Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

**Лабораторная работа №4 по курсу**

**«Операционные системы»**

Группа: М8О-210Б-23

Студент: Попов А.В.

Преподаватель: Бахарев В.Д.

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

Дата: 26.12.24

Москва, 2024

**Постановка задачи**

**Вариант 3.**

Исследовать два аллокатора памяти: необходимо реализовать два алгоритма

аллокации памяти и сравнить их по следующим характеристикам:

– Фактор использования

– Скорость выделения блоков

– Скорость освобождения блоков

– Простота использования аллокатора

Требуется создать две динамические библиотеки, реализующие два аллокатора,

соответственно. Библиотеки загружаются в память с помощью интерфейса ОС

(dlopen / LoadLibrary) для работы с динамическими библиотеками. Выбор

библиотеки, реализующей аллокатор, осуществляется чтением первого аргумента при

запуске программы (argv[1]). Этот аргумент должен содержать путь до

динамической библиотеки (относительный или абсолютный).

Если аргумент не передан или по переданному пути библиотеки не оказалось, то

указатели на функции, реализующие API аллокатора ниже, должны быть присвоены

функциям, которые оборачивают системный аллокатор ОС (mmap / VirtualAlloc)

в этот API. Эти аварийные оберточные функции должны быть реализованы внутри

программы, которая загружает динамические библиотеки (см. пример на GitHub Gist).

Каждый аллокатор памяти должен иметь функции аналогичные стандартным

функциям malloc и free (realloc, опционально). Перед работой каждый

аллокатор инициализируется свободными страницами памяти, выделенными

стандартными средствами ядра (mmap / VirtualAlloc). 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.

Каждый аллокатор должен обладать следующим интерфейсом (могут быть отличия в

зависимости от особенностей алгоритма):

- Allocator\* allocator\_create(void \*const memory, const size\_t

size) (инициализация аллокатора на памяти memory размера size);

* void allocator\_destroy(Allocator \*const allocator)

(деинициализация структуры аллокатора);

* void\* allocator\_alloc(Allocator \*const allocator, const

size\_t size) (выделение памяти аллокатором памяти размера size);

**Вариант 3**

Списки свободных блоков (первое подходящее) и алгоритм двойников

**Общий метод и алгоритм решения**

**Списки свободных блоков**

Данный алгоритм основан на хранении блоков свободной памяти в списке. Когда пользователь запрашивает новый блок памяти, алгоритм проходится по списку свободных блоков и находит первый подходящий блок. Если блок размера большего, чем нужный, то он разделяется на две части, ненужная часть добавляется в список свободных блоков. При очищении блок добавляется в список свободных блоков и также может выполняться слияние с соседними блоками, если они свободны.

**Алгоритм двойников**

Все блоки памяти имеют размер степени двойки. Когда нужно выделить новый блок, алгоритм проходится по массиву списков из свободных блоков размера 2^n, где n – индекс в этом массиве, и находит ближайший блок к нужному размеру. Если найденный блок больше, то он делится пополам, пока он не достигнет ближайшего возможного размера. Ненужные половинки добавляются в массив списков свободных блоков.

**Код программы**

**main.c**

/\*\*  
 \* @file  
 \* @brief  
 \* @details  
 \* @author xsestech   
 \* @date 23.12.2024  
 \*/  
  
#include <liballoc/alloc.h>  
#include <liballoc/fallback.h>  
#include <libio/io.h>  
#include <dlfcn.h>  
#include <sys/mman.h>  
#include <sys/time.h>  
  
#include "timer.h"  
  
#define ALLOC\_MEMORY\_SIZE 33554732 // 32MB  
  
int main(int argc, char \*\*argv) {  
 (void) argc;  
  
