы на выделение и освобождение памяти.

Существуют разные алгоритмы для реализации аллокаторов. Каждый из них имеет свои особенности и недостатки. Для данной курсовой работы мной были выбраны два алгоритма аллокации

- Алгоритм с выбором наиболее подходящего участка памяти, основанный на списках
- Простые списки, основанные на степени двойки.

Рассмотрим подробнее алгоритмы их реализации и характеристики.

Наиболее подходящий участок(списки)

Этот способ отслеживает память с помощью связных списков распределенных и свободных сегментов памяти, где сегмент содержит либо свободную, либо выделенную память. Каждый элемент списка хранит внутри свое обозначение — является ли он хранилищем выделенной или освобожденной памяти, а также размер участка памяти и указатель на его начало.

Список поддерживает инвариант отсортированности элементов по адресам с самой инициализации аллокатора. Благодаря этому, упрощается обновление списка при выделении или освобождении памяти. Для таких списков существует 4 алгоритма выделения памяти:

- **Первое подходящее** — список сегментов сканируется, пока не будет найдено пустое простанство подходящего размера. После этого сегмент разбивается на два сегмента, один из которых будет пустым. Данный алгоритм довольно быстр, ведь поиск ведется с наименьшими затратами времени.
- **Следующее подходящее** — работает примерно так же, как и предыдущий алгоритм, за исключением того, что запоминает свое местоположение при выделении. При следующем запросе на выделение памяти поиск начинается с того места, на котором алгоритм остановился в предыдущий раз. Исследование работы алгоритма показало, что его производительность несколько хуже, чем у **«первого подходящего»**
- **Наиболее подходящее** — этот алгоритм интересует нас больше всего. При нем ведется линейный поиск наименьшего по размеру подходящего сегмента. Это делается для того, чтобы наилучшим образом соответствовать запросу и имеющимся пустым пространствам в списке
- **Наименее подходящее** — алгоритм, противоположный вышеописанному — при выделении используется наибольший возможный сегмент памяти. Моделирование показало, что использование данного алгоритма не является хорошей идеей.

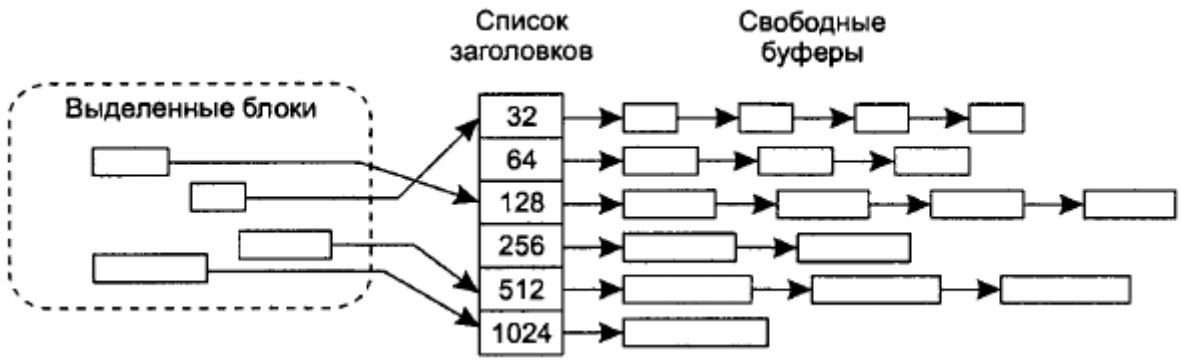
Далее речь будет идти об алгоритме «наиболее подходящее». Этот алгоритм работает медленнее, чем «первое подходящее»(за счет того, что при каждом запросе ведется поиск по всему списку). Парадоксально, но его применение даже приводит к более расточительному использованию памяти, чем использование «первого подходящего» или «следующего подходящего», так как

алгоритм стремится выбирать сегменты, наиболее близко подходящие к необходимому размеру памяти, оставляя при этом небольшие бесполезные пустые пространства.

