Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики
«Кафедра вычислительной математики и программирования»

Курсовая работа по предмету «Операционные системы»

Студент: Брюханов З. Д.
Преподаватель: Миронов Е. С.
Группа: М8О-207Б-22
Дата: 09.12.2023
Оценка:
Подпись:

Москва 2023 г.

Оглавление

Постановка задачи

Задание: Необходимо релаизовать 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 19: сравнить алгоритм аллокации на списках свободных блоков и на блоках по 2 в степени n.

Общая информация о аллокаторах

Аллокатор — специализированный класс, реализующий и инкапсулирующий малозначимые (с прикладной точки зрения) детали распределения и освобождения ресурсов компьютерной памяти.

Стандартные функции *malloc* и *free* на самом деле имеют много проблем:

- Выделение нескольких байтов с помощью *malloc* происходит точно так же, как и выделение нескольких мегабайтов. В расчёт не берётся информация о том, что это за данные, где они будут располагаться и какой у них будет цикл жизни.
- Выделение памяти при помощи стандартных библиотечных функций или операторов обычно требует обращений к ядру операционной системы. Это может сказываться на производительности приложения.
- Они приводят к фрагментации кучи — состоянию, при котором информация в памяти разбросана в разных, не идущих последовательно блоках, из-за чего даже при достаточном суммарном объёме памяти возможна такая ситуация, что выделить блок в памяти для размещения информации будет невозможно.
- Плохая локальность указателей. Нет никакого способа узнать, какое именно место в память выделит вам *malloc*. Это может привести к тому, что будет происходить больше дорогостоящих промахов в кеше.

Аллокатор на списках свободных блоков

Суть аллокатора на списках свободных блоков заключается в том, что при каждом запросе на выделение памяти из доступной аллокатору памяти выделяется блока запрашиваемого размера (если это возможно). В конечном итоге вся память аллокатора будет разделена на список подряд идущих блоков, некоторые из которых будут заняты, а некоторые свободны.

Вся память аллокатора разделена на блоки, каждый из которых содержит заголовок. В заголовке представлена информация о размере этого блока, размере предыдущего блока и логическая переменная, показывающая свободен ли блок. При создании такого аллокатора сразу выделяется память запрашиваемого размера, в которой создаётся один блок. При запросе на выделение памяти определённого размера ищется первый подходящий свободный блок, из которого выделяется блок запрашиваемого размера. При деаллокации ранее выделенной памяти сначала проверяется валидность переданного указателя — если пользователь хочет его вернуть аллокатору, значит этот указатель должен был быть когда-то этим же аллокатором выдан. После в заголовке меняем информацию о том, что блок доступен для использования.

Важной деталью этого аллокатора является возможность дефрагментации. При деаллокации определённого блока памяти происходит проверка

доступности соседних блоков — если какой-то из этих блоков доступен, то происходит слияние текущего блока со свободным соседним.

---

free_blocks_allocator.cpp

```
#pragma once

#include "allocator.h"

class FreeBlocksAllocator : public Allocator {
public:

    explicit FreeBlocksAllocator(size_type size) {
        if ((startPointer = malloc(size)) == nullptr) {
            std::cerr << "Failed to allocate memory\n";
            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::cerr << "Size must be bigger than 0\n";
            return nullptr;
        }
        if (size > totalSize - usedSize) { return nullptr; }
        auto* header = find(size);
        if (header == nullptr) { 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;
        defragmentation(header);
    };
};
```

Тестирование программы.

```
int main() {
    auto allocator = FreeBlocksAllocator(1024);
    allocator.memoryDump();
    auto ptr1 = allocator.allocate(200);
    allocator.memoryDump();
    auto ptr2 = allocator.allocate(100);
    allocator.memoryDump();
    auto ptr3 = allocator.allocate(300);
    allocator.memoryDump();
    allocator.deallocate(ptr2);
    allocator.memoryDump();
    allocator.deallocate(ptr1);
    allocator.memoryDump();
    allocator.deallocate(ptr3);
    allocator.memoryDump();
}
```

Результат работы.

```
Total size : 1024
Used: 24
Header size : 24
+ 0x559b8ea269c0 1000

Total size : 1024
Used: 248
Header size : 24
− 0x559b8ea269c0 200
+ 0x559b8ea26aa0 776

Total size: 1024
Used: 372
Header size: 24
− 0x559b8ea269c0 200
− 0x559b8ea26aa0 100
+ 0x559b8ea26b1c 652

Total size: 1024
```

```
Used: 696
Header size: 24
− 0x559b8ea269c0 200
− 0x559b8ea26aa0 100
− 0x559b8ea26b1c 300
+ 0x559b8ea26c60 328

Total size: 1024
Used: 596
Header size: 24
− 0x559b8ea269c0 200
+ 0x559b8ea26aa0 100
− 0x559b8ea26b1c 300
+ 0x559b8ea26c60 328

Total size: 1024
Used: 372
Header size: 24
+ 0x559b8ea269c0 324
− 0x559b8ea26b1c 300
+ 0x559b8ea26c60 328

Total size: 1024
Used: 24
Header size: 24
+ 0x559b8ea269c0 1000

Process finished with exit code 0
```

Аллокатор на блоках размером 2^n

Принцип работы аллокатора на блоках размером 2^n аналогичен вышеописанному аллокатору на списках свободных блоков за тем исключением, что при создании аллокатора или при запросе на выделение памяти размер блоков выравнивается до ближайшей большей степени числа 2. Таким образом все используемые в данный момент блоки обязательно будут иметь размер 2^n. Такой алгоритм довольно близок к тому, что используется в стандартном системном аллокаторе, ведь он тоже оперирует с блоками, размер которых является степенью числа 2.