 if (argc != 2) {  
 print\_fd(STDERR\_FILENO, "Error: wrong number of args\n");  
 return -1;  
 }  
 allocator\_lib\_t lib;  
 if (allocator\_lib\_load(&lib, argv[1]) == -1) {  
 print\_fd(STDERR\_FILENO, "Error: couldn't load allocator");  
 return -1;  
 }  
 void \*memory = mmap(NULL, ALLOC\_MEMORY\_SIZE, PROT\_WRITE | PROT\_READ, MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0);  
 if (memory == MAP\_FAILED) {  
 print\_fd(STDERR\_FILENO, "Error: can't allocate memory\n");  
 return -1;  
 }  
 void\* allocator = lib.allocator\_create(memory, ALLOC\_MEMORY\_SIZE);  
 if (allocator == NULL) {  
 print\_fd(STDERR\_FILENO, "Alloc error\n");  
 munmap(memory, ALLOC\_MEMORY\_SIZE);  
 return -1;  
 }  
 srand(0);  
 timer\_t timer\_alloc, timer\_free;  
  
 const size\_t mems\_size = ALLOC\_MEMORY\_SIZE / 1024 \* 0.8;  
 void\*\* mems[mems\_size];  
 timer\_start(&timer\_alloc);  
 for (size\_t i = 0; i < mems\_size; ++i) {  
 mems[i] = lib.allocator\_alloc(allocator, 600);  
 if (!mems[i]) {  
 print\_fd(STDERR\_FILENO, "Error during allocation");  
 lib.allocator\_destroy(allocator);  
 munmap(memory, ALLOC\_MEMORY\_SIZE);  
 return -1;  
 }  
 }  
 timer\_end(&timer\_alloc);  
 print\_fd(STDOUT\_FILENO, "Allocation time:\n");  
 timer\_print(&timer\_alloc);  
 size\_t deallocated = 0;  
 for (size\_t i = 0; i < mems\_size; ++i) {  
 if (rand() % 2 == 0) {  
 lib.allocator\_free(allocator, mems[i]);  
 deallocated++;  
 }  
 }  
 timer\_start(&timer\_free);  
 deallocated \*= 0.8;  
 for (size\_t i = 0; i < deallocated / 2; ++i) {  
 if (!lib.allocator\_alloc(allocator, 512)) {  
 print\_fd(STDERR\_FILENO, "Error during allocation");  
 lib.allocator\_destroy(allocator);  
 munmap(memory, ALLOC\_MEMORY\_SIZE);  
 return -1;  
 }  
 }  
 timer\_end(&timer\_free);  
 print\_fd(STDOUT\_FILENO, "Free time:\n");  
 timer\_print(&timer\_free);  
  
 size\_t allocated\_count = 0;  
 while(lib.allocator\_alloc(allocator, 256)) {  
 allocated\_count++;  
 }  
  
 print\_fd(STDOUT\_FILENO, "Max 256 allocation count: %lu", allocated\_count);  
 lib.allocator\_destroy(allocator);  
  
  
  
 munmap(memory, ALLOC\_MEMORY\_SIZE);  
 return 0;  
}

}

**buddy.c**

*/\*\*  
 \* @file  
 \* @brief  
 \* @details  
 \* @author xsestech   
 \* @date 19.12.2024  
 \*/*#include <stdbool.h>  
#include <liballoc/buddy/buddy.h>  
  
#define MAX\_SIZE\_T\_POW2 32  
#define MIN\_BLOCK\_SIZE\_POW2 5  
  
#define get\_block\_mem\_from\_meta(block) \  
 &(block[1])  
  
#define get\_meta\_from\_mem\_ptr(mem) \  
 &(((allocator\_block\_meta\_t\*)mem)[-1])  
  
#define get\_next\_block\_ptr(block, new\_size) \  
(void\*)((char\*)block + (1 << new\_size))  
  
typedef struct allocator\_block\_meta\_t allocator\_block\_meta\_t;  
  
struct allocator\_block\_meta\_t {  
 size\_t size;  
 allocator\_block\_meta\_t \*prev\_in\_mem;  
 allocator\_block\_meta\_t \*prev;  
 allocator\_block\_meta\_t \*next;  
};  
  
struct allocator\_t {  
 allocator\_block\_meta\_t \*pow2blocks[MAX\_SIZE\_T\_POW2];  
 void \*max\_ptr;  
};  
  