Работа всех вышеописанных алгоритмов может быть ускорена за счет ведения отдельных списков для занятых и для пустых пространств. Это ускоряет выделение памяти, но замедляет процедуру освобождения памяти. Так или иначе, даже при всех улучшениях данные алгоритмы достаточно сильно страдают от фрагментации

Списки, основанные на степени двойки

Списки, основанные на степени числа 2, чаще всего применяются для реализации процедур выделения и освобождения памяти в библиотеке C прикладного уровня. Эта методика использует набор списков свободной памяти, в каждом из которых хранятся буферы определенного размера. Размер буфера всегда кратен степени двойки



При этом каждый свободный буфер хранит в себе либо указатель на следующий свободный буфер, либо размер буфера, либо указатель на список, которому принадлежит буфер(в моей реализации буфер содержит свой размер).

При поступлении запроса на аллокацию памяти, к запрошенному размеру прибавляется размер памяти, необходимый для хранения размера буфера. Из списка, содержащего минимальные по размеру буферы, удовлетворяющие запросу извлекается(и удаляется) любой элемент(для простоты будем извлекать первый). Размер памяти был записан в буфер при инициализации, так что достаточно будет увеличить указатель на начало буфера на число байт, необходимое для хранения размера и вернуть его из функции. При операции освобождения, буфер освобождается только целиком, нет возможности частично освободить буфер

Если список оказался пустым, существует несколько способов решения этой проблемы

- Заблокировать запрос до освобождения буфера нужной длины
- Удовлетворить запрос путем выделения буфера большего размера(именно такой способ используется в моей реализации)
- Запросить дополнительную память от распределителя страничного уровня.

Данный алгоритм является весьма простым и быстрым. Этот аллокатор предоставляет понятный программный интерфейс, важнейшим преимуществом которого является процедура освобождения, а именно то, что в нее не нужно передавать размер буфера. Однако алгоритм имеет несколько значимых недостатков. Округление количества памяти вверх до ближайшей степени двойки приводит к неэкономному распределению памяти. Так же проблемой являются запросы, равные или очень близкие к очередной степени двойки. Необходимость хранения заголовка буфера влечет к перерасходу выделенной памяти в 2 раза. Так же не поддерживается слияние смежных буферов.

Реализации аллокаторов

Списки, основанные на степени двойки

N2Allocator.h

```
#pragma once
#include <iostream>
#include <list>
#include <algorithm>
#include <vector>
struct N2AllocatorInit {
    unsigned int block_16 = 0;
    unsigned int block_32 = 0;
    unsigned int block_64 = 0;
    unsigned int block_128 = 0;
    unsigned int block_256 = 0;
    unsigned int block_512 = 0;
    unsigned int block_1024 = 0;
};
class N2Allocator {
public:
    explicit N2Allocator(const N2AllocatorInit& init_data);
```

```
    ~N2Allocator();
    void* allocate(size_t mem_size);
    void deallocate(void *ptr);
    void PrintStatus(std::ostream& os) const;
private:
    const std::vector<int> index_to_size = {16,32,64,128,256,512,1024};
    std::vector<std::list<char*>> lists;
    char* data;
    int mem_size;
};
```

