Московский Авиационный Институт

(Национальный Исследовательский институт)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовая работа по курсу

«Операционные системы»

«Сравнения алгоритмов аллокаторов памяти»

Группа: М8О-208Б-20
Студент: Каширин К.Д.
Преподаватель: Миронов Е.С.
Оценка:_____
Дата:_____

Москва, 2021

Содержание

Постановка задачи

Необходимо сравнить два алгоритма аллокации: списки свободных блоков (первое подходящее) и алгоритм двойников.

Сведения о программе

Программа написана на C++, в среде WSL Ubuntu 18.04.3, сборка произведена посредством make-файла. Программа использует библиотеку STL, в частности контейнеры (vector, list), алгоритмы(find_if). 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. Согласно моему варианту в данной курсовой работы я реализовал следующие алгоритмы аллокации:

- Алгоритм с выбором первого подходящего участка памяти, основанный на списках
- Алгоритм двойников (Buddy Allocation).

Рассмотрим подробнее алгоритмы их реализации и характеристики.

Первый 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 пространство 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** — 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 «первое подходящее». Этот алгоритм работает быстрее, чем «наиболее подходящее» (за счет того, что при каждом запросе не ведется поиск по всему списку). Парадоксально, но его применение даже приводит к менее расточительному использованию памяти, чем использование «наиболее подходящего» или «следующего подходящего».

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.

Алгоритм двойников

В данном алгоритме свободная часть памяти разбивается до тех пор, пока не выйдет блок памяти нужного размера, в каждом блоке есть идентификатор, обозначающий занят или свободен блок. Если освобождается блок и его двойник оказывается свободен, то двойников сливают. Полученный блок пытаются слить с его двойником. Блок, который не удалось слить добавляют в список свободных блоков. Свободные блоки хранятся в двусвязном списке

Реализации аллокаторов

BlocksAllocator.h

```
#include <iostream>
#include <algorithm>
#include <list>
#include <string>
using namespace std;

struct MemoryNode {
    char* begin;
    size_t capacity;
    string type;
};

ostream& operator << (ostream& os, const MemoryNode& node);
class BlocksAllocator {
public:

    explicit BlocksAllocator(size_t size_of_data);
    void* alloc(size_t memory_size);
    void dealloc(void* block);
    void print_memory(ostream& os) const;
    ~BlocksAllocator();

private:
    list<MemoryNode> mem_list;
    char* data;
```

```
};
```

BlocksAllocator.cpp

```
#include "BlocksAllocator.h"
using namespace std;

BlocksAllocator::BlocksAllocator(size_t size_of_data) {
    data = (char *)malloc(size_of_data);
    mem_list.push_front({data, size_of_data, "Freely"});
}

void *BlocksAllocator::alloc(size_t memory_size) {
    if (memory_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 == "Freely" && it->capacity >= memory_size) {
            size_of_node = it->capacity;
            needed_node = it;
        }
    }
    if (size_of_node == 0) {
        throw std::bad_alloc();
    }
    if (memory_size == size_of_node) {
        needed_node->type = "Occupied";
    } else {
        MemoryNode new_node{needed_node->begin + memory_size,
needed_node->capacity - memory_size, "Freely"};
        needed_node->capacity = memory_size;
        needed_node->type = "Occupied";
        mem_list.insert(next(needed_node), new_node);
    }
    return (void *)(needed_node->begin);
}

void BlocksAllocator::dealloc(void *block) {
    auto it = find_if(mem_list.begin(), mem_list.end(), [block](const
MemoryNode &node) {
```

```
        return node.begin == (char *) block && node.type ==
"Occupied";
    });
    if (it == mem_list.end()) {
        cout << "This pointer wasnt allocated by this allocator";
    }
    it->type = "Freely";
    if (it != mem_list.begin() && prev(it)->type == "Freely") {
        auto prev_it = prev(it);
        prev_it->capacity += it->capacity;
        mem_list.erase(it);
        it = prev_it;
    }
    if (next(it) != mem_list.end() && next(it)->type == "Freely") {
        auto next_it = next(it);
        it->capacity += next_it->capacity;
        mem_list.erase(next_it);
    }
}

void BlocksAllocator::print_memory(ostream& os) const {
    int occ_sum = 0, free_sum = 0;
    for (auto& elem : mem_list) {
        os << elem << endl;
        if (elem.type == "Freely") {
            free_sum += elem.capacity;
        } else {
            occ_sum += elem.capacity;
        }
    }
    os << "Occupied memory " << occ_sum << endl;
    os << "Free memory " << free_sum << endl;

}

std::ostream& operator << (std::ostream& os, const MemoryNode& node)
{
    if ("Freely" == node.type) {
        return os << "Node: capacity " << node.capacity << ", type "
<< "Freely";
    } else {
```

```
        return os << "Node: capacity " << node.capacity << ", type "
<< "Occupied";
    }
}

BlocksAllocator::~BlocksAllocator() {
    free(data);
}
```