allocator\_t \*allocator\_create(void \*const memory, size\_t size) {  
 if (!memory) {  
 return NULL;  
 }  
 if (size < sizeof(allocator\_t)) {  
 return NULL;  
 }  
 allocator\_t \*alloc = memory;  
 alloc->max\_ptr = (char \*) memory + size;  
 allocator\_block\_meta\_t \*next\_free = memory + sizeof(allocator\_t);  
 next\_free->size = log2(size - sizeof(allocator\_t));  
 next\_free->next = NULL;  
 next\_free->prev\_in\_mem = NULL;  
 for (size\_t i = 0; i < MAX\_SIZE\_T\_POW2; ++i) {  
 alloc->pow2blocks[i] = NULL;  
 }  
 alloc->pow2blocks[next\_free->size] = next\_free;  
 return alloc;  
}  
void allocator\_destroy(allocator\_t \*const allocator) {  
 for (size\_t i = 0; i < MAX\_SIZE\_T\_POW2; ++i) {  
 allocator->pow2blocks[i] = NULL;  
 }  
 allocator->max\_ptr = NULL;  
}  
  
static void remove\_from\_pow2blocks(allocator\_t \*const allocator, allocator\_block\_meta\_t \*block, size\_t level) {  
 if (block->prev) {  
 block->prev->next = block->next;  
 } else {  
 allocator->pow2blocks[level] = block->next;  
 }  
 if (block->next) {  
 block->next->prev = block->prev;  
 }  
}  
  
static void add\_to\_pow2blocks(allocator\_t \*const allocator, allocator\_block\_meta\_t \*block, size\_t level) {  
 block->next = allocator->pow2blocks[level];  
 block->prev = NULL;  
 if (allocator->pow2blocks[level]) {  
 allocator->pow2blocks[level]->prev = block;  
 }  
 allocator->pow2blocks[level] = block;  
}  
  
void \*allocator\_alloc(allocator\_t \*const allocator, const size\_t size) {  
 size\_t pow2size = ceil(log2(size + sizeof(allocator\_block\_meta\_t)));  
 if (pow2size < MIN\_BLOCK\_SIZE\_POW2) {  
 pow2size = MIN\_BLOCK\_SIZE\_POW2;  
 }  
 allocator\_block\_meta\_t \*block = NULL;  
 size\_t closest\_idx;  
 for (closest\_idx = pow2size; closest\_idx < MAX\_SIZE\_T\_POW2; ++closest\_idx) {  
 if (allocator->pow2blocks[closest\_idx]) {  
 block = allocator->pow2blocks[closest\_idx];  
 break;  
 }  
 }  
 if (block == NULL) {  
 return NULL;  
 }  
 if (closest\_idx == pow2size) {  
 remove\_from\_pow2blocks(allocator, block, pow2size);  
 block->next = NULL;  
 block->prev = NULL;  
 return get\_block\_mem\_from\_meta(block);  
 }  
 while (closest\_idx > pow2size) {  
 remove\_from\_pow2blocks(allocator, block, closest\_idx);  
 size\_t new\_size = block->size - 1;  
 allocator\_block\_meta\_t \*new\_block = get\_next\_block\_ptr(block, new\_size);  
 new\_block->size = new\_size;  
 new\_block->next = block;  
 new\_block->prev\_in\_mem = block;  
 new\_block->prev = NULL;  
  
 block->size = new\_size;  
 block->prev = new\_block;  
 block->prev\_in\_mem = NULL;  
 add\_to\_pow2blocks(allocator, new\_block, new\_size);  
 closest\_idx--;  
 }  
 remove\_from\_pow2blocks(allocator, block, pow2size);  
 block->next = NULL;  
 block->prev = NULL;  
 return get\_block\_mem\_from\_meta(block);  
}  
  
allocator\_block\_meta\_t \*allocator\_merge(allocator\_t \*const allocator, allocator\_block\_meta\_t \*block, bool \*was\_merged) {  
 \*was\_merged = false;  
  
 // Merge with the previous block if possible  
 if (block->prev\_in\_mem && block->prev\_in\_mem->size == block->size &&  
 get\_next\_block\_ptr(block->prev\_in\_mem, block->prev\_in\_mem->size) == block) {  
 remove\_from\_pow2blocks(allocator, block->prev\_in\_mem, block->size);  
 block->prev\_in\_mem->size++;  
 block = block->prev\_in\_mem;  
 \*was\_merged = true;  
 }  
  
