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»

Студент: Чирикова П. С.
Группа: М8О-201Б-21
Вариант: №18
Преподаватель: Миронов Е.С.
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Содержание

1. Цель работы

- Приобретение практических навыков в использовании знаний, полученных в течении курса
- Проведение исследования в выбранной предметной области

2. Постановка задачи

Необходимо спроектировать и реализовать программный прототип в соответствии с выбранным вариантом. Произвести анализ и сделать вывод на основании данных, полученных при работе программного прототипа.

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.

Вариант №18: необходимо сравнить два алгоритма аллокации: блоки по 2 в степени n и алгоритм двойников.

Каждый аллокатор должен обладать следующим интерфейсом (могут быть отличия в зависимости от особенностей алгоритма):

- 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 аллокатору).

3. Общие сведения о программе

Программа представляет из себя 6 файлов: main.cpp N2Allocator.h

N2Allocator.cpp BuddyAllocator.h BuddyAllocator.cpp CMakeLists.txt

4. Подробное описание алгоритмов аллокации 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. 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 данного курсового проекта мне были предоставлены два алгоритма аллокации:

- Алгоритм двойников
- Блоки по 2^n

Рассмотрим подробнее алгоритмы: их реализации и характеристики.

Алгоритм двойников

В данном алгоритме свободный пул памяти разбивается до тех пор, пока не выйдет блок памяти нужного размера, в каждом блоке есть тэг, 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.

Алгоритм выделения блоков по 2^n

В данном алгоритме используется набор списков свободной памяти. В каждом списке хранятся буферы определенного размера, который всегда кратен степени числа 2. Каждый буфер имеет заголовок длинной в одно слово, этим фактом ограничивается возможности использования соотносимой с ним области памяти. Если буфер свободен, то в его заголовке хранится указатель на следующий свободный буфер. В другом случае в заголовок буфера помещается указатель на список, в которой он должен быть возвращен при освобождении.

5. Исходный код

main.cpp

```
#include <iostream>
#include <chrono>
```

```
#include "N2Allocator.h"
#include "BuddyAllocator.h"

int main() {
    using namespace std::chrono;
    {
        steady_clock::time_point buddy_allocator_init_start = steady_clock::now();
        BuddyAllocator buddy_allocator(4096);
        steady_clock::time_point buddy_allocator_init_end = steady_clock::now();
        std::cerr << "Buddy allocator initialization with one page of memory :"
                << std::chrono::duration_cast<std::chrono::nanoseconds>(
                        buddy_allocator_init_end - buddy_allocator_init_start).count()
                << " ns" << std::endl;


        steady_clock::time_point n2_allocator_init_start = steady_clock::now();
        N2Allocator n2_allocator(
                {.block_16 = 64, .block_32 = 32, .block_64 = 16, .block_128 = 4,
.block_256 = 2}); // 1 страница
        steady_clock::time_point n2_allocator_init_end = steady_clock::now();
        std::cerr << "N2 allocator initialization with one page of memory :"
                << std::chrono::duration_cast<std::chrono::nanoseconds>(
                        n2_allocator_init_end - n2_allocator_init_start).count()
                << " ns" << std::endl;

        std::cerr << "\n";


    }
    std::cerr << "First test: Allocate 10 char[256] arrays, free 5 of them, allocate 10
char[128] arrays:\n";
    {
        std::vector<char *> pointers(15, 0);
        N2Allocator allocator(
                {.block_16 = 0, .block_32 = 0,.block_64 = 20, .block_128 = 20, .block_256
= 20, .block_512 = 10});
        steady_clock::time_point n2_test1_start = steady_clock::now();
        for (int i = 0; i < 10; ++i) {
            pointers[i] = (char *) allocator.allocate(256);
        }
        for (int i = 5; i < 10; ++i) {
            allocator.deallocate(pointers[i]);
        }
```

```
        for (int i = 5; i < 15; ++i) {
            pointers[i] = (char *) allocator.allocate(128);
        }
        steady_clock::time_point n2_test1_end = steady_clock::now();
        std::cerr << "N2 allocator first test:"
                << std::chrono::duration_cast<std::chrono::microseconds>(n2_test1_end -
n2_test1_start).count()
                << " microseconds" << std::endl;
        allocator.PrintStatus(std::cerr);
        for (int i = 0; i < 15; ++i) {
            allocator.deallocate(pointers[i]);
        }
    }
    {
        BuddyAllocator allocator(8192);
        std::vector<char *> pointers(1000, 0);
        steady_clock::time_point test1_start = steady_clock::now();
        for (int i = 0; i < 10; ++i) {
            pointers[i] = (char *) allocator.allocate(256);
        }
        for (int i = 5; i < 10; ++i) {
            allocator.deallocate(pointers[i]);
        }
        for (int i = 5; i < 15; ++i) {
            pointers[i] = (char *) allocator.allocate(128);
        }
        steady_clock::time_point test1_end = steady_clock::now();
        std::cout << "Buddy allocator first test:"
                << std::chrono::duration_cast<std::chrono::microseconds>(test1_end -
test1_start).count()
                << " microseconds" << std::endl;
        allocator.PrintStatus(std::cerr);
        for (int i = 0; i < 15; ++i) {
            allocator.deallocate(pointers[i]);
        }
    }
    std::cerr << "Second test: Allocate and free 75 20 bytes arrays:\n";
    {
        N2Allocator allocator({.block_16 = 0, .block_32 = 400, .block_64 = 400});
        std::vector<char *> pointers(75, 0);
        steady_clock::time_point alloc_start = steady_clock::now();
        for (int i = 0; i < 75; ++i) {
```

```
            pointers[i] = (char *) allocator.allocate(20);
        }
        steady_clock::time_point alloc_end = steady_clock::now();
        for (int i = 0; i < 75; ++i) {
            allocator.deallocate(pointers[i]);
        }
        steady_clock::time_point test_end = steady_clock::now();
        std::cerr << "N2 allocator second test:\n"
                << "Allocation :" << duration_cast<std::chrono::microseconds>(alloc_end
- alloc_start).count() << " microseconds" << "\n"
                << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end - alloc_end).count() << "
microseconds" << "\n";
    }
    {
        BuddyAllocator allocator(16000);
        std::vector<char *> pointers(75, 0);
        steady_clock::time_point alloc_start = steady_clock::now();
        for (int i = 0; i < 75; ++i) {
            pointers[i] = (char *) allocator.allocate(20);
        }
        steady_clock::time_point alloc_end = steady_clock::now();
        for (int i = 0; i < 75; ++i) {
            allocator.deallocate(pointers[i]);
        }
        steady_clock::time_point test_end = steady_clock::now();
        std::cerr << "Buddy allocator second test:\n"
                << "Allocation :" << duration_cast<std::chrono::microseconds>(alloc_end
- alloc_start).count() << " microseconds" << "\n"
                << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end - alloc_end).count() << "
microseconds" << "\n";
    }
    std::cerr << "Third test: Allocate 50 20 bytes arrays, deallocate every second,
allocate 25 12 bytes :\n";
    {
        N2Allocator allocator({.block_16 = 400, .block_32 = 700});
        std::vector<char *> pointers(75, 0);
        steady_clock::time_point test_start = steady_clock::now();
        for (int i = 0; i < 50; ++i) {
            pointers[i] = (char *) allocator.allocate(20);
        }
```

```
        for (int i = 0; i < 25; ++i) {
            allocator.deallocate(pointers[i * 2]);
        }
        for (int i = 500; i < 75; ++i) {
            pointers[i] = (char*) allocator.allocate(12);
        }
        steady_clock::time_point test_end = steady_clock::now();
        std::cerr << "N2 allocator third test:"
                << std::chrono::duration_cast<std::chrono::microseconds>(test_end -
test_start).count()
                << " microseconds" << std::endl;
        allocator.PrintStatus(std::cerr);
        for (int i = 0; i < 25; ++i) {
            allocator.deallocate(pointers[i * 2 + 1]);
        }
        for (int i = 500; i < 75; ++i) {
            allocator.deallocate(pointers[i]);
        }
    }
    {
        BuddyAllocator allocator(16000);
        std::vector<char *> pointers(75, 0);
        steady_clock::time_point test_start = steady_clock::now();
        for (int i = 0; i < 50; ++i) {
            pointers[i] = (char *) allocator.allocate(20);
        }
        for (int i = 0; i < 25; ++i) {
            allocator.deallocate(pointers[i * 2]);
        }
        for (int i = 500; i < 75; ++i) {
            pointers[i] = (char*) allocator.allocate(12);
        }
        steady_clock::time_point test_end = steady_clock::now();
        std::cerr << "Buddy allocator third test:"
                << std::chrono::duration_cast<std::chrono::microseconds>(test_end -
test_start).count()
                << " microseconds" << std::endl;
        allocator.PrintStatus(std::cerr);
        for (int i = 0; i < 25; ++i) {
            allocator.deallocate(pointers[i * 2 + 1]);
        }
        for (int i = 500; i < 75; ++i) {
```

```
            allocator.deallocate(pointers[i]);
        }
    }
    std::cerr << "Fourth test: Allocate and free 150 20 bytes arrays:\n";
    {
        N2Allocator allocator({.block_16 = 0, .block_32 = 800, .block_64 = 800});
        std::vector<char *> pointers(150, 0);
        steady_clock::time_point alloc_start = steady_clock::now();
        for (int i = 0; i < 150; ++i) {
            pointers[i] = (char *) allocator.allocate(20);
        }
        steady_clock::time_point alloc_end = steady_clock::now();
        for (int i = 0; i < 150; ++i) {
            allocator.deallocate(pointers[i]);
        }
        steady_clock::time_point test_end = steady_clock::now();
        std::cerr << "N2 allocator fourth test:\n"
                  << "Allocation :" << duration_cast<std::chrono::microseconds>(alloc_end
- alloc_start).count() << " microseconds" << "\n"
                  << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end - alloc_end).count() << "
microseconds" << "\n";
    }
    {
        BuddyAllocator allocator(32000);
        std::vector<char *> pointers(150, 0);
        steady_clock::time_point alloc_start = steady_clock::now();
        for (int i = 0; i < 150; ++i) {
            pointers[i] = (char *) allocator.allocate(20);
        }
        steady_clock::time_point alloc_end = steady_clock::now();
        for (int i = 0; i < 150; ++i) {
            allocator.deallocate(pointers[i]);
        }
        steady_clock::time_point test_end = steady_clock::now();
        std::cerr << "Buddy allocator fourth test:\n"
                  << "Allocation :" << duration_cast<std::chrono::microseconds>(alloc_end
- alloc_start).count() << " microseconds" << "\n"
                  << "Deallocation :" <<
duration_cast<std::chrono::microseconds>(test_end - alloc_end).count() << "
microseconds" << "\n";
    }
```

```
    return 0;
}
```

N2Allocator.h

```
#ifndef N2ALLOCATOR_H
#define N2ALLOCATOR_H
#include <vector>
#include <iostream>
#include <list>
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:
    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, 512, 1024};
    std::vector<std::list<char*>> lists;
    char* data;
    int mem_size;
};
#endif //N2ALLOCATOR_H
```

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 wasn't 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.size() <<
std::endl;
        free_sum += lists[i].size() * index_to_size[i];
    }
    int occ_sum = mem_size - free_sum;

    os << "Occupied memory " << occ_sum << std::endl;
    os << "Free memory " << free_sum << std::endl << std::endl;
}
```

BuddyAllocator.h

```
#ifndef BUDDYALLOCATOR_H
#define BUDDYALLOCATOR_H
#include <vector>
#include <iostream>

class BuddyAllocator {
public:
   BuddyAllocator(const size_t allowedSize);

   ~BuddyAllocator();
   void* allocate(size_t mem_size);
   void deallocate(void *ptr);
   void PrintStatus(std::ostream& os) const;

private:
   std::vector<char*> freeBlocks;
   char* data;
   size_t mem_size;
};
#endif //BUDDYALLOCATOR_H
```

BuddyAllocator.cpp

```
#include "BuddyAllocator.h"
#include <algorithm>
void setBlock(char* p, size_t size) {
   *((int*) p) = size;
}

int getSize(char* p) {
   return *((int*) p);
}

BuddyAllocator::BuddyAllocator(const size_t allowedSize) :
mem_size{allowedSize} {
   data = (char*)malloc(allowedSize);
   setBlock(data, allowedSize);
   freeBlocks.push_back(data);
}
```

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

void *BuddyAllocator::allocate(size_t mem_size) {
   if (mem_size == 0) {
      return nullptr;
   }
   int index = -1;
   mem_size += sizeof(int);
   for (int i = 0; i < freeBlocks.size(); ++i) {
      if (getSize(freeBlocks[i]) >= mem_size) {
         index = i;
         break;
      }
   }
   if (index == -1) {
      throw std::bad_alloc();
   }
   size_t currentBlockSize = getSize(freeBlocks[index]);
   while ((currentBlockSize % 2 == 0) && (currentBlockSize / 2>= mem_size)) {
      currentBlockSize /= 2;
      char* newBlock = freeBlocks[index] + currentBlockSize;
      setBlock(newBlock, currentBlockSize);
      freeBlocks.push_back(newBlock);
   }
   setBlock(freeBlocks[index], currentBlockSize);
   freeBlocks.erase(std::next(freeBlocks.begin(), index));
   return freeBlocks[index] + sizeof(int);
}

void BuddyAllocator::deallocate(void *ptr) {
   char *c_ptr = (char*) ptr - sizeof(int);
   size_t size = getSize(c_ptr);
   auto found = std::find(freeBlocks.begin(), freeBlocks.end(), c_ptr + size);
   if (found != freeBlocks.end()) {
      freeBlocks.erase(found);
      setBlock(c_ptr, size * 2 );
      freeBlocks.push_back(c_ptr);
```

```
        return;
    }
    found = std::find(freeBlocks.begin(), freeBlocks.end(), c_ptr - size);
    if (found != freeBlocks.end()) {
        setBlock(c_ptr - size, size * 2);
        return;
    }
    freeBlocks.push_back(c_ptr);
}

void BuddyAllocator::PrintStatus(std::ostream &os) const {
    int free_sum = 0;
    for (auto block : freeBlocks) {
        free_sum += getSize(block);
    }
    int occ_sum = mem_size - free_sum;
    os << "Occupied memory: " << occ_sum << std::endl;
    os << "Free memory: " << free_sum << std::endl << std::endl;
}
```

CMakeLists.txt

```
add_executable(kp main.cpp src/BuddyAllocator.cpp src/N2Allocator.cpp)

target_include_directories(kp PRIVATE include)
```

5. Тестирование

Будем тестировать следующие характеристики:

1. Скорость выделения и освобождения блоков
2. Фрагментацию
3. Экономичность

Напишем программу для тестов и запустим их:

```
polina@polina-Vostro-3400:~$ cd OS
polina@polina-Vostro-3400:~/OS$ cd cmake-build-debug
polina@polina-Vostro-3400:~/OS/cmake-build-debug$ cd kp
polina@polina-Vostro-3400:~/OS/cmake-build-debug/kp$ ./kp
```

Buddy allocator initialization with one page of memory :4889 ns

N2 allocator initialization with one page of memory :27867 ns

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

N2 allocator first test:5 microseconds

List with 16 byte blocks, size: 6

List with 32 byte blocks, size: 6

List with 64 byte blocks, size: 6

List with 128 byte blocks, size: 6

List with 512 byte blocks, size: 6

List with 1024 byte blocks, size: 6

Occupied memory 7680

Free memory 16640

Buddy allocator first test:9 microseconds

Occupied memory: 6400

Free memory: 1792

Second test: Allocate and free 75 20 bytes arrays:

N2 allocator second test:

Allocation :9 microseconds

Deallocation :17 microseconds

Buddy allocator second test:

Allocation :23 microseconds

Deallocation :49 microseconds

Third test: Allocate 50 20 bytes arrays, deallocate every second, allocate 25 12 bytes :

N2 allocator third test:14 microseconds

List with 16 byte blocks, size: 6

List with 32 byte blocks, size: 6

List with 64 byte blocks, size: 6

List with 128 byte blocks, size: 6

List with 512 byte blocks, size: 6

List with 1024 byte blocks, size: 6

Occupied memory 800

Free memory 28000

Buddy allocator third test:35 microseconds

Occupied memory: 3125

Free memory: 12875

Fourth test: Allocate and free 150 20 bytes arrays:

N2 allocator fourth test:

Allocation :19 microseconds

Deallocation :35 microseconds

Buddy allocator fourth test:

Allocation :39 microseconds

Deallocation :150 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, что алгоритм двойников не эффективен по времени, но гораздо более эффективней по памяти

Аллокация и удаление 75 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 «двойник» приходится при каждой аллокации создавать подходящий блок делением, а затем также соединять свободные блоки

Аллокация 50 раз по 20 байт, освобождение каждого второго полученного указателя, аллокация 25 раз по 12 байт. Данный тест хорошо показывает, как эффективен алгоритм блоков по два при большом количестве маленьких блоков.

Аллокация и деаллокация 150 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 раза.

Время аллокации и деаллокации на алгоритме «двойников» увеличилось примерно в 2 раза.

Очевидно, что оба алгоритма устойчивы к увеличению запросов.

Анализируя выше написанное, можно сказать, что алгоритм выделения блоков по 2^n работает немного быстрее, чем алгоритм двойников, но оно и понятно, ведь к преимущества первого алгоритма можно отнести быстроту использования, простой интерфейс, где важнейший особенностью является процедура free(), потому что в ней не нужно задавать размер буфера, но у него также есть и недостатки: невозможно освободить буфер частично, что сильно портит работоспособность алгоритма при большой фрагментации, неэкономное распределение памяти из-за необходимости хранения заголовков буфера (например, для предоставления 512 байтов будет выделено 1024 байта, потому что заголовок займет еще 4 байта), а также данный алгоритм не может

обеспечить обратную передачу буферов распределителю страничного уровня после освобождения.

Говоря про алгоритм двойников, можно выделить следующие положительные аспекты: гибкость за счет возможности изменения размеров участков памяти и их повторное использование, легкий обмен памятью между распределителем и страничной системой. К недостаткам данного метода можно отнести его производительность, ведь каждый раз при освобождении буфера распределитель пытается соединить вместе как можно больше участков памяти, а также интерфейс, потому что процедура освобождения должна получать в качества аргумента и адрес, и размер буфера. Таким образом, можно заметить, что эти алгоритмы достаточно похожи и выбор между ними зависит от поставленной перед вами задачей: если вам нужна скорость, то больше подойдет алгоритм выделения блоков по 2^n, а если необходимо часто контактировать со страничной системой и не бояться фрагментации, то выбор падает на алгоритм двойников.

7. Выводы

Выполнив задания курсового проекта, я лучше стала понимать, как устроена память, узнала, зачем нужны аллокаторы памяти. Они нужны для обслуживания запросов по выделению и освобождению памяти от различных клиентов. Критериями оценки их эффективность является фактор использования, простой программный интерфейс, возможность освобождения лишь части занимаемой памяти и производительность (скорость выделения/освобождения блоков памяти). На практике я реализовала два алгоритма аллокации памяти: алгоритм двойников и алгоритм выделения блоков по 2 в степени n. Проведя их сравнения, я выяснила, что для разных целей полезен определенный из них. При выполнении опреленной задачи придется жертвовать каким-то плюсом другого.