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-211Б-23

Студент: Тропн М.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 02.10.24

Москва, 2024

Постановка задачи

Вариант 6.

Цель работы

Приобретение практических навыков в:

1) Создании аллокаторов памяти и их анализу;

2) Создании динамических библиотек и программ, использующие динамические библиотеки.

Задание

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);
- void allocator_free(Allocator *const allocator, void *const memory) (возвращает выделенную память аллокатору);

Задание:

6. Блоки по 2^n и алгоритм двойников;

Общий метод и алгоритм решения

Использованные системные вызовы:

- void *mmap(void *__addr, size_t __len, int __prot, int __flags, int __fd, off_t __offset) — создает новый маппинг в виртуальном адресном пространстве
- void exit(int status) - завершения выполнения процесса и возвращение статуса
- ssize_t read(int __fd, void* __buf, size_t __nbytes) — чтение из fd в буфер
- ssize_t write(int __fd, const void* __buf, size_t __n) — запись байтов в буфер
- int munmap(void *__addr, size_t __len) — удаление и освобождение mmap
- void *dlopen(const char *__file, int __mode) — открывает shared object и отображает его в память
- void *dlsym(void *__restrict__ __handle, const char *__restrict__ __name) — находит адрес времени выполнения в общем объекте HANDLE, на который ссылается символ с именем NAME
- int dlclose(void *__handle) — удаляет отображение и связь с shared object и закрывает его.

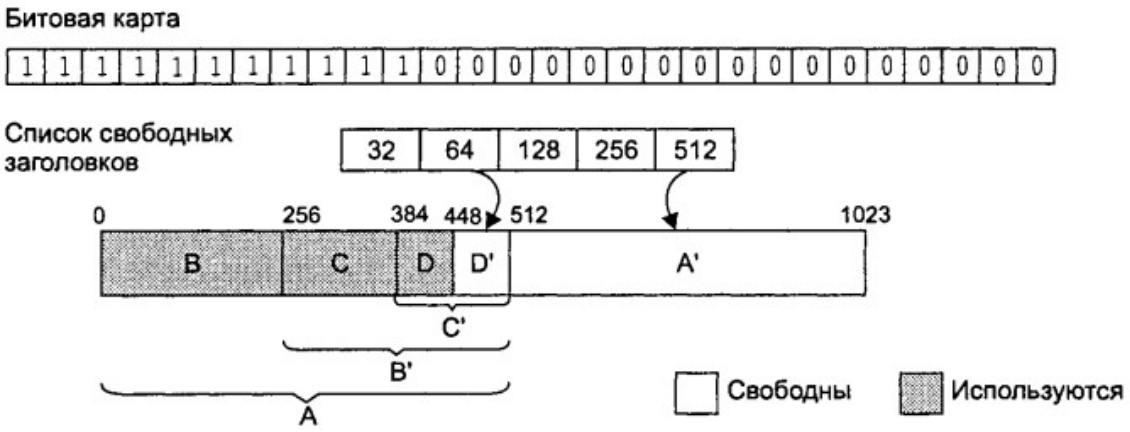
Алгоритм buddy allocator основан на управлении памятью путём разделения её на блоки, размеры которых являются степенями двойки. Для отслеживания состояния блоков используется битовая карта (bitmap), где каждому блоку соответствует один бит, определяющий, занят он или свободен.

Для создания аллокатора происходит инициализация структуры данных, в которой указываются: общий размер памяти, размер минимального блока (базовый размер), количество блоков и битовая карта. Битовая карта размещается в начале выделенной памяти и изначально заполняется нулями, указывая, что все блоки свободны. После битовой карты выделенная память разбивается на блоки минимального размера, готовые для последующих операций.

При запросе на выделение памяти размер запрашиваемого блока округляется до ближайшей степени двойки, превышающей размер. Аллокатор ищет в битовой карте первый доступный блок заданного или большего размера и помечает его как занятый. Если блок большего размера используется, он делится на более мелкие блоки до требуемого размера, а разделённые блоки отражаются в битовой карте.

При освобождении блока память помечается как свободная в битовой карте. Затем аллокатор проверяет "друга" (buddy) освобождаемого блока — соседний блок того же размера. Если и освобождаемый блок, и его "друг" свободны, они объединяются в один блок большего размера. Этот процесс может продолжаться рекурсивно, объединяя блоки верхнего уровня.

Таким образом, алгоритм поддерживает эффективное использование памяти, минимизируя фрагментацию и ускоряя операции выделения и освобождения блоков благодаря использованию битовой карты для управления состоянием блоков памяти.



Аллокатор «блоки по 2^n» управляет памятью, разделяя её на блоки размеров, являющихся степенями двойки. Он использует массив списков (freelists), где каждый список содержит свободные блоки определённого размера. Индексация массива списков соответствует степени двойки для размера блоков.

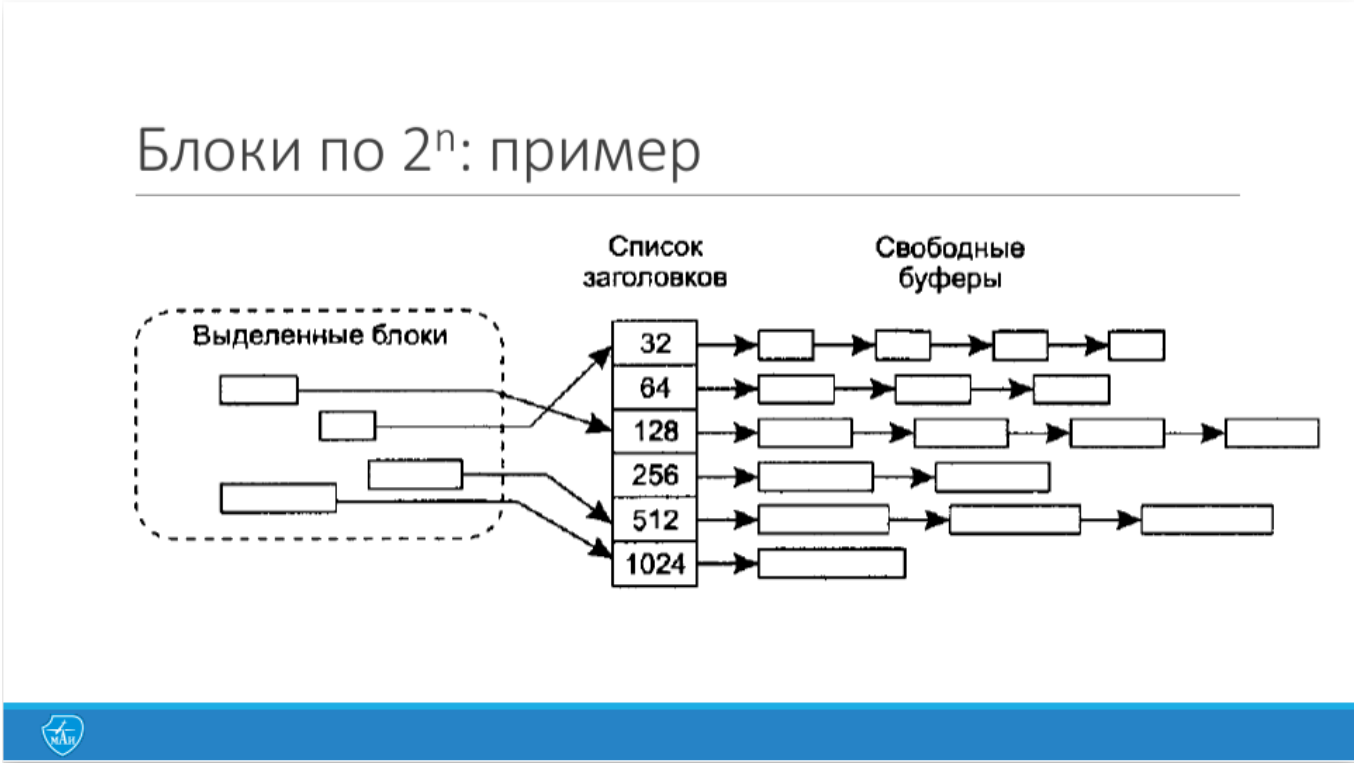
Для создания аллокатора память делится на блоки от минимального размера (MIN_BLOCK_SIZE) до максимально допустимого размера (MAX_BLOCK_SIZE), кратного степени двойки. Блоки группируются по размерам в списки свободных блоков и хранятся в массиве free_lists. Каждый блок содержит указатель на следующий блок в списке.

При запросе на выделение памяти размер округляется до ближайшей степени двойки, которая больше или равна запрашиваемому размеру. Аллокатор находит первый доступный блок из соответствующего списка в массиве free_lists. Если блоков нужного размера нет, выбирается блок большего размера из соответствующего списка, который затем делится на два блока меньшего

размера. Один из этих блоков возвращается, а второй добавляется в список свободных блоков меньшего размера.

Освобождение блока начинается с определением его размера. Размер хранится в специальной структуре block_t. Далее блок добавляется в соответствующий список свободных блоков. Если освобождаемый блок и его "сосед" (разделённый ранее из большего блока) свободны, они объединяются в блок большего размера, который затем добавляется в список свободных блоков большего размера. Этот процесс объединения может рекурсивно повторяться до тех пор, пока возможно слияние.

Алгоритм обеспечивает логарифмическую сложность операций выделения и освобождения памяти за счёт использования списков свободных блоков и структуры памяти, разделённой на блоки степеней двойки, что минимизирует внутреннюю фрагментацию.



Тестирование:

Проводиось функциональное тестирование (проверка работоспособности всех функций API), тест на корректное выделение памяти и отсуствие пересечений (со структурой и строкой), тест на «отлавливание» переполнения в аллокаторе и стресс-тестрование.

Помимо этого для каждого аллокатора высчитан фактор исползования. Для этого сумма всей необходимой к аллокации памяти делится на сумму всей затраченой аллокатором памяти для аллокаций.

Для тестового набора (строка (39 символов), динамический массив типа int32 (размер 52) и структура (приблизительный размер 72)) результаты:

buddy allocator - 0.675847 [319 / 472]

blocks2n allocator - 0.712054 [319 / 448]

Помимо этого проводились замеры времени аллокации и освобождения блоков. Для тестового блока динамический массив типа int32 размеров 52 (общий приблизительный размер 208) время составило:

Аллокация:

buddy allocator - 0.000002

blocks2n allocator - 0.000002

Освобождение:

buddy allocator - 0.000001

blocks2n allocator - 0.000002

Код программы

main.h

```
#ifndef __MAIN_H

#define __MAIN_H


#include <assert.h>

#include <dlfcn.h>  // dlopen, dlsym, dlclose, RTLD_*

#include <math.h>

#include <stdarg.h>

#include <stddef.h>

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/mman.h>

#include <time.h>

#include <unistd.h>

#include <unistd.h>  // write


// NOTE: MSVC compiler does not export symbols unless annotated

#ifdef _MSC_VER

#define EXPORT __declspec(dllexport)

#else

#define EXPORT

#endif


#define BUFSIZ 8192

typedef enum STATES { LOG_s, ERROR_s } STATES;


int _print(char mode, char *fmt, ...) {

  if (fmt == NULL) {

    write(STDERR_FILENO, "ERROR", 6);
```

```
    return 1;
  }
  va_list vargs;
  va_start(vargs, fmt);
  char msg[BUFSIZ];
  vsprintf(msg, fmt, vargs);
  write(mode == ERROR_s ? STDERR_FILENO : STDOUT_FILENO, msg, strlen(msg));
  va_end(vargs);
  return 0;
}

#define LOG(fmt, ...) _print(LOG_s, fmt, ##__VA_ARGS__)
#define ERROR(fmt, ...) _print(ERROR_s, fmt, ##__VA_ARGS__)

#define TIMER_INIT()            \
  clock_t start_time, end_time; \
  double timer_res;
#define TIMER_START() start_time = clock();
#define TIMER_END(text)                                   \
  end_time = clock();                                     \
  timer_res = (double)(end_time - start_time) / CLOCKS_PER_SEC; \
  LOG("%s %.6lf\n", text, timer_res);

#define HEAP_INIT(name, siz)                                        \
  Allocator *allocator;                                             \
  void *name;                                                       \
  name = mmap(NULL, siz, PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, \
              -1, 0);                                               \
  if (name == MAP_FAILED) {                                         \
    ERROR("mmap failed\n");                                         \
    return 1;                                                       \
  }
```

```
#define HEAP_DESTROY(name, siz)     \
  if (munmap(memory, SIZE) == -1) { \
    ERROR("munmap failed\n");       \
    exit(EXIT_FAILURE);             \
  };

struct Allocator {
  size_t total_size;        // общий размер
  void *memory;             // указатель на память
  long long in_use_mem;     // занятая память
  long long requested_mem;  // запрашиваемая память
};
typedef struct Allocator Allocator;

typedef struct Allocator_extra {
  size_t total_size;        // общий размер
  void *memory;             // указатель на память
  long long in_use_mem;     // занятая память
  long long requested_mem;  // запрашиваемая память
  size_t offset;
} Allocator_extra;

// инициализация аллокатора на памяти memory размера size
typedef Allocator *allocator_create_f(void *const memory, const size_t size);
// деинициализация структуры аллокатора
typedef void allocator_destroy_f(Allocator *const allocator);
// выделение памяти аллокатором памяти размера size
typedef void *allocator_alloc_f(Allocator *const allocator, const size_t size);
// возвращает выделенную память аллокатору
typedef void allocator_free_f(Allocator *const allocator, void *const memory);

typedef double allocator_usage_factor_f(Allocator *const allocator);
```

```
/* блок замененок */
static Allocator *allocator_create_extra(void *const memory,
                                         const size_t size) {
  if (memory == NULL || size == 0) {
    return NULL;
  }

  Allocator_extra *allocator = (Allocator_extra *)memory;
  allocator->memory = (void *)((uintptr_t)memory + sizeof(Allocator_extra));
  allocator->total_size = size - sizeof(Allocator_extra);
  allocator->offset = 0;
  return (Allocator *)allocator;
}

static void allocator_destroy_extra(Allocator *const allocator) {
  if (allocator == NULL) {
    return;
  }
  Allocator_extra *allo = (Allocator_extra *)allocator;

  allo->total_size = 0;
  allo->offset = 0;
}

static void *allocator_alloc_extra(Allocator *const allocator,
                                   const size_t size) {
  if (allocator == NULL || size == 0) {
    return NULL;
  }

  Allocator_extra *alloc = (Allocator_extra *)allocator;

  if (alloc->offset + size > alloc->total_size) {
```

```
    return NULL;
  }

  void *allocated_memory = (void *)((uintptr_t)alloc->memory + alloc->offset);
  alloc->offset += size;
  alloc->in_use_mem += size;
  return allocated_memory;
}

static void allocator_free_extra(Allocator *const allocator,
                                 void *const memory) {
  (void)allocator;
  (void)memory;
}
/**/

#endif  // __MAIN_H
```

main.c

```
#include "main.h"


#define SIZE (BUFSIZ)


static allocator_create_f* allocator_create;
static allocator_destroy_f* allocator_destroy;
static allocator_alloc_f* allocator_alloc;
static allocator_free_f* allocator_free;
static allocator_usage_factor_f* allocator_usage_factor;


typedef struct test_struct {
  int32_t meow;
  char name[53];
  double dbl;
} test_struct;


int main(int argc, char** argv) {
  (void)argc;
  LOG("mem create");


  void* library = dlopen(argv[1], RTLD_LOCAL | RTLD_NOW);
  /* библиотека смогла открыться */
  if (library != NULL) {
    allocator_create = dlsym(library, "allocator_create");
    if (allocator_create == NULL) {
      const char msg[] =
          "warning: failed to find allocator_create function implementation\n";
      write(STDERR_FILENO, msg, sizeof(msg));
      allocator_create = allocator_create_extra;
    }


    allocator_destroy = dlsym(library, "allocator_destroy");
```

```
  if (allocator_destroy == NULL) {
    const char msg[] =
        "warning: failed to find allocator_destroy function implementation\n";
    write(STDERR_FILENO, msg, sizeof(msg));
    allocator_destroy = allocator_destroy_extra;
  }

  allocator_alloc = dlsym(library, "allocator_alloc");
  if (allocator_alloc == NULL) {
    const char msg[] =
        "warning: failed to find allocator_alloc function implementation\n";
    write(STDERR_FILENO, msg, sizeof(msg));
    allocator_alloc = allocator_alloc_extra;
  }

  allocator_free = dlsym(library, "allocator_free");
  if (allocator_free == NULL) {
    const char msg[] =
        "warning: failed to find allocator_free function implementation\n";
    write(STDERR_FILENO, msg, sizeof(msg));
    allocator_free = allocator_free_extra;
  }

  allocator_usage_factor = dlsym(library, "allocator_usage_factor");
  if (allocator_free == NULL) {
    const char msg[] =
        "warning: failed to find allocator_usage_factor function "
        "implementation\n";
    write(STDERR_FILENO, msg, sizeof(msg));
    // allocator_usage_factor = allocator_usage_factor_extra;
  }
}
/* ====================================== */
```

```
/* использование стандартной библиотеки */
else {
  const char msg[] =
      "warning: library failed to load, trying standard implemntations\n";
  write(STDERR_FILENO, msg, sizeof(msg));
  allocator_create = allocator_create_extra;
  allocator_destroy = allocator_destroy_extra;
  allocator_alloc = allocator_alloc_extra;
  allocator_free = allocator_free_extra;
}

/* сами действие */
{
  Allocator* allocator;       // алокатор
  void* memory;               // пул памяти
  int* block1;                // тестовый блок 1
  char* block2;               // тестовый блок 2
  test_struct* block_struct;  // тестовый блок 3
  TIMER_INIT();               // времечко

  LOG("Создаем memory\n");
  HEAP_INIT(memory, SIZE);

  LOG("Создаем аллокатор\n");
  allocator = allocator_create(memory, SIZE);

  LOG("Аллоцируем\n");
  TIMER_START();
  block1 = (int*)allocator_alloc(allocator, sizeof(int) * 52);
  if (block1 == NULL) {
    ERROR("block1 NULL\n");
    exit(EXIT_FAILURE);
  }
```

```
TIMER_END("Аллокация заняла ");

for (size_t i = 0; i < 53; i++) {
  block1[i] = 27022005 - (i % 52);
}

int* test_for_free = block1 + 2;

block2 = (char*)allocator_alloc(allocator, 39);
if (block2 == NULL) {
  ERROR("block2 NULL\n");
  exit(EXIT_FAILURE);
}

/* тест на правильное выделение памяти и отсуствие пересечений */
block_struct =
    (test_struct*)allocator_alloc(allocator, sizeof(test_struct));

block_struct->dbl = __DBL_MAX__;
strncpy(block_struct->name,
        "qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM", 53);
block_struct->meow = INT32_MAX;
/**/

LOG("Фактор использования == %lf  [%zu]\n",
    (double)(39 + (sizeof(int) * 52) + sizeof(test_struct)) /
        (double)allocator->in_use_mem,
    allocator->in_use_mem);

allocator_free(allocator, block_struct);

sprintf(block2, "Meow meow meow ^_^\nHappy New Year!!!\0");
```

```
LOG("Алоцированный блок 1 живет по адресу %p и там есть %d\n", block1,
    *test_for_free);

LOG("Алоцированный блок 2 живет по адресу %p\n", block2);

LOG("block2 == %s\n", block2);

TIMER_START();
allocator_free(allocator, block1);
TIMER_END("Чистка блока заняла ");

allocator_free(allocator, block2);

void* TEST[SIZE];

/* тест на переполнение */
for (size_t i = 0; i < SIZE; i++) {
  // LOG("%zu saved\n", i);
  TEST[i] = allocator_alloc(allocator, 1);
}

void* block3 = allocator_alloc(allocator, 8192 * 2);  // Должно вернуть NULL
if (block3 == NULL) {
  LOG("Заполнилось всё - вернуло NULL\n");
}
// LOG("1\n");

/* просто рандомные стресс-тесты */
for (size_t i = 0; i < SIZE / 2; i++) {
  allocator_free(allocator, TEST[i]);
}

for (size_t i = 0; i < SIZE / 2; i++) {
```

```
    TEST[i] = allocator_alloc(allocator, 1);
  }

  for (size_t i = 0; i < SIZE; i++) {
    allocator_free(allocator, TEST[i]);
  }

  TIMER_START();
  for (size_t i = 0; i < INT16_MAX; i++) {
    void* block = allocator_alloc(allocator, 16);
    allocator_free(allocator, block);
  }
  TIMER_END("Массовое выделение и освобождение заняло ");

  LOG("Очищено\n");
  allocator_destroy(allocator);

  HEAP_DESTROY(memory, SIZE);
 }

 /* ==================================== */
 /* Отключение библиотек при наличии */
 if (library) {
  dlclose(library);
 }
 /* ==================================== */
}
```

buddys.c

```
#include "main.h"


#define PAGE_SIZE 32


typedef struct block_t {
  size_t size;  // Размер блока
} block_t;


typedef struct BuddyAllocator {
  size_t total_size;        // общий размер
  void *memory;             // указатель на память
  long long in_use_mem;     // занятая память
  long long requested_mem;  // запрашиваемая память
  size_t block_size;        // размер блока памяти
  size_t num_blocks;        // Общее количество блоков
  uint8_t *bitmap;  // Битовая карта для отслеживания свободных/занятых блоков
} BuddyAllocator;


EXPORT Allocator *allocator_create(void *const memory, const size_t size) {
  if (memory == NULL) {
    return NULL;
  }
  if (size < sizeof(BuddyAllocator)) {
    return NULL;
  }

  BuddyAllocator *allocator = (BuddyAllocator *)memory;
  // Устанавливаем параметры
  allocator->block_size = PAGE_SIZE;
  // Определяем количество блоков
  allocator->num_blocks = size / PAGE_SIZE;
  allocator->in_use_mem = 0;
```

```
  // bitmap = 1bit/block; флаг занятости
  size_t bitmap_size = (allocator->num_blocks + 7) / 8;

  allocator->memory =
      (void *)((uintptr_t)memory + sizeof(BuddyAllocator) + bitmap_size);