BuddyAllocator.h

```
#include <exception>
#include <map>
#include <iostream>
using namespace std;

class BuddyAllocator {
public:

    using MapType = multimap<size_t, char*>;
    BuddyAllocator(size_t minBlockSizeArg, size_t maxSizeArg);
    void* alloc(const size_t size);
    void dealloc(void* ptr, size_t size);
    void print_memory(ostream& os) const;
    ~BuddyAllocator();

private:

    size_t Rounder(const size_t size);
    size_t minBlockSize, maxSize;
    MapType allocMap;
    char *allocatedBlocks;

};
```

BuddyAllocator.cpp

```
#include "BuddyAllocator.h"
using MapType = multimap<size_t, char*>;
BuddyAllocator::BuddyAllocator(size_t minBlockSizeArg, size_t
maxSizeArg):minBlockSize(minBlockSizeArg),maxSize(maxSizeArg) {
    allocatedBlocks = static_cast<char*>(::operator new(maxSize));
    allocMap.insert({maxSize, allocatedBlocks});
}
```

```
void *BuddyAllocator::alloc(const size_t size) {

    if (size == 0) {
        return nullptr;
    }
    size_t roundUp = Rounder(size);

    for (MapType::iterator it = allocMap.begin(); it !=
allocMap.end(); ++it) {
        if (it->first == size) { // Found the block of appropriate
size
            void* result {
                it->second
            };
            allocMap.erase(it);
            return result;
        } else if (it->first > size) { // Map stores block sizes in
ascending order
            int allocSize = it->first;
            char *blockStart = it->second;

            while (allocSize > size) {
                allocSize /= 2;
                if (allocSize < size) {
                    break;
                }
                allocMap.insert(pair<size_t, char*> (allocSize,
blockStart + allocSize));
            }
            allocMap.erase(it);
            return blockStart;
        }
    }
    throw std::bad_alloc();
}

void BuddyAllocator::dealloc(void* ptr, size_t size) {
    size_t bufferedSize = size;
    while(true) {
        pair<MapType::iterator, MapType::iterator> range
(allocMap.equal_range(bufferedSize));
```

```
        if (range.first != allocMap.end() && range.first->first ==
bufferedSize) { // Free blocks of same size found
            auto it = range.first;
            short index = ((char*)ptr - allocatedBlocks) /
bufferedSize % 2;
            // If index is 1 then element is odd, else even

            while (it != range.second) {
                    // If difference between the two pointers equals
to size,
                    // then we can merge them into one, since they
are both free
                if (index == 1) {
                    if (((char*)ptr - it->second) == bufferedSize) {
                        bufferedSize *= 2;
                        ptr = it->second;
                        allocMap.erase(it);
                        break; // Element found, break of while loop
                    }
                } else {
                    if ((it->second - (char*)ptr) == bufferedSize) {
                        bufferedSize *= 2;
                        allocMap.erase(it);
                        break; // Element found, break of while loop
                    }
                }
                ++it;
            }
            if (it == range.second) { // No elements found in this
size range
                allocMap.insert(std::pair<size_t, char*>
(bufferedSize, (char*)ptr));
                break;
            } else {
                continue;
            }
        } // No elements of this size found; break out
        else {
            allocMap.insert(std::pair<size_t, char*> (bufferedSize,
(char*)ptr));
            break;
        }
```

```
    }
    return;
}

void BuddyAllocator::print_memory(ostream& os) const {
    int occ_mem = 0;
    for (auto& p: allocMap) {
        occ_mem += p.first;
    }
    int free_mem = maxSize - occ_mem;
    os << "Occupied memory " << occ_mem << "\n" << "Free memory " <<
free_mem  << "\n\n";
}

BuddyAllocator::~BuddyAllocator() {
    ::operator delete(allocatedBlocks);
    allocMap.clear();
}

size_t BuddyAllocator::Rounder(const size_t size) {
        if (size <= minBlockSize) return minBlockSize;

        size_t roundUp = 1;
        while (roundUp < size) {
            roundUp <<= 1;
        }
        return roundUp;
}
```