---

binary_allocator.cpp

```
#pragma once
```

```
#include <cmath>
#include "allocator.h"

class BinaryAllocator : public Allocator {
public:

    explicit BinaryAllocator(size_type size) {
        size = align(size);
        if ((startPointer = malloc(size)) == nullptr) {
            std::cerr << "Failed to allocate memory\n";
            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;
    };

    static size_type align(size_type size) {
        int i = 0;
        while (pow(2, i) < size) {
            i++;
        }
        return (size_type) pow(2, i);
    }

    pointer allocate(size_type size) override {
        if (size <= 0) {
            std::cerr << "Size must be bigger than 0\n";
            return nullptr;
        }
        size = align(size);
        if (size > totalSize - usedSize) { return nullptr; }
        auto* header = find(size);
        if (header == nullptr) { 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;
        defragmentation(header);
    };
};
```

Тестирование программы.

```
int main() {
    auto allocator = BinaryAllocator(1024);
    allocator.memoryDump();
    auto ptr1 = allocator.allocate(200);
    allocator.memoryDump();
    auto ptr2 = allocator.allocate(100);
    allocator.memoryDump();
    auto ptr3 = allocator.allocate(300);
    allocator.memoryDump();
    allocator.deallocate(ptr2);
    allocator.memoryDump();
    allocator.deallocate(ptr1);
    allocator.memoryDump();
    allocator.deallocate(ptr3);
    allocator.memoryDump();
}
```

Результат работы.

```
Total size : 1024
Used: 24
Header size : 24
+ 0x55c4a420d9c0 1000

Total size : 1024
Used: 304
Header size : 24
− 0x55c4a420d9c0 256
+ 0x55c4a420dad8 720

Total size: 1024
Used: 456
Header size: 24
```

− 0x55c4a420d9c0 256
− 0x55c4a420dad8 128
+ 0x55c4a420db70 568

Total size: 1024
Used: 992
Header size: 24
− 0x55c4a420d9c0 256
− 0x55c4a420dad8 128
− 0x55c4a420db70 512
+ 0x55c4a420dd88 32

Total size: 1024
Used: 864
Header size: 24
− 0x55c4a420d9c0 256
+ 0x55c4a420dad8 128
− 0x55c4a420db70 512
+ 0x55c4a420dd88 32

Total size: 1024
Used: 584
Header size: 24
+ 0x55c4a420d9c0 408
− 0x55c4a420db70 512
+ 0x55c4a420dd88 32

Total size: 1024
Used: 24
Header size: 24
+ 0x55c4a420d9c0 1000

Process finished with exit code 0

Сравнение алгоритмов аллокации

Для сравнения алгоритмов аллокации будем замерять общее время работы аллокатора по аллокации/деаллокации блоков памяти. Для того, чтобы значения были более наглядными, тестовый файл содержит 50000 запросов на аллокацию/деаллокацию. На основе тестирования были получены следующие показатели:

- Аллокатор на списке свободных блоков с дефрагментацией: 196.185 мс, 184.035 мс, 193.589 мс.

- Аллокатор на списке свободных блоков без дефрагментации: 317.919 мс, 320.539 мс, 325.483 мс.
- Аллокатор на блоках по 2^n с дефрагментацией: 155.929 мс, 163.766 мс, 166.549 мс.
- Аллокатор на блоках по 2^n без дефрагментации: 129.728 мс, 126.534 мс, 132.30 мс.

Вывод

По результатам тестирования видно, что самый худший результат показывает алгоритм аллокации на списках свободных блоков без дефрагментации, а самый лучший - аллокатор на блоках по 2^n, причём так же без дефрагментации. Причиной этого является то, что процессор и вся система памяти изначально настроена на работу в бинарной системе, то есть с блоками памяти, размер которых является степенью числа 2. При аллокации блока памяти, размер которого отличается от степени 2, система всё равно приводит его к такому виду. В случае с аллокатором на блоках по 2^n мы сразу работаем с "более удобными"для процессора блоками. Причина того, почему лучше всего работает алгоритм без дефрагментации также проста — при выбранной реализации аллокатора в общем случае невозможно провести дефрагментацию двух соседних блоков памяти так, чтобы размер полученного блока памяти всё равно был степенью числа 2 из-за того, что вместе с каждым блоком памяти хранится его заголовок, который при слиянии становится доступной памятью. То есть при слиянии двух блоков размер полученного блока равен $s=2^{n_1}+2^{n_2}+24$, где 2^{n_1} и 2^{n_2} - размеры соседних блоков в байтах, а 24 — размер заголовка. В общем случае s не будет являться степенью числа 2 (хоть это и возможно, например, если $n_1 = n_2 = 2$, так как в таком случае $4+4+24 = 32 = 2^5$).