 // Merge with the next block if possible  
 allocator\_block\_meta\_t \*next\_block = get\_next\_block\_ptr(block, block->size);  
 if ((void \*) next\_block < allocator->max\_ptr && next\_block->size == block->size &&  
 (next\_block->next || next\_block == allocator->pow2blocks[next\_block->size])) {  
 remove\_from\_pow2blocks(allocator, next\_block, block->size);  
 block->size++;  
 \*was\_merged = true;  
 }  
  
 return block;  
}  
  
void allocator\_free(allocator\_t \*const allocator, void \*const memory) {  
 if (!allocator || !memory) {  
 return;  
 }  
 allocator\_block\_meta\_t \*block = get\_meta\_from\_mem\_ptr(memory);  
 bool was\_merged = true;  
 while (was\_merged) {  
 block = allocator\_merge(allocator, block, &was\_merged);  
 }  
 add\_to\_pow2blocks(allocator, block, block->size);  
}

**io.c**

/\*\*  
 \* @file  
 \* @brief  
 \* @details  
 \* @author xsestech   
 \* @date 27.10.2024  
 \*/  
  
#include <libio/io.h>  
  
ssize\_t print\_fd(const int fd, char \*fmt, ...) {  
 va\_list args;  
 va\_start(args, fmt);  
 char buff[IO\_MAX\_STR\_LEN];  
 size\_t len = vsnprintf(buff, IO\_MAX\_STR\_LEN - 1, fmt, args);  
 const ssize\_t writen\_bytes = write(fd, buff, len);  
 va\_end(args);  
 return writen\_bytes;  
}  
ssize\_t write\_str(const int fd, const char \*buff) {  
 return write(fd, buff, strlen(buff));  
}  
  
ssize\_t reads\_fd(const int fd, char \*buff, const size\_t buff\_size) {  
 ssize\_t read\_bytes = 0;  
 return read(fd, buff, buff\_size);  
}

**child.c**

*/\*\*  
 \* @file  
 \* @brief  
 \* @details  
 \* @author xsestech   
 \* @date 26.10.2024  
 \*/*#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <fcntl.h>  
#include <libconfig/config.h>  
  
#include <libio/io.h>  
  
int main(const int argc, char \*argv[]) {  
 if (argc != 2) {  
 print\_fd(STDERR\_FILENO, "No file specified");  
 exit(EXIT\_FAILURE);  
 }  
 const pid\_t pid = getpid();  
 int file = open(argv[1], O\_WRONLY | O\_CREAT | O\_TRUNC | O\_APPEND, 0600);  
  
 if (file == -1) {  
 print\_fd(STDERR\_FILENO, "%d: Error opening file %s\n", pid, argv[1]);  
 exit(EXIT\_FAILURE);  
 }  
  
 print\_fd(STDOUT\_FILENO, "%d: opened file %s\n", getpid(), argv[1]);  
 char buffer[MAX\_LINE\_LENGTH];  
 ssize\_t bytes = 0;  
 while ((bytes = read(STDIN\_FILENO, buffer, MAX\_LINE\_LENGTH)) > 0) {  
 if (bytes == -1) {  
 print\_fd(STDERR\_FILENO, "%d: Error reading from pipe\n", pid);  
 exit(EXIT\_FAILURE);  
 }  
 if (buffer[0] == '\n') {  
 break;  
 }  
  
  
 buffer[bytes - 1] = '\0'; // remove newline  
 print\_fd(STDOUT\_FILENO, "%d: got: %s\n", pid, buffer);  
 if (write(file, buffer, bytes - 1) != bytes - 1) {  
 print\_fd(STDERR\_FILENO, "%d: Error writing to file\n", pid);  
 exit(EXIT\_FAILURE);  
 }  
 }  
 const char term = '\0';  
 write(file, &term, sizeof(term));  
 close(file);  
 return 0;  
}

**Freeblocks.c**

*/\*\*  
 \* @file  
 \* @brief  
 \* @details  
 \* @author xsestech   
 \* @date 19.12.2024  
 \*/*#include <liballoc/freeblocks/freeblocks.h>  
  
#define get\_block\_mem\_from\_meta(block) \  
 &(block[1])  
  
#define get\_meta\_from\_mem\_ptr(mem) \  
 &(((allocator\_block\_meta\_t\*)mem)[-1])  
  