main.cpp

```
#include <iostream>
#include "BuddyAllocator.h"
#include <chrono>
#include "BlocksAllocator.h"
#include <vector>
using namespace std;
int main() {
    using namespace std::chrono;
    {
        steady_clock::time_point list_allocator_init_start =
steady_clock::now();
        BlocksAllocator list_allocator(4096);
```

```
        steady_clock::time_point list_allocator_init_end =
steady_clock::now();
        cerr << "List allocator initialization with one page of
memory :"
                  << chrono::duration_cast<std::chrono::nanoseconds>(
                          list_allocator_init_end -
list_allocator_init_start).count()
                  << " ns" << endl;


        steady_clock::time_point buddy_allocator_init_start =
steady_clock::now();
        BuddyAllocator bAlloc(32, 4096);
        steady_clock::time_point buddy_allocator_init_end =
steady_clock::now();
        cerr << "Buddy allocator initialization with one page of
memory :"
                  << chrono::duration_cast<std::chrono::nanoseconds>(
                  buddy_allocator_init_end -
buddy_allocator_init_start).count()
                  << " ns" << endl;
        cerr << endl;
    }
    cerr << "First test: Allocate 10 char[256] arrays, free 5 of
them, allocate 10 char[128] arrays:" << endl;
    {
        BlocksAllocator allocator(4096);
        vector<char *> pointers(15, nullptr);
        steady_clock::time_point test1_start = steady_clock::now();
        for (int i = 0; i < 10; ++i) {
            pointers[i] = (char *) allocator.alloc(256);
        }
        for (int i = 5; i < 10; ++i) {
            allocator.dealloc(pointers[i]);
        }
        for (int i = 5; i < 15; ++i) {
            pointers[i] = (char *) allocator.alloc(128);
        }
        steady_clock::time_point test1_end = steady_clock::now();
        cerr << "List allocator first test: "
             <<
std::chrono::duration_cast<chrono::microseconds>(test1_end -
test1_start).count()
```

```
             << " microseconds" << endl;
        allocator.print_memory(cerr);
        for (int i = 0; i < 15; ++i) {
            allocator.dealloc(pointers[i]);
        }
    }
    {
        BuddyAllocator bAlloc(16, 4096);
        vector<char *> pointer(15, nullptr);
        steady_clock::time_point buddy_test1_start =
steady_clock::now();
        for (int i = 0; i < 10; ++i) {
            pointer[i] = (char *) bAlloc.alloc(256);
        }
        for (int i = 5; i < 10; ++i) {
            bAlloc.dealloc(pointer[i], 256);
        }
        for (int i = 5; i < 15; ++i) {
            pointer[i] = (char *) bAlloc.alloc(128);
        }
        steady_clock::time_point buddy_test1_end =
steady_clock::now();
        cerr << "Buddy allocator first test: "
                <<
chrono::duration_cast<std::chrono::microseconds>(buddy_test1_end -
buddy_test1_start).count()
                << " microseconds" << std::endl;
        for (int i = 0; i < 5; ++i) {
            bAlloc.dealloc(pointer[i], 256);
        }
        bAlloc.print_memory(cerr);
        for (int i = 5; i<15; ++i) {
            bAlloc.dealloc(pointer[i], 128);
        }
    }
    cerr << "Second test: Allocate and free 750 20 bytes arrays:\n";
    {
        BlocksAllocator allocator(16000);
        std::vector<char *> pointers(900, nullptr);
        int arr_size = 10;
        steady_clock::time_point alloc_start = steady_clock::now();
        for (int i = 0; i < 900; ++i) {
```

```
            pointers[i] = (char *) allocator.alloc(arr_size);
        }
        steady_clock::time_point alloc_end = steady_clock::now();
        for (int i = 0; i < 900; ++i) {
            allocator.dealloc(pointers[i]);
        }
        cerr << "List allocator second test:" << endl
                  << "Allocation: " <<
duration_cast<chrono::microseconds>(alloc_end - alloc_start).count()
                  << " microseconds" << endl << endl;
    }
    {
        BuddyAllocator allocator(2, 16000);
        vector<char *> pointers(900, nullptr);
        steady_clock::time_point alloc_start = steady_clock::now();
        int arr_size = 10;
        for (int i = 0; i < 900; ++i) {
            pointers[i] = (char *) allocator.alloc(arr_size);
        }
        steady_clock::time_point alloc_end = steady_clock::now();
        for (int i = 0; i < 900; ++i) {
            allocator.dealloc(pointers[i], arr_size);
        }
        steady_clock::time_point test_end = steady_clock::now();
        cerr << "Buddy allocator second test:" << endl
                  << "Allocation :" <<
duration_cast<chrono::microseconds>(alloc_end - alloc_start).count()
                  << " microseconds" << endl;
    }
}
```