N2Allocator.cpp

```
#include "N2Allocator.h"
N2Allocator::N2Allocator(const N2AllocatorInit &init_data)
        : lists(index_to_size.size()) {
    std::vector<unsigned int> mem_sizes = {init_data.block_16,
                                           init_data.block_32,
                                           init_data.block_64,
                                           init_data.block_128,
                                           init_data.block_256,
                                           init_data.block_512,
                                           init_data.block_1024};
    unsigned int sum = 0;
    for (int i = 0; i < mem_sizes.size(); ++i) {
        sum += mem_sizes[i] * index_to_size[i];
    }
    data = (char *) malloc(sum);
    char *data_copy = data;
    for (int i = 0; i < mem_sizes.size(); ++i) {
        for (int j = 0; j < mem_sizes[i]; ++j) {
            lists[i].push_back(data_copy);
            *((int*)data_copy) = (int)index_to_size[i];
            data_copy += index_to_size[i];
        }
    }
    mem_size = sum;
}
N2Allocator::~N2Allocator() {
    free(data);
}
void *N2Allocator::allocate(size_t mem_size) {
    if (mem_size == 0) {
        return nullptr;
```

```
    }
    mem_size += sizeof(int);
    int index = -1;
    for (int i = 0; i < lists.size(); ++i) {
        if (index_to_size[i] >= mem_size && !lists[i].empty()) {
            index = i;
            break;
        }
    }
    if (index == -1) {
        throw std::bad_alloc();
    }
    char *to_return = lists[index].front();
    lists[index].pop_front();
    return (void*)(to_return + sizeof(int));
}
void N2Allocator::deallocate(void *ptr) {
    char *c_ptr = (char *) (ptr);
    c_ptr = c_ptr - sizeof(int);
    int block_size = *((int*)c_ptr);
    int index = std::lower_bound(index_to_size.begin(), index_to_size.end(),
block_size) - index_to_size.begin();
    if (index == index_to_size.size()) {
        throw std::logic_error("this pointer wasnt allocated by this
allocator");
    }
    lists[index].push_back(c_ptr);
}
void N2Allocator::PrintStatus(std::ostream& os) const {
    int free_sum = 0;
    for (int i = 0; i < lists.size(); ++i) {
        os << "List with " << index_to_size[i] << " byte blocks, size: " <<
lists[i].size() << "\n";
        free_sum += lists[i].size() * index_to_size[i];
    }
    int occ_sum = mem_size - free_sum;
    os << "Occupied memory " << occ_sum << "\n";
    os << "Free memory " << free_sum << "\n\n";
}
```

Наиболее подходящий участок(списки)

ListAllocator.h

```
#pragma once
#include <iostream>
#include <algorithm>
#include <list>
enum class MemoryNodeType {
    Hole,
    Occupied
};
struct MemoryNode {
    char* beginning;
    size_t capacity;
    MemoryNodeType type;
};
std::ostream& operator << (std::ostream& os, const MemoryNode& node);
class ListAllocator {
public:
    explicit ListAllocator(size_t data_size);
    ~ListAllocator();
    void* allocate(size_t mem_size);
    void deallocate(void* ptr);
    void PrintStatus(std::ostream& os) const;
private:
    std::list<MemoryNode> mem_list;
    char* data;
};
```

