Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

**Курсовой проект по курсу**

**«Операционные системы»**

Группа: М8О-215Б-23

Студент: Шаталов М.А.

Преподаватель: Миронов Е.С.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 26.12.24

Москва, 2024

**Постановка задачи**

**Вариант 15.**

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

* Фактор использования
* Скорость выделения блоков
* Скорость освобождения блоков
* Простота использования аллокатора

Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям free и malloc (realloc, опционально).  Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра. Необходимо самостоятельно разработать стратегию тестирования для определения ключевых характеристик аллокаторов памяти. При тестировании нужно свести к минимуму потери точности из-за накладных расходов при измерении ключевых характеристик, описанных выше.

**Общий метод и алгоритм решения**

Использованные системные вызовы:

* void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);– выделяет страницу памяти.
* int munmap(void \*addr, size\_t length); освобождения памяти, ранее отображенной с помощью mmap

Цель данной лабораторной работы сравнить две разные стратегии выделения памяти на примере написания двух аллокаторов: аллокатора, использующего список свободных блоков, и аллокатора, использующего метод двойников. Рассмотрим подробнее каждый из алгоритмом, а потом сравним их на различных тестах.

Разберем аллокатор, использующий списки свободных блоков. Он представляет собой метод управления памятью, который использует список свободных блоков памяти для выделения памяти по запросам.

Основные принципы работы

1. Инициализация и хранение внутренней структуры: при запуске программы мы выделяем системными средствами ядра некоторый объем памяти, который передаем аллокатору. Аллокатор должен помещаться ровно в выделенной памяти, поэтому вся структура блоков будет храниться внутри буфера. Список блоков будет односвязным, так что в каждом блоке будем хранить: размер текущего блока, ссылку на следующий блок и флаг, указывающий на статус блока(занят/свободен).
2. Запрос памяти(alloc): когда программа запрашивает память определенного размера, аллокатор начинает проходить по списку блоков. Поиск подходящего блока осуществляется последовательно, начиная с первого элемента списка. Если найденный блок точно соответствует запрошенному размеру, он выделяется и помечается, как занятый, при этому пользователю возвращается адрес указателя на блок + размер блока. Если найденный блок больше запрошенного размера, он может быть разделен на два блока: один для удовлетворения запроса, а другой (оставшийся) остается в списке свободных блоков. Этот процесс называется разбиением блока.
3. Освобождение памяти и объединение блоков: когда память больше не нужна, она снова становится свободной. Для уменьшения внутренней фрагментации, аллокатор объединяет смежные свободные блоки. Этот процесс называется объединением блоков. Объединение блоков осуществляется путем проверки соседних блоков в списке свободных блоков. Если соседние блоки также свободны, они объединяются в один больший блок. Это позволяет уменьшить количество мелких блоков и увеличить вероятность нахождения подходящего блока при следующем запросе памяти.

Благодаря такому алгоритму аллокатор эффективно выделяет память и скорость работы зависит не от размера запрашиваемой памяти, а от длины списка.

Теперь разберем работу аллокатора, основанного на методе близнецов. Скорость выделения памяти можно существенно повысить, если выделять ее блоками заранее определенного размера, при этом может быть одновременно использовано несколько альтернативных размеров блока. Скорость выделения памяти возрастает потому, что не требуется искать в списке подходящий по размеру блок. Такой прием использован алгоритме близнецов, обеспечивающем очень быстрое выделение и освобождение памяти блоками кратного размера.

Сущность алгоритма близнецов состоит в следующем. Размер кучи выбирается кратным целой степени двойки. Положим, что размер кучи равен 2N. В ответ на запрос о выделении памяти, может быть выделен блок памяти размером ,2N/2 = 2N-1, 2N-2 и т.д., вплоть до минимального блока размером 2N-M (всего предусматривается M+1 возможных размеров блоков, причем M<=N ).

В алгоритме близнецов блок меньшего размера получают за счет деления пополам одного более крупного блока (см. рис. 1). Пара блоков, полученная делением на две части одного более крупного блока, называется близнецами или двойниками.

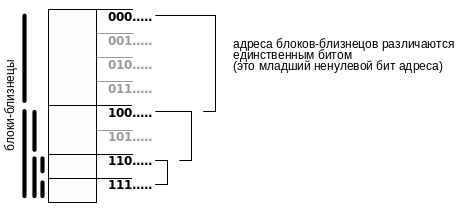
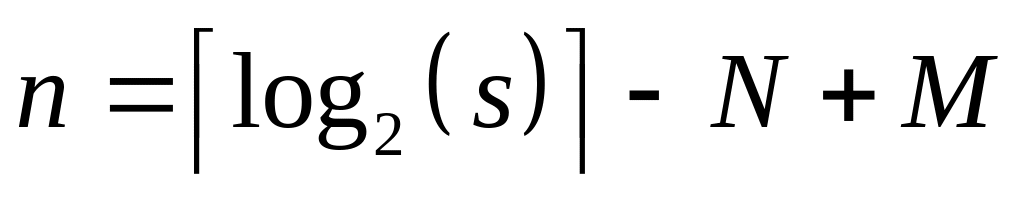


Рис.1. Формирование блоков-двойников

Заметим, что адреса блоков-близнецов отличаются только одним битом (000 и 100, 100 и 110, 100 и 101, 110 и 111, и т.д.). Этот факт будет использоваться в алгоритме близнецов при объединении освобождаемых блоков.

В соответствии с алгоритмом близнецов, адреса всех свободных блоков размещаются в очередях, причем для блоков каждого размера предусмотрена собственная очередь. Пронумеруем эти очереди от до в направлении возрастания размеров свободных блоков. Свяжем блоки, записывая в начале каждого свободного блока адрес следующего свободного блока того же размера. По мере динамического выделения памяти, свободные блоки меньших размеров распределяются по ассоциированным с ними очередям.