Тестирование

Будем тестировать следующие характеристики:

- Скорость выделения и освобождения блоков
- Фрагментацию
- Экономичность

Демонстрация работы программы

```
kirill@LAPTOP-F153AKTP:~/OS/kp$ g++ BuddyAllocator.cpp BlocksAllocator.cpp main.cpp
kirill@LAPTOP-F153AKTP:~/OS/kp$ ./a.out
List allocator initialization with one page of memory :7404 ns
Buddy allocator initialization with one page of memory :6632 ns

First test: Allocate 10 char[256] arrays, free 5 of them, allocate 10 char[128] arrays:
List allocator first test: 24 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 Freely
Occupied memory 2560
Free memory 1536
Buddy allocator first test: 26 microseconds
Occupied memory 2816
Free memory 1280

Second test: Allocate and free 750 20 bytes arrays:
List allocator second test:
Allocation: 18143 microseconds

Buddy allocator second test:
Allocation :554 microseconds
```

Результаты тестов

Как видно из вывода, программа была запущена на 3 тестах

- Проверка времени, требуемого для инициализации —алгоритму на списках требуется больше времени для инициализации заголовков блоков, однако разница с алгоритмом двойников несущественная.

- **Аллокация 256 байт 10 раз, освобождение 5 из полученных указателей, аллокация 128 байт 10 раз**. Оба алгоритма хорошо справились с этим тестом, однако алгоритм списка блоков был чуть эффективнее как по времени, так и по занимаемой памяти.
- **Аллокация и удаление 750 раз по 20 байт.** Данный тест призван сравнить быстродействие аллокаторов. Как видно, аллокатор, основанный на алгоритме двойников справился значительно быстрее. Это связано с тем, что аллокатору «первое подходящее» 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 законстантное время.

  Очевидно, что алгоритм, основанный на двойниках гораздо более устойчив к увеличению числа запросов, что в очередной раз подчеркиваетего преимущество над списковым аллокатором.

Заключение

Из сравнения стало ясно, что аллокатор «первое подхоядщее» имеет гораздо больше недостатков, чем достоинств, и в целом проигрывает «спискам, основанным на двойниках» 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.