#define get\_next\_block\_ptr(block, new\_size) \  
(void\*)((char\*)get\_block\_mem\_from\_meta(block) + new\_size - 1)  
  
typedef struct allocator\_block\_meta\_t allocator\_block\_meta\_t;  
  
struct allocator\_block\_meta\_t {  
 size\_t size;  
 allocator\_block\_meta\_t \*prev\_in\_mem;  
 allocator\_block\_meta\_t \*prev;  
 allocator\_block\_meta\_t \*next;  
};  
  
struct allocator\_t {  
 allocator\_block\_meta\_t \*next\_free;  
 void \*max\_ptr;  
};  
  
allocator\_t \*allocator\_create(void \*const memory, size\_t size) {  
 if (!memory) {  
 return NULL;  
 }  
 if (size < sizeof(allocator\_t)) {  
 return NULL;  
 }  
 allocator\_t \*alloc = memory;  
 alloc->max\_ptr = (char \*) memory + size;  
 allocator\_block\_meta\_t \*next\_free = get\_next\_block\_ptr(memory, sizeof(allocator\_t));  
 next\_free->size = size - sizeof(allocator\_t) - sizeof(allocator\_block\_meta\_t);  
 next\_free->next = NULL;  
 next\_free->prev\_in\_mem = NULL;  
 alloc->next\_free = next\_free;  
 return alloc;  
}  
void allocator\_destroy(allocator\_t \*const allocator) {  
 allocator->next\_free = NULL;  
 allocator->max\_ptr = NULL;  
}  
  
void \*allocator\_alloc(allocator\_t \*const allocator, const size\_t size) {  
 allocator\_block\_meta\_t \*block = allocator->next\_free;  
 allocator\_block\_meta\_t \*prev\_block = NULL;  
 if (block == NULL) {  
 return NULL;  
 }  
 while (block->next && block->size < size) {  
 prev\_block = block;  
 block = block->next;  
 }  
 if (block->size < size) {  
 return NULL;  
 }  
 if (block->size < size + sizeof(allocator\_block\_meta\_t)) {  
 if (block->next) {  
 allocator->next\_free = block->next;  
 block->next->prev = NULL; // Update prev pointer  
 }  
 block->next = NULL;  
 return get\_block\_mem\_from\_meta(block);  
 }  
 allocator\_block\_meta\_t \*new\_free\_block = get\_next\_block\_ptr(block, size);  
 new\_free\_block->size = block->size - size - sizeof(allocator\_block\_meta\_t);  
 new\_free\_block->next = block->next;  
 new\_free\_block->prev\_in\_mem = block;  
 new\_free\_block->prev = prev\_block;  
 if (prev\_block != NULL) {  
 prev\_block->next = new\_free\_block;  
 } else {  
 allocator->next\_free = new\_free\_block;  
 }  
 if (new\_free\_block->next) {  
 new\_free\_block->next->prev = new\_free\_block;  
 }  
 block->size = size;  
 block->next = NULL;  
 return get\_block\_mem\_from\_meta(block);  
}  
  
allocator\_block\_meta\_t \*allocator\_merge(allocator\_t \*const allocator, allocator\_block\_meta\_t \*block) {  
 if (block->prev\_in\_mem && get\_next\_block\_ptr(block->prev\_in\_mem, block->prev\_in\_mem->size) == block) {  
 block->prev\_in\_mem->size += block->size + sizeof(allocator\_block\_meta\_t);  
 block->prev\_in\_mem->prev = block->prev\_in\_mem->next;  
 block->prev\_in\_mem->next = block->next;  
 block = block->prev\_in\_mem;  
 }  
 allocator\_block\_meta\_t \*next\_block = get\_next\_block\_ptr(block, block->size);  
 if ((void \*) next\_block < allocator->max\_ptr && (next\_block->next || next\_block == allocator->next\_free)) {  
 block->size += next\_block->size + sizeof(allocator\_block\_meta\_t);  
 }  
 return block;  
}  
  
void allocator\_free(allocator\_t \*const allocator, void \*const memory) {  
 allocator\_block\_meta\_t \*block = get\_meta\_from\_mem\_ptr(memory);  
 block->next = allocator->next\_free;  
 if(allocator->next\_free) {  
 allocator->next\_free->prev = block;  
 }  
 block = allocator\_merge(allocator, block);  
 allocator->next\_free = block;  
}