1. Запрос на выделение памяти. Когда поступает запрос на выделение блока памяти размером S , где 2N, прежде всего, находят очередь, содержащую блоки, лучше всего подходящие по размеру. Требуемую очередь легко найти по номеру, используя следующую формулу:  
     
     
   Если найденная очередь (очередь номер ) не пустая, то из ее головы извлекается адрес свободного блока и передается в ответ на запрос о выделении памяти. Если же очередь номер окажется пустой, то обращаются к следующей очереди (номер n+1), и так далее, пока не будет найдена не пустая очередь. Предположим, что очередь не пустая. Тогда из нее извлекается один свободный блок и разбивается на два блока половинного размера – близнецов. Адрес одного из блоков-близнецов заносится в очередь, а адрес второго блока-близнеца передается в ответ на запрос о выделении памяти.
2. Освобождение памяти. Рассмотрим теперь процедуру освобождения блока. Для работы алгоритма двойников необходимо, чтобы размер освобождаемого блока был известен, поэтому храним его в заголовке блока. По размеру блока определяется номер очереди свободных блоков, которая могла бы содержать двойник освобождаемого блока. Эта очередь просматривается, и адреса всех свободных блоков из очереди сопоставляются с адресом освобождаемого блока на наличие совпадения старших битов. Если при проверке условие выполнилось, то проверяемый блок является двойником для освобождаемого блока. Проверяемый блок извлекается из очереди свободных блоков и объединяется с освобождаемым блоком. Адресом объединенного блока является меньший из адресов объединяемых блоков. После объединения блоков, предпринимается новая попытка найти двойника уже для объединенного блока в следующей очереди и т.д. по рекурсии, пока либо все блоки не будут объединены в единственный, либо при проверке очереди в ней не будет найден двойник (или очередная очередь окажется пустой).

**Код программы**

**main.cpp**

#include <First\_fit.h>

#include <Buddy.h>

#include <sys/mman.h>

#include <chrono>

#include <vector>

#include <thread>

#include <random>

#define NOW std::chrono::high\_resolution\_clock::now()

template <typename T>

void print(T st)

{

    std::cout << st << '\n';

}

int main()

{

    print("Creating allocators..");

    size\_t size = 1024 \* 1024 \* 1024;

    //MAP\_PRIVATE | MAP\_ANONYMOUS: Память анонимная и изменения видны только текущему процессу

    void \*mem1 = mmap(NULL, size, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0);

    void \*mem2 = mmap(NULL, size, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0);

    Allocator\_FirstFit al1(mem1, size);

    Allocator\_Buddy al2(mem2, size);

    std::cout << "All memory: " << al1.get\_free\_memory() << ' ' << al2.get\_free\_memory() << '\n';

    print("Created");

    print("TEST 1 -- allocate lots of memory of a single little size");

    int cycl\_count = 5;

    std::cout << "Cycles count = " << cycl\_count << "\n\n";

    int blocks\_count = 10000;

    int t1\_sz[cycl\_count] = {1, 4, 8, 16, 32, 64};

    for (int i = 0; i < cycl\_count; i++)

    {

        std::cout << "Block size = " << t1\_sz[i] << " Block count = " << blocks\_count << std::endl;

        print("Testing...");

        /////////////allocate

        std::vector<void \*> ptrs1(blocks\_count);

        auto start = std::chrono::high\_resolution\_clock::now();

        for (int j = 0; j < blocks\_count; j++)

        {

            ptrs1[j] = al1.alloc(1 << t1\_sz[i]);

        }

        auto end = std::chrono::high\_resolution\_clock::now();

        auto duration1 = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

        std::vector<void \*> ptrs2(blocks\_count);

        start = std::chrono::high\_resolution\_clock::now();

        for (int j = 0; j < blocks\_count; j++)

        {

            ptrs2[j] = al2.alloc(1 << t1\_sz[i]);

        }

        end = std::chrono::high\_resolution\_clock::now();

        auto duration2 = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

        /////////////free

        start = std::chrono::high\_resolution\_clock::now();

        for (int j = 0; j < blocks\_count; j++)

        {

            al1.free(ptrs1[j]);

        }

        end = std::chrono::high\_resolution\_clock::now();

        auto duration3 = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

        start = std::chrono::high\_resolution\_clock::now();

        for (int j = 0; j < blocks\_count; j++)

        {

            al2.free(ptrs2[j]);

        }

        end = std::chrono::high\_resolution\_clock::now();

        auto duration4 = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

        std::cout << "Result Allocate:\nFirst-fit: " << duration1 << " Buddy:" << duration2 << " " << float(duration1.count()) / float(duration2.count()) << "\n";

        std::cout << "Result Free:\nFirst-fit: " << duration3 << " Buddy:" << duration4 << " " << float(duration3.count()) / float(duration4.count()) << "\n\n";

    }

    {

        print("\nTEST 2 -- allocate a lot of memory in the face of high fragmentation");

        print("Allocate 5000 blocks size 100...");

        int blocks\_count = 10000;

        std::vector<void \*> frament\_blocks1(blocks\_count);

        for (int i = 0; i < blocks\_count; ++i)

        {

            frament\_blocks1[i] = al1.alloc(100);

        }

        print("Free all even blocks alloc1...");

        for (int i = 0; i < blocks\_count; ++i)

        {

            if (i % 2 == 0)

                al1.free(frament\_blocks1[i]);

        }

        std::vector<void \*> frament\_blocks2(blocks\_count);

        for (int i = 0; i < blocks\_count; ++i)

        {

            frament\_blocks2[i] = al2.alloc(100);

        }

        print("Free all even blocks alloc2...");

        for (int i = 0; i < blocks\_count; ++i)

        {

            if (i % 2 == 0)

                al2.free(frament\_blocks2[i]);

        }

        print("Fragmentation completed");

        std::cout << "Block size = " << 80 << " Block count = " << blocks\_count / 2 << std::endl;

        print("Testing...");

        /////////////allocate

        std::vector<void \*> ptrs1(blocks\_count / 2);

        auto start = std::chrono::high\_resolution\_clock::now();

        for (int j = 0; j < blocks\_count / 2; j++)

        {

            ptrs1[j] = al1.alloc(80);

        }

        auto end = std::chrono::high\_resolution\_clock::now();

        auto duration1 = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

        std::vector<void \*> ptrs2(blocks\_count / 2);

        start = std::chrono::high\_resolution\_clock::now();

        for (int j = 0; j < blocks\_count / 2; j++)

        {

            ptrs2[j] = al2.alloc(80);

        }

        end = std::chrono::high\_resolution\_clock::now();

        auto duration2 = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

        /////////////free

        start = std::chrono::high\_resolution\_clock::now();

        for (int j = 0; j < blocks\_count / 2; j++)

        {

            al1.free(ptrs1[j]);

        }

        end = std::chrono::high\_resolution\_clock::now();

        auto duration3 = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

        start = std::chrono::high\_resolution\_clock::now();

        for (int j = 0; j < blocks\_count / 2; j++)

        {

            al2.free(ptrs2[j]);

        }

        end = std::chrono::high\_resolution\_clock::now();

        auto duration4 = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

        std::cout << "Result Allocate:\nFirst-fit: " << duration1 << " Buddy:" << duration2 << " " << float(duration1.count()) / float(duration2.count()) << "\n";

        std::cout << "Result Free:\nFirst-fit: " << duration3 << " Buddy:" << duration4 << " " << float(duration3.count()) / float(duration4.count()) << "\n\n";

        for (int i = 0; i < blocks\_count; ++i)

        {

            if (i % 2 != 0) // Free only the odd blocks, as even blocks were already freed

            {

                al1.free(frament\_blocks1[i]);

                al2.free(frament\_blocks2[i]);

            }

        }

    }

    {

        print("\nTEST 3 -- allocate random blocks");

        int num\_iterations = 5;

        int num\_blocks = 10000; // Use a lot of blocks

        std::random\_device rd;

        std::mt19937 gen(rd());

        std::uniform\_int\_distribution<> distrib(4, 1024); // Random block sizes up to 1KB

        for (int iter = 0; iter < num\_iterations; ++iter)

        {

            std::vector<size\_t> sizes(num\_blocks);

            std::vector<void \*> ptrs1(num\_blocks);

            std::vector<void \*> ptrs2(num\_blocks);

            // Generate random block sizes

            for (int i = 0; i < num\_blocks; ++i)

            {

                sizes[i] = distrib(gen);

            }

            auto start = std::chrono::high\_resolution\_clock::now();

            for (int i = 0; i < num\_blocks; ++i)

            {

                ptrs1[i] = al1.alloc(sizes[i]);

            }

            auto end = std::chrono::high\_resolution\_clock::now();

            auto duration1 = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

            start = std::chrono::high\_resolution\_clock::now();

            for (int i = 0; i < num\_blocks; ++i)

            {

                ptrs2[i] = al2.alloc(sizes[i]);

            }

            end = std::chrono::high\_resolution\_clock::now();

            auto duration2 = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

            start = std::chrono::high\_resolution\_clock::now();

            for (int i = 0; i < num\_blocks; ++i)

            {

                al1.free(ptrs1[i]);

            }

            end = std::chrono::high\_resolution\_clock::now();

            auto duration3 = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

            start = std::chrono::high\_resolution\_clock::now();

            for (int i = 0; i < num\_blocks; ++i)

            {

                al2.free(ptrs2[i]);

            }

            end = std::chrono::high\_resolution\_clock::now();

            auto duration4 = std::chrono::duration\_cast<std::chrono::microseconds>(end - start);

            std::cout << "Iteration " << iter + 1 << ":\n";

            std::cout << "Result Allocate:\nFirst-fit: " << duration1 << " Buddy:" << duration2 << " " << (duration1.count() > 0 && duration2.count() > 0 ? float(duration1.count()) / float(duration2.count()) : 0) << "\n";

            std::cout << "Result Free:\nFirst-fit: " << duration3 << " Buddy:" << duration4 << " " << (duration3.count() > 0 && duration4.count() > 0 ? float(duration3.count()) / float(duration4.count()) : 0) << "\n\n";

        }

    }

    {

        print("\nTEST 4 -- Usage factor First-fit");

        const int num\_requests = 10000;    // Количество запросов на выделение памяти

        const size\_t max\_block\_size = 512; // Максимальный размер блока

        std::uniform\_int\_distribution<size\_t> dist(16, max\_block\_size); // Равномерное распределение

        std::random\_device rd;

        std::mt19937 gen(rd());

        size\_t total\_allocated = 0;

        size\_t memory\_size = al1.get\_free\_memory();

        std::cout << "All memory: " << memory\_size << '\n';

        std::vector<void \*> ptrs(num\_requests);

        for (int i = 0; i < num\_requests; ++i)

        {

            size\_t req\_size = dist(gen);

            ptrs[i] = al1.alloc(req\_size);

            total\_allocated += req\_size;

        }

        double utilization\_factor = (double)(total\_allocated) / (memory\_size - al1.get\_free\_memory());

        std::cout << "Total allocated: " << total\_allocated << " bytes" << std::endl;

        std::cout << "Free memory: " << al1.get\_free\_memory() << " bytes" << std::endl;

        std::cout << "Utilization factor: " << utilization\_factor << std::endl;

    }

    {

        print("\nTEST 5 -- Usage factor Buddy");

        const int num\_requests = 10000;    // Количество запросов на выделение памяти

        const size\_t max\_block\_size = 512; // Максимальный размер блока

        std::uniform\_int\_distribution<size\_t> dist(16, max\_block\_size); // Равномерное распределение

        std::random\_device rd;

        std::mt19937 gen(rd());

        size\_t total\_allocated = 0;

        size\_t memory\_size = al2.get\_free\_memory();

        std::cout << "All memory: " << memory\_size << '\n';

        std::vector<void \*> ptrs(num\_requests);

        for (int i = 0; i < num\_requests; ++i)

        {

            size\_t req\_size = dist(gen);

            ptrs[i] = al2.alloc(req\_size);

            total\_allocated += req\_size;

        }

        double utilization\_factor = (double)(total\_allocated) / (memory\_size - al2.get\_free\_memory());

        std::cout << "Total allocated: " << total\_allocated << " bytes" << std::endl;

        std::cout << "Free memory: " << al2.get\_free\_memory() << " bytes" << std::endl;

        std::cout << "Utilization factor: " << utilization\_factor << std::endl;

    }

    // Освобождаем память

    munmap(mem1, size);

    munmap(mem2, size);

}

**First\_fit.h**

#pragma once

#include <iostream>

#include <list>

class Allocator\_FirstFit

{

    struct Block

    {

        size\_t size;

        Block \*next;

        bool free;

    };

private:

    Block \*head; // голова односвязного списка блоков

public:

    Allocator\_FirstFit(void \*realMemory, size\_t memory\_size);

    void \*alloc(size\_t block\_size);

    void free(void \*block);

    size\_t get\_free\_memory();

};

**First\_fit.cpp**

#include <First\_fit.h>

size\_t Allocator\_FirstFit::get\_free\_memory()

{

    size\_t ans=0;

    Block \*cur = head;

    while (cur != nullptr)

    {

        if (cur->free)

            ans += cur->size;

        cur = cur->next;

    }

    return ans;

}

Allocator\_FirstFit::Allocator\_FirstFit(void \*realMemory, size\_t memory\_size)

{

    head = (Block \*)(realMemory);

    head->free = true;

    head->next = nullptr;

    head->size = memory\_size - sizeof(Block);

}

void \*Allocator\_FirstFit::alloc(size\_t block\_size)

{

    Block \*current = head;

    while (current != nullptr)

    {

        if (current->free && current->size >= block\_size)

        {

            if (current->size > block\_size + sizeof(Block))

            {

                Block \*freeBlock = (Block \*)((char \*)current + sizeof(Block) + block\_size);

                freeBlock->size = current->size - block\_size - sizeof(Block);

                freeBlock->next = current->next;

                freeBlock->free = true;

                current->size = block\_size;

                current->next = freeBlock;

                current->free = false;

                return (void \*)(current + 1);

            }

            else

            {

                current->free = false;

                return (void \*)(current + 1);

            }

        }

        current = current->next;

    }

    return nullptr; // у аллокатора закончилась память

}

void Allocator\_FirstFit::free(void \*block)

{

    if (block == nullptr)

        return;

    Block \*current = (Block \*)block - 1;

    current->free = true;

    // Пытаемся объеденить с предыдущим свободным блоком

    Block \*previous = head; // находим предыдущего

    if (current != previous)

    {

        while (previous->next != current)

        {

            previous = previous->next;

        }

        if (previous->free) // пред свободен->объединяем

        {

            previous->size += sizeof(Block) + current->size;

            previous->next = current->next;

            current = previous;

        }

    }

    // Пытаемся объединить со следующим свободным блоком

    if (current->next != nullptr && current->next->free)

    {

        current->size += sizeof(Block) + current->next->size;

        current->next = current->next->next;

    }

}

**Buddy.h**

#pragma once

#include <iostream>

#include <string>

#define MINLEVEL 4

class Allocator\_Buddy

{

    union Block

    {

        int size;

        Block \*next;

    };

private:

    void \*buffer; // буффер

    Block \*\*queues;

    int levels;

    bool is\_bro(Block\* bro1, Block\*bro2, int teck\_level);

    int log2\_top(size\_t x);

    int log2\_down(size\_t x);

public:

    Allocator\_Buddy(void \*realMemory, size\_t memory\_size);

    void \*alloc(size\_t block\_size);

    void free(void \*block);

    void print(std::string st);

    size\_t get\_free\_memory();

};

**Buddy.cpp**

#include <Buddy.h>

int Allocator\_Buddy::log2\_top(size\_t x) // логарифм с округлением вверх

{

    int log2 = 0;

    size\_t x2 = x;

    while (x >>= 1)

    {

        log2++;

    }

    if (x2 != 1 << log2)

    {

        log2++;

    }

    return log2;

}

int Allocator\_Buddy::log2\_down(size\_t x) // логарифм с округлением вниз

{

    int log2 = 0;

    while (x >>= 1)

    {

        log2++;

    }

    return log2;

}

void Allocator\_Buddy::print(std::string st) // принтер очередей

{

    std::cout << "Start print " << st << "\n";

    for (int i = 0; i < levels - MINLEVEL; ++i)

    {

        std::cout << "Que " << i << " - ";

        Block \*cur = queues[i];

        while (cur != nullptr)

        {

            std::cout << size\_t(cur) - size\_t(buffer) << " ";

            cur = cur->next;

        }

        std::cout << std::endl;

    }

}

size\_t Allocator\_Buddy::get\_free\_memory()

{

    long ans = 0;

    for (int i = 0; i < levels - MINLEVEL; ++i)

    {

        Block \*cur = queues[i];

        while (cur != nullptr)

        {

            ans += (1 << (levels - i));

            cur = cur->next;

        }

    }

    return ans;

}

Allocator\_Buddy::Allocator\_Buddy(void \*realMemory, size\_t memory\_size)

{

    levels = log2\_down(memory\_size);

    int q\_mem = sizeof(Block) \* (levels - MINLEVEL + 1);

    levels = log2\_down(memory\_size - q\_mem);

    queues = (Block \*\*)realMemory;

    buffer = (char \*)realMemory + q\_mem;

    queues[0] = (Block \*)buffer;

    queues[0]->next = nullptr;

    for (int i = 1; i <= levels - MINLEVEL; ++i)

        queues[i] = nullptr;

}

void \*Allocator\_Buddy::alloc(size\_t block\_size)

{

    if (levels < log2\_top(block\_size + sizeof(Block))) // если попросили больше чем размер аллокатора

        return nullptr;

    size\_t required\_level = levels - log2\_top(block\_size + sizeof(Block)); // определяем какой уровень нам нужен

    if (required\_level > levels - MINLEVEL)                                // округляем маленький запрос до размера минимального блока в аллокаторе

        required\_level = levels - MINLEVEL;

    if (queues[required\_level] != nullptr) // в нужной очередь есть подходящий блок

    {

        Block \*ans = (Block \*)queues[required\_level];

        queues[required\_level] = queues[required\_level]->next;

        ans->size = required\_level;

        return ans + 1;

    }

    // не повезло, находим минимальный  непустой уровень

    int i = required\_level - 1;

    for (i; i >= 0; --i)

        if (queues[i] != nullptr) // опааа, нашлии очередь с блоком свободным

            break;

    if (i == -1)        // нет свободных блоков

        return nullptr; // памяти нет :(

    while (queues[required\_level] == nullptr) // пока в нужной очереди не появились блоки делим более большие

    {

        Block \*big = queues[i]; // убираем из очереди голову

        queues[i] = queues[i]->next;

        Block \*bro1, \*bro2;

        bro1 = big;

        bro2 = (Block \*)((char \*)big + (1 << (levels - i)) / 2);

        bro1->next = bro2;

        bro2->next = queues[i + 1];

        queues[i + 1] = bro1;

        ++i;

    }

    // вышли из цикла, значит доделили что в нашей очереди есть блоки. Возвращаем его

    Block \*ans = (Block \*)queues[required\_level];

    queues[required\_level] = queues[required\_level]->next;

    ans->size = required\_level;

    return ans + 1;

}

bool Allocator\_Buddy::is\_bro(Block \*bro1, Block \*bro2, int teck\_level)

{

    size\_t tbro1 = size\_t(bro1) - size\_t(buffer);

    size\_t tbro2 = size\_t(bro2) - size\_t(buffer);

    size\_t mask = 1 << (levels - teck\_level);

    mask = ~mask;

    return (tbro1 & mask) == (tbro2 & mask);

}

void Allocator\_Buddy::free(void \*block)

{

    if (block == nullptr)

        return;

    Block \*cur = (Block \*)block - 1; // поняли в каком мы блоке сейчас находимся

    int teck\_level = cur->size;      // нашли текущий уровень

    cur->next = queues[teck\_level];

    queues[teck\_level] = cur; // добавили в очередь свободных блоков нужного размера

    // теперь проходимся вверх и пытаемся соеденить cur с двойником

    bool bro\_find = true;

    while (bro\_find && teck\_level >= 0)

    {

        bro\_find = false;

        cur = queues[teck\_level];

        Block \*prev = queues[teck\_level];

        while (prev != nullptr)

        {

            if (is\_bro(cur, prev->next, teck\_level))

            {

                bro\_find = true;

                Block \*uni = std::min(prev->next, cur);

                prev->next = prev->next->next;

                queues[teck\_level] = cur->next;

                uni->next = queues[teck\_level - 1];

                queues[teck\_level - 1] = uni;

                break;

            }

            prev = prev->next;

        }

        --teck\_level;

    }

}

**Протокол работы программы**

**Тестирование:**

Creating allocators..

All memory: 1073741800 536870912

Created

TEST 1 -- allocate lots of memory of a single little size

Cycles count = 5

Block size = 1 Block count = 10000

Testing...

Result Allocate:

First-fit: 173691µs Buddy:4502µs 38.5809

Result Free:

First-fit: 171µs Buddy:335µs 0.510448

Block size = 4 Block count = 10000

Testing...

Result Allocate:

First-fit: 148672µs Buddy:440µs 337.891

Result Free:

First-fit: 179µs Buddy:342µs 0.523392

Block size = 8 Block count = 10000

Testing...

Result Allocate:

First-fit: 154928µs Buddy:982µs 157.768

Result Free:

First-fit: 171µs Buddy:361µs 0.473684

Block size = 16 Block count = 10000

Testing...

Result Allocate:

First-fit: 1475508µs Buddy:208048µs 7.09215

Result Free:

First-fit: 667µs Buddy:1301µs 0.512683

Block size = 32 Block count = 10000

Testing...

Result Allocate:

First-fit: 192108µs Buddy:339µs 566.69

Result Free:

First-fit: 174µs Buddy:331µs 0.52568

TEST 2 -- allocate a lot of memory in the face of high fragmentation

Allocate 5000 blocks size 100...

Free all even blocks alloc1...

Free all even blocks alloc2...

Fragmentation completed

Block size = 80 Block count = 5000

Testing...

Result Allocate:

First-fit: 89364µs Buddy:181µs 493.724

Result Free:

First-fit: 83506µs Buddy:52921µs 1.57794

TEST 3 -- allocate random blocks

Iteration 1:

Result Allocate:

First-fit: 345135µs Buddy:708µs 487.479

Result Free:

First-fit: 295µs Buddy:1509µs 0.195494

Iteration 2:

Result Allocate:

First-fit: 326799µs Buddy:730µs 447.67

Result Free:

First-fit: 216µs Buddy:758µs 0.28496

Iteration 3:

Result Allocate:

First-fit: 369220µs Buddy:718µs 514.234

Result Free:

First-fit: 179µs Buddy:736µs 0.243207

Iteration 4:

Result Allocate:

First-fit: 345903µs Buddy:730µs 473.84

Result Free:

First-fit: 185µs Buddy:735µs 0.251701

Iteration 5:

Result Allocate:

First-fit: 310672µs Buddy:817µs 380.259

Result Free:

First-fit: 193µs Buddy:764µs 0.252618

TEST 4 -- Usage factor First-fit

All memory: 1073741800

Total allocated: 2637802 bytes

Free memory: 1070863998 bytes

Utilization factor: 0.916603

TEST 5 -- Usage factor Buddy

All memory: 536870912

Total allocated: 2630546 bytes

Free memory: 533204800 bytes

Utilization factor: 0.71753

**Strace:**

execve("./main", ["./main"], 0x7ffd6814d380 /\* 34 vars \*/) = 0

brk(NULL) = 0x555996e83000

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f6d78e48000

access("/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)

openat(AT\_FDCWD, "/etc/ld.so.cache", O\_RDONLY|O\_CLOEXEC) = 3

newfstatat(3, "", {st\_mode=S\_IFREG|0644, st\_size=26550, ...}, AT\_EMPTY\_PATH) = 0

mmap(NULL, 26550, PROT\_READ, MAP\_PRIVATE, 3, 0) = 0x7f6d78e41000

close(3) = 0

openat(AT\_FDCWD, "/usr/local/lib64/libstdc++.so.6", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832

newfstatat(3, "", {st\_mode=S\_IFREG|0755, st\_size=2530008, ...}, AT\_EMPTY\_PATH) = 0

mmap(NULL, 2543808, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f6d78bd3000

mmap(0x7f6d78c78000, 1216512, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0xa5000) = 0x7f6d78c78000

mmap(0x7f6d78da1000, 581632, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1ce000) = 0x7f6d78da1000

mmap(0x7f6d78e2f000, 57344, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x25c000) = 0x7f6d78e2f000

mmap(0x7f6d78e3d000, 12480, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7f6d78e3d000

close(3) = 0

openat(AT\_FDCWD, "/usr/local/lib64/libgcc\_s.so.1", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832

newfstatat(3, "", {st\_mode=S\_IFREG|0644, st\_size=906528, ...}, AT\_EMPTY\_PATH) = 0

mmap(NULL, 181160, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f6d78ba6000

mmap(0x7f6d78baa000, 143360, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x4000) = 0x7f6d78baa000

mmap(0x7f6d78bcd000, 16384, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x27000) = 0x7f6d78bcd000

mmap(0x7f6d78bd1000, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x2b000) = 0x7f6d78bd1000

close(3) = 0

openat(AT\_FDCWD, "/lib/x86\_64-linux-gnu/libc.so.6", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20t\2\0\0\0\0\0"..., 832) = 832

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

newfstatat(3, "", {st\_mode=S\_IFREG|0755, st\_size=1922136, ...}, AT\_EMPTY\_PATH) = 0

pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784

mmap(NULL, 1970000, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f6d789c5000

mmap(0x7f6d789eb000, 1396736, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x26000) = 0x7f6d789eb000

mmap(0x7f6d78b40000, 339968, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x17b000) = 0x7f6d78b40000

mmap(0x7f6d78b93000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1ce000) = 0x7f6d78b93000

mmap(0x7f6d78b99000, 53072, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7f6d78b99000

close(3) = 0

openat(AT\_FDCWD, "/lib/x86\_64-linux-gnu/libm.so.6", O\_RDONLY|O\_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0\0\0\0\0\0\0\0"..., 832) = 832

newfstatat(3, "", {st\_mode=S\_IFREG|0644, st\_size=907784, ...}, AT\_EMPTY\_PATH) = 0

mmap(NULL, 909560, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7f6d788e6000

mmap(0x7f6d788f6000, 471040, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x10000) = 0x7f6d788f6000

mmap(0x7f6d78969000, 368640, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x83000) = 0x7f6d78969000

mmap(0x7f6d789c3000, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0xdc000) = 0x7f6d789c3000

close(3) = 0

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f6d788e4000

arch\_prctl(ARCH\_SET\_FS, 0x7f6d788e5480) = 0

set\_tid\_address(0x7f6d788e5750) = 59499

set\_robust\_list(0x7f6d788e5760, 24) = 0

rseq(0x7f6d788e5da0, 0x20, 0, 0x53053053) = 0

mprotect(0x7f6d78b93000, 16384, PROT\_READ) = 0

mprotect(0x7f6d789c3000, 4096, PROT\_READ) = 0

mprotect(0x7f6d78bd1000, 4096, PROT\_READ) = 0

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7f6d788e2000

mprotect(0x7f6d78e2f000, 45056, PROT\_READ) = 0

mprotect(0x555995c52000, 4096, PROT\_READ) = 0

mprotect(0x7f6d78e7a000, 8192, PROT\_READ) = 0

prlimit64(0, RLIMIT\_STACK, NULL, {rlim\_cur=8192\*1024, rlim\_max=RLIM64\_INFINITY}) = 0

munmap(0x7f6d78e41000, 26550) = 0

futex(0x7f6d78e3d73c, FUTEX\_WAKE\_PRIVATE, 2147483647) = 0

getrandom("\x32\xaa\x1f\x6d\xd7\x46\x28\x81", 8, GRND\_NONBLOCK) = 8

brk(NULL) = 0x555996e83000

brk(0x555996ea4000) = 0x555996ea4000

newfstatat(1, "", {st\_mode=S\_IFCHR|0620, st\_rdev=makedev(0x88, 0x1), ...}, AT\_EMPTY\_PATH) = 0

write(1, "Creating allocators..\n", 22Creating allocators..

) = 22

**mmap(NULL, 10485760, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE| MAP\_ANONYMOUS, -1, 0) = 0x7f6d77ee2000**

**mmap(NULL, 10485760, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE| MAP\_ANONYMOUS, -1, 0) = 0x7f6d774e2000**

write(1, "All memory: 10485736 8388608\n", 29All memory: 10485736 8388608

) = 29

write(1, "Created\n", 8Created

) = 8

write(1, "TEST 1 -- allocate lots of memor"..., 58TEST 1 -- allocate lots of memory of a single little size

) = 58

write(1, "Cycles count = 5\n\n", 18Cycles count = 5

) = 18

write(1, "Block size = 1 Block count = 100"..., 35Block size = 1 Block count = 10000

) = 35

write(1, "Testing...\n", 11Testing...

) = 11

brk(0x555996ec9000) = 0x555996ec9000

write(1, "Result Allocate:\n", 17Result Allocate:

) = 17

write(1, "First-fit: 180714\302\265s Buddy:1505\302"..., 43First-fit: 180714µs Buddy:1505µs 120.076

) = 43

write(1, "Result Free:\n", 13Result Free:

) = 13

write(1, "First-fit: 163\302\265s Buddy:322\302\265s 0"..., 41First-fit: 163µs Buddy:322µs 0.506211

) = 41

brk(0x555996eb6000) = 0x555996eb6000

write(1, "Block size = 4 Block count = 100"..., 35Block size = 4 Block count = 10000

) = 35

write(1, "Testing...\n", 11Testing...

) = 11

brk(0x555996edd000) = 0x555996edd000

write(1, "Result Allocate:\n", 17Result Allocate:

) = 17

write(1, "First-fit: 162732\302\265s Buddy:737\302\265"..., 42First-fit: 162732µs Buddy:737µs 220.803

) = 42

write(1, "Result Free:\n", 13Result Free:

) = 13

write(1, "First-fit: 324\302\265s Buddy:672\302\265s 0"..., 41First-fit: 324µs Buddy:672µs 0.482143

) = 41

brk(0x555996ec9000) = 0x555996ec9000

brk(0x555996eb6000) = 0x555996eb6000

write(1, "Block size = 8 Block count = 100"..., 35Block size = 8 Block count = 10000

) = 35

write(1, "Testing...\n", 11Testing...

) = 11

brk(0x555996edd000) = 0x555996edd000

write(1, "Result Allocate:\n", 17Result Allocate:

) = 17

write(1, "First-fit: 156905\302\265s Buddy:1294\302"..., 43First-fit: 156905µs Buddy:1294µs 121.256

) = 43

write(1, "Result Free:\n", 13Result Free:

) = 13

write(1, "First-fit: 165\302\265s Buddy:361\302\265s 0"..., 41First-fit: 165µs Buddy:361µs 0.457064

) = 41

brk(0x555996ec9000) = 0x555996ec9000

brk(0x555996eb6000) = 0x555996eb6000

write(1, "Block size = 16 Block count = 10"..., 36Block size = 16 Block count = 10000

) = 36

write(1, "Testing...\n", 11Testing...

) = 11

brk(0x555996edd000) = 0x555996edd000

write(1, "Result Allocate:\n", 17Result Allocate:

) = 17

write(1, "First-fit: 18015\302\265s Buddy:5426\302\265"..., 42First-fit: 18015µs Buddy:5426µs 3.32013

) = 42

write(1, "Result Free:\n", 13Result Free:

) = 13

write(1, "First-fit: 70\302\265s Buddy:78\302\265s 0.8"..., 39First-fit: 70µs Buddy:78µs 0.897436

) = 39

brk(0x555996ec9000) = 0x555996ec9000

brk(0x555996eb6000) = 0x555996eb6000

write(1, "Block size = 32 Block count = 10"..., 36Block size = 32 Block count = 10000

) = 36

write(1, "Testing...\n", 11Testing...

) = 11

brk(0x555996edd000) = 0x555996edd000

write(1, "Result Allocate:\n", 17Result Allocate:

) = 17

write(1, "First-fit: 189435\302\265s Buddy:313\302\265"..., 42First-fit: 189435µs Buddy:313µs 605.224

) = 42

write(1, "Result Free:\n", 13Result Free:

) = 13

write(1, "First-fit: 162\302\265s Buddy:397\302\265s 0"..., 40First-fit: 162µs Buddy:397µs 0.40806

) = 40

brk(0x555996ec9000) = 0x555996ec9000

brk(0x555996eb6000) = 0x555996eb6000

write(1, "\n", 1

) = 1

write(1, "TEST 2 -- allocate a lot of memo"..., 69TEST 2 -- allocate a lot of memory in the face of high fragmentation

) = 69

write(1, "Allocate 5000 blocks size 100..."..., 33Allocate 5000 blocks size 100...

) = 33

write(1, "Free all even blocks alloc1...\n", 31Free all even blocks alloc1...

) = 31

brk(0x555996edd000) = 0x555996edd000

write(1, "Free all even blocks alloc2...\n", 31Free all even blocks alloc2...

) = 31

write(1, "Fragmentation completed\n", 24Fragmentation completed

) = 24

write(1, "Block size = 80 Block count = 50"..., 35Block size = 80 Block count = 5000

) = 35

write(1, "Testing...\n", 11Testing...

) = 11

write(1, "Result Allocate:\n", 17Result Allocate:

) = 17

write(1, "First-fit: 200846\302\265s Buddy:136\302\265"..., 42First-fit: 200846µs Buddy:136µs 1476.81

) = 42

write(1, "Result Free:\n", 13Result Free:

) = 13

write(1, "First-fit: 80438\302\265s Buddy:50194\302"..., 44First-fit: 80438µs Buddy:50194µs 1.60254

) = 44

brk(0x555996ec9000) = 0x555996ec9000

brk(0x555996eb6000) = 0x555996eb6000

write(1, "\n", 1

) = 1

write(1, "TEST 3 -- allocate random blocks"..., 33TEST 3 -- allocate random blocks

) = 33

brk(0x555996edd000) = 0x555996edd000

write(1, "Iteration 1:\n", 13Iteration 1:

) = 13

write(1, "Result Allocate:\n", 17Result Allocate:

) = 17

write(1, "First-fit: 348051\302\265s Buddy:724\302\265"..., 42First-fit: 348051µs Buddy:724µs 480.733

) = 42

write(1, "Result Free:\n", 13Result Free:

) = 13

write(1, "First-fit: 319\302\265s Buddy:817\302\265s 0"..., 41First-fit: 319µs Buddy:817µs 0.390453

) = 41

brk(0x555996ec9000) = 0x555996ec9000

brk(0x555996eb6000) = 0x555996eb6000

brk(0x555996edd000) = 0x555996edd000

write(1, "Iteration 2:\n", 13Iteration 2:

) = 13

write(1, "Result Allocate:\n", 17Result Allocate:

) = 17

write(1, "First-fit: 373500\302\265s Buddy:701\302\265"..., 41First-fit: 373500µs Buddy:701µs 532.81

) = 41

write(1, "Result Free:\n", 13Result Free:

) = 13

write(1, "First-fit: 301\302\265s Buddy:1044\302\265s "..., 42First-fit: 301µs Buddy:1044µs 0.288314

) = 42

brk(0x555996ec9000) = 0x555996ec9000

brk(0x555996eb6000) = 0x555996eb6000

brk(0x555996edd000) = 0x555996edd000

write(1, "Iteration 3:\n", 13Iteration 3:

) = 13

write(1, "Result Allocate:\n", 17Result Allocate:

) = 17

write(1, "First-fit: 333288\302\265s Buddy:701\302\265"..., 42First-fit: 333288µs Buddy:701µs 475.447

) = 42

write(1, "Result Free:\n", 13Result Free:

) = 13

write(1, "First-fit: 210\302\265s Buddy:845\302\265s 0"..., 41First-fit: 210µs Buddy:845µs 0.248521

) = 41

brk(0x555996ec9000) = 0x555996ec9000

brk(0x555996eb6000) = 0x555996eb6000

brk(0x555996edd000) = 0x555996edd000

write(1, "Iteration 4:\n", 13Iteration 4:

) = 13

write(1, "Result Allocate:\n", 17Result Allocate:

) = 17

write(1, "First-fit: 329978\302\265s Buddy:771\302\265"..., 42First-fit: 329978µs Buddy:771µs 427.987

) = 42

write(1, "Result Free:\n", 13Result Free:

) = 13

write(1, "First-fit: 337\302\265s Buddy:986\302\265s 0"..., 41First-fit: 337µs Buddy:986µs 0.341785

) = 41

brk(0x555996ec9000) = 0x555996ec9000

brk(0x555996eb6000) = 0x555996eb6000

brk(0x555996edd000) = 0x555996edd000

write(1, "Iteration 5:\n", 13Iteration 5:

) = 13

write(1, "Result Allocate:\n", 17Result Allocate:

) = 17

write(1, "First-fit: 347821\302\265s Buddy:1864\302"..., 43First-fit: 347821µs Buddy:1864µs 186.599

) = 43

write(1, "Result Free:\n", 13Result Free:

) = 13

write(1, "First-fit: 568\302\265s Buddy:2502\302\265s "..., 42First-fit: 568µs Buddy:2502µs 0.227018

) = 42

brk(0x555996ec9000) = 0x555996ec9000

brk(0x555996eb6000) = 0x555996eb6000

write(1, "\n", 1

) = 1

write(1, "TEST 4 -- Usage factor First-fit"..., 33TEST 4 -- Usage factor First-fit

) = 33

write(1, "All memory: 10485736\n", 21All memory: 10485736

) = 21

write(1, "Total allocated: 2635792 bytes\n", 31Total allocated: 2635792 bytes

) = 31

write(1, "Free memory: 7609944 bytes\n", 27Free memory: 7609944 bytes

) = 27

write(1, "Utilization factor: 0.916545\n", 29Utilization factor: 0.916545

) = 29

write(1, "\n", 1

) = 1

write(1, "TEST 5 -- Usage factor Buddy\n", 29TEST 5 -- Usage factor Buddy

) = 29

write(1, "All memory: 8388608\n", 20All memory: 8388608

) = 20

write(1, "Total allocated: 2609148 bytes\n", 31Total allocated: 2609148 bytes

) = 31

write(1, "Free memory: 4748832 bytes\n", 27Free memory: 4748832 bytes

) = 27

write(1, "Utilization factor: 0.716843\n", 29Utilization factor: 0.716843

) = 29

**munmap(0x7f6d77ee2000, 10485760) = 0**

**munmap(0x7f6d774e2000, 10485760) = 0**

exit\_group(0) = ?

+++ exited with 0 +++

**Результат сравнения**

Для оценки эффективности аллокаторов были выбраны тесты, которые моделируют реальные сценарии использования памяти. Это включает в себя различные паттерны выделения и освобождения памяти, такие как частое выделение и освобождение небольших блоков, выделение и освобождение больших блоков, а также смешанные сценарии. Такой подход позволяет получить представление о производительности аллокаторов в различных условиях и выявить их сильные и слабые стороны. Кроме того, тесты были проведены несколько раз, чтобы обеспечить объективность и повторяемость результатов.

* **Фактор использования**

Фактор использования является критически важным параметром, так как он определяет, насколько эффективно используется память. Высокий фактор использования означает, что большая часть выделенной памяти действительно используется, а не простаивает. Для того, чтобы посчитать фактор использования классы аллокаторов должны иметь доступ к количеству свободной памяти. Проверку будем проводить путем замера количества памяти до выделения блоков и после, а затем сравнить. Для того чтобы тест был максимально приближен к реальному размеры блоков будем выбирать произвольный из диапазона. Результаты в таблице.

|  |  |  |
| --- | --- | --- |
| Allocator | First-fit allocator  bytes | Buddy allocator  bytes |
| Buffer size | 8388608 | 8388608 |
| All memory | 8388584 | 4194304 |
| Total allocated | 2650884 | 2644665 |
| Free memory | 5497700 | 521376 |
| Utilization factor | 0.91698 | 0.720043 |

Исходя из алгоритма, можно заметить, что аллокатор на списке выделяет ровно столько памяти, сколько от него попросили, погрешность лишь составляет размер заголовка блока, который, к сожалению, довольно большой. В моей реализации он 24 байта, что порой существенно больше, чем запрашиваемая память. У аллокатора близнецов идеальный фактор использования составляет 75% при всех оптимизациях, так что результат в 72% считаю близким.

* **Скорость выделения и освобождения блоков**

Скорость выделения и освобождния блоков памяти является также ключевым показателем производительности аллокатора. Эффективный аллокатор должен минимизировать время, необходимое для выделения памяти, чтобы обеспечить быстрое выполнение программ.

Для того, чтобы при измерении времени работы разных аллокаторов учесть особенности каждого из алгоритмов тестирование включает в себя несколько этапов. Мы должны измерить время с разными размерами, и тд. Рассмотрим результаты тестов в таблицах.

1. Выделение большого числа блоков одинакового размера, идущих подряд. Число блоков 10000

|  |  |  |
| --- | --- | --- |
| Block size/Allocator | First-fit allocator  us | Buddy allocator  us |
| 4 | 186512 | 1027 |
| 8 | 158852 | 958 |
| 16 | 168647 | 826 |
| 32 | 170009 | 398 |
| 64 | 173157 | 340 |

В результате видно, что время выделения памяти у алгоритма на списках более стабильное и не зависит от размера блока. Скорость выделения зависит только от числа блоков в списке. Благодаря древовидной архитектуре и очередям свободных блоков разных размеров, метод двойников способен молниеносно выделять блоки памяти.

1. Освобождение большого числа блоков одинакового размера, идущих подряд. Число блоков 100

|  |  |  |
| --- | --- | --- |
| Block size/Allocator | First-fit allocator  us | Buddy allocator  us |
| 4 | 358 | 662 |
| 8 | 355 | 400 |
| 16 | 311 | 390 |
| 32 | 229 | 398 |

Результаты тоже связаны с архитектурой каждого из аллокаторов. При выделении одинаковых блоков подряд оба аллокатора имеют примерно одинаковое время удаления так как ни метод двойника ни список не могут найти свободного двойника или блок для объединения.

1. Аллокация большого числа блоков в условиях высокой фрагментации памяти

Очень важно, чтобы аллокатор быстро работал в условиях высокой фрагментации памяти. Для имитации такой ситуации в начале выделим блоки подряд, а затем половину блоков удалим. Посмотрим, как аллокаторы справятся.

Количество блоков 5000. Размер блока 70 байт, размер пропусков 100.

|  |  |  |
| --- | --- | --- |
| Allocator | First-fit allocator  us | Buddy allocator  us |
| Time alloc | 197016 | 185 |
| Time free | 104781 | 64559 |

И снова аллокатор метода двойников побеждает в несколько раз. Дело в том, что у аллокатора по алгоритму близнецов в момент фрагментации уже полностью построено дерево блоков и в каждой из очередей есть свободные блоки разных размеров, а в случае со списком, ему приходится проходить его заново каждый раз и искать свободный блок.

1. Аллокация блоков случайного размера.

|  |  |  |
| --- | --- | --- |
| Block`s count/Allocator | First-fit allocator  us | Buddy allocator  us |
| 10000 | 377167 | 720 |
| 10000 | 380743 | 766 |
| 100 | 16 | 7 |
| 100 | 17 | 7 |

Очень важно, чтобы аллокатор мог быстро выделять память случайного размера, так как при использовании аллокатора в операционной системе, данное действие будет выполняться постоянно. Так как размер блоков выбирается случайным образом, то очень важно, чтобы аллокаторы были в равных условиях, поэтому вначале сгенерируем массив случайных размеров, и затем будем создавать блоки в таком паттерне каждым аллокатором. Как видим Аллокатор на списках очень в несколько раз быстрее выделяет память благодаря своей архитектуре.

1. Освобождение блоков случайного размера.

|  |  |  |
| --- | --- | --- |
| Block`s count/Allocator | First-fit allocator  us | Buddy allocator  us |
| 10000 | 219 | 709 |
| 10000 | 206 | 697 |
| 100 | 1 | 6 |
| 100 | 1 | 7 |

Теперь рассмотрим освобождения блоков при выполнении теста номер 4. Как мы видим, аллокатор на списках наконец одерживает победу над аллокатором через метод двойников. Так как мы выделяем блоки случайного размера, то освобождаемые блоки не так часто сразу являются двойниками друг другу и аллокатору приходится при объединении уходит глубоко в дерево для поиска двойников в очереди. В отличии от этого аллокатор на списках всегда просто проходится по списку и объединяет соседей, если они свободны. В условиях малого числа блоков, это дает молниеносное освобождение памяти.

* **Простота использования аллокатора**

Аллокатор с использованием списков блоков является одним из самых простых аллокаторов, но тем не менее является довольно эффективным в некоторых ситуациях, он не требует сложной архитектуры и прост в написании и использовании. В отличии от него метод двойников использует довольно сложную внутреннюю архитектуру и из-за этого, он труден в написании. Если использовать аллокаторы в качестве библиотек, то они оба одинаковы легки в использовании, так как имеют функции с одинаковыми сигнатурами.

**Вывод**

В данной лабораторной работе мной были рассмотрены два ключевых метода аллокации памяти: с использованием списков свободных блоков и метод близнецов. Были написаны программы, реализующие оба алгоритма и проведено всестороннее тестирование для определения слабых и сильных сторон алгоритмов. Каждый из алгоритмов имеет свои особенности, позволяющие побеждать или проигрывать в тестах друг другу, так же не стоит забывать, что каждый из алгоритмов можно продолжать модифицировать и улучшать, для возрастания скорости работы. По результатам тестов лучшим выбором для большинства задач является аллокатор, использующий метод двойников. Благодаря своей архитектуре, метод близнецов обеспечивает очень высокую скорость динамического выделения и освобождения памяти, и используется в составе многих современных операционных систем для динамического распределения памяти в ядре системы, драйверах или в других ответственных компонентах системы, критичных к скорости работы.