ListAllocator.cpp

```
#include "ListAllocator.h"
std::ostream& operator << (std::ostream& os, const MemoryNode& node) {
    return os << "Node: capacity " << node.capacity << ", type " << (node.type
== MemoryNodeType::Hole ? "Hole" : "Occupied");
}
ListAllocator::ListAllocator(size_t data_size) {
    data = (char *) malloc(data_size);
    mem_list.push_front({data, data_size, MemoryNodeType::Hole});
}
ListAllocator::~ListAllocator() {
```

```
    free(data);
}
void *ListAllocator::allocate(size_t mem_size) {
    if (mem_size == 0) {
        return nullptr;
    }
    size_t size_of_node = 0;
    auto needed_node = mem_list.end();
    for (auto it = mem_list.begin(); it != mem_list.end(); ++it) {
        if (it->type == MemoryNodeType::Hole && it->capacity >= mem_size &&
            (size_of_node == 0 || it->capacity < size_of_node)) {
            size_of_node = it->capacity;
            needed_node = it;
        }
    }
    if (size_of_node == 0) {
        throw std::bad_alloc();
    }
    if (mem_size == size_of_node) {
        needed_node->type = MemoryNodeType::Occupied;
    } else {
        MemoryNode new_node{needed_node->beginning + mem_size,
                            needed_node->capacity - mem_size,
                            MemoryNodeType::Hole};
        needed_node->capacity = mem_size;
        needed_node->type = MemoryNodeType::Occupied;
        mem_list.insert(std::next(needed_node), new_node);
    }
    return (void *) (needed_node->beginning);
}
void ListAllocator::deallocate(void *ptr) {
    auto it = std::find_if(mem_list.begin(), mem_list.end(), [ptr](const
MemoryNode &node) {
        return node.beginning == (char *) ptr && node.type ==
MemoryNodeType::Occupied;
    });
    if (it == mem_list.end()) {
        throw std::logic_error("This pointer wasnt allocated by this
allocator");
    }
    it->type = MemoryNodeType::Hole;
    if (it != mem_list.begin() && std::prev(it)->type == MemoryNodeType::Hole) {
        auto prev_it = std::prev(it);
        prev_it->capacity += it->capacity;
```

```
        mem_list.erase(it);
        it = prev_it;
    }
    if (std::next(it) != mem_list.end() && std::next(it)->type ==
MemoryNodeType::Hole) {
        auto next_it = std::next(it);
        it->capacity += next_it->capacity;
        mem_list.erase(next_it);
    }
}
void ListAllocator::PrintStatus(std::ostream& os) const {
    int occ_sum = 0;
    int free_sum = 0;
    for (const auto& elem : mem_list) {
        os << elem << "\n";
        if (elem.type == MemoryNodeType::Hole) {
            free_sum += elem.capacity;
        } else {
            occ_sum += elem.capacity;
        }
    }
    os << "Occupied memory " << occ_sum << "\n";
    os << "Free memory " << free_sum << "\n\n";
}
```

Тестирование

Будем тестировать следующие характеристики:

- Скорость выделения и освобождения блоков
- Фрагментацию
- Экономичность

Напишем программу для тестов и запустим их:

Запуск тестирующей программы

ilya@ilya-lenovo:~/CLionProjects/oskp/cmake-build-debug$ ./oskp

List allocator initialization with one page of memory :4649 ns

N2 allocator initialization with one page of memory :39516 ns

First test: Allocate 10 char[256] arrays, free 5 of them, allocate 10 char[128] arrays:

N2 allocator first test:8 microseconds

List with 16 byte blocks, size: 0

List with 32 byte blocks, size: 0

List with 64 byte blocks, size: 20

List with 128 byte blocks, size: 20

List with 256 byte blocks, size: 10

List with 512 byte blocks, size: 5

List with 1024 byte blocks, size: 0

Occupied memory 5120

Free memory 8960

List allocator first test:20 microseconds

Node: capacity 256, type Occupied

Node: capacity 256, type Occupied

Node: capacity 256, type Occupied

Node: capacity 256, type Occupied

Node: capacity 256, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 128, type Occupied

Node: capacity 1536, type Hole

Occupied memory 2560

Free memory 1536

Second test: Allocate and free 750 20 bytes arrays:

N2 allocator second test:

Allocation :105 microseconds

Deallocation :222 microseconds

List allocator second test:

Allocation :8025 microseconds

Deallocation :294 microseconds

Third test: Allocate 500 20 bytes arrays, deallocate every second, allocate 250 12 bytes :

N2 allocator third test:233 microseconds

List with 16 byte blocks, size: 150

List with 32 byte blocks, size: 450

List with 64 byte blocks, size: 0

List with 128 byte blocks, size: 0

List with 256 byte blocks, size: 0

List with 512 byte blocks, size: 0

List with 1024 byte blocks, size: 0

Occupied memory 12000

Free memory 16800

List allocator third test:12035 microseconds

Node: capacity 12, type Occupied

Node: capacity 8, type Hole

Node: capacity 20, type Occupied

Node: capacity 12, type Occupied

Node: capacity 8, type Hole

Node: capacity 20, type Occupied

Node: capacity 12, type Occupied

……..

Node: capacity 12, type Occupied

Node: capacity 8, type Hole

Node: capacity 20, type Occupied

Node: capacity 12, type Occupied

Node: capacity 8, type Hole

Node: capacity 20, type Occupied

Node: capacity 12, type Occupied

Node: capacity 8, type Hole

Node: capacity 20, type Occupied

Node: capacity 6000, type Hole

Occupied memory 8000

Free memory 8000

Fourth test: Allocate and free 1500 20 bytes arrays:

N2 allocator fourth test:

Allocation :284 microseconds

Deallocation :581 microseconds

List allocator fourth test:

Allocation :33756 microseconds

Deallocation :809 microseconds

Результаты тестов

Как видно из вывода, программа была запущена на 5 тестах

- Проверка времени, требуемого для инициализации — очевидно, что алгоритму на списках степени 2 требуется много времени для инициализации заголовков блоков.

- **Аллокация 256 байт 10 раз, освобождение 5 из полученных указателей, аллокация 128 байт 10 раз**. Данный тест показывает, как неэффективно может расходовать память аллокатор, основанный на степенях числа 2. Ему потребовалось примерно в 2 раза больше памяти, чем аллокатору, ищущему наилучшее совпадение. Причина этого в том, что такой аллокатор может выделять память только блоками определенного размера и не может разделять их на более мелкие. Второй же аллокатор выделил ровно столько памяти, сколько было нужно.

- **Аллокация и удаление 750 раз по 20 байт.** Данный тест призван сравнить быстродействие аллокаторов. Как видно, аллокатор, основанный на степенях двойки справился значительно быстрее. Это связано с тем, что аллокатору «наилучшее совпадение» приходится при каждой аллокации итерироваться по всему списку в поисках наиболее подходящего сегмента памяти, в то время как у другого аллокатора поиск сегмента происходит практически за константное время. Стоит заметить, что время деаллокации у менеджера памяти, основанного на степенях двойки можно улучшить, если в заголовке каждого узла хранить указатель на список, к которому он принадлежит.

- **Аллокация 500 раз по 20 байт, освобождение каждого второго полученного указателя, аллокация 250 раз по 12 байт.** Данный тест хорошо показывет, как быстро может произойти фрагментация в аллокаторе, основанном на поиске наилучшего сегмента памяти. С другой же стороны, во втором аллокаторе такой проблемы не наблюдается(хотя расход памяти в нем все равно нельзя назвать очень эффективным по причинам, описанным в предыдущих разделах).

- **Аллокация и деаллокация 1500 раз по 20 байт.** Этот тест повторяет второй , но количество запросов к аллокаторам увеличено в два раза. Данный тест показывает, насколько каждый из алгоритмов устойчив к увеличению количества входных данных. Результаты теста таковы:
  При увеличении количества входных данных в два раза:

  Время аллокации и деаллокации на алгоритме «списки степени 2» увеличилось примерно в 2-2.5 раза.

  Время аллокации и деаллокации на алгоритме «наиболее подходящее» увеличилось примерно в 4 раза.

  Очевидно, что алгоритм, основанный на степенях двойки гораздо более устойчив к увеличению числа запросов, что в очередной раз подчеркивает его преимущество над списковым аллокатором.

Заключение

Из сравнения стало ясно, что аллокатор «наилучшее совпадение» имеет гораздо больше недостатков, чем достоинств, и в целом проигрывает «спискам, основанным на степенях двойки» почти по всем критериям — он работает относительно медленно, и неустойчив к увеличению числа запросов(время работы при увеличении числа входных данных в 2 раза увеличилось в 4). Его оппонент достаточно быстр и прост в реализации, не имеет проблемы фрагментации, но не поддерживает слияние и приводит к не очень экономному расходу памяти.

При выполнении данной курсовой работы я познакомился с несколькими видами аллокаторов, а так же более подробно исследовал два из них. Благодаря этой работе я чуть лучше изучил принципы и особенности работы UNIX систем и узнал много нового

Список литературы

- Вахалия. Ю. «UNIX изнутри»
- Таненбаум. Э. «Современные операционные системы»