**Alloc.c**

*/\*\*  
 \* @file  
 \* @brief  
 \* @details  
 \* @author xsestech   
 \* @date 25.12.2024  
 \*/*#include <liballoc/alloc.h>  
  
void allocator\_lib\_assign\_fallbacks(allocator\_lib\_t\* lib) {  
 lib->is\_fallback = true;  
 lib->allocator\_alloc = allocator\_alloc\_fallback;  
 lib->allocator\_free = allocator\_free\_fallback;  
 lib->allocator\_destroy = allocator\_destroy\_fallback;  
 lib->allocator\_create = allocator\_create\_fallback;  
}  
  
void allocator\_lib\_load\_func(allocator\_lib\_t \*lib, void \*\*pointer\_to\_func\_field, const char \*func\_name) {  
 if(!lib->is\_fallback) {  
 \*pointer\_to\_func\_field = dlsym(lib->handle, func\_name);  
 if (!lib->allocator\_alloc) {  
 print\_fd(STDERR\_FILENO, "Warning: Could not load function %s, using fallbacks: %s", func\_name, dlerror());  
 dlclose(lib->handle);  
 lib->handle = NULL;  
 allocator\_lib\_assign\_fallbacks(lib);  
 }  
 }  
}  
  
  
int allocator\_lib\_load(allocator\_lib\_t \*lib, const char \*path) {  
 if (!lib || !path) {  
 print\_fd(STDERR\_FILENO, "Error: Path or lib pointer can't be null");  
 return -1;  
 }  
 lib->handle = dlopen(path, RTLD\_LOCAL | RTLD\_NOW);  
 if (!lib->handle) {  
 print\_fd(STDERR\_FILENO, "Warning: Could not load library, using fallback functions: %s\n", dlerror());  
 allocator\_lib\_assign\_fallbacks(lib);  
 return 0;  
 }  
  
 dlerror();  
 allocator\_lib\_load\_func(lib, (void\*\*)&lib->allocator\_alloc, "allocator\_alloc");  
 allocator\_lib\_load\_func(lib, (void\*\*)&lib->allocator\_free, "allocator\_free");  
 allocator\_lib\_load\_func(lib, (void\*\*)&lib->allocator\_destroy, "allocator\_destroy");  
 allocator\_lib\_load\_func(lib, (void\*\*)&lib->allocator\_create, "allocator\_create");  
 return 0;  
}  
void allocator\_lib\_unload(allocator\_lib\_t \*lib) {  
 if (lib->handle) {  
 dlclose(lib->handle);  
 lib->handle = NULL;  
 }  
}

**Протокол работы программы**

**Методика тестирования:**

Для сравнения аллокаторов мы замерим время очищения памяти и ее выделения, а также для оценки эффективности использования памяти мы замерим, сколько блоков размера 256 будет возможно выделить при заполнении памяти на 80%. Для замера времени выделения будем выделять блоки размера 600, а вот очищение будет происходить случайным образом, но с зафиксированным стартовым числом генератора. Это позволит сравнить затраты на слияние памяти.

**Тестирование:**

**Buddy:**

alloc cmake-build-debug/liballoc-buddy/libliballoc-buddy.dylib

Allocation time:

Elapsed time: 0.002843

Free time:

Elapsed time: 0.000160

Max 256 allocation count: 14463

**Freeblocks prev\_ptr:**

alloc cmake-build-debug/liballoc-freeblocks/libliballoc-freeblocks.dylib

Allocation time:

Elapsed time: 0.001596

Free time:

Elapsed time: 0.042390

Max 256 allocation count: 100950

**Freeblocks:**

Allocation time:

Elapsed time: 0.001640

Free time:

Elapsed time: 0.090969

Max 256 allocation count: 74722

**Вывод**

Как мы можем видеть алгоритм двойников медленнее выделяет память, т.к. производится разделение блоков, но быстрее ее очищает. Фрагментация у алгоритма двойников выше, т.к. много памяти теряется на выделении блоков размера степени двойки. При добавлении указателя на предыдущий блок в памяти алгоритм свободных блоков фрагментируется еще меньше, но скорость очищения уменьшается в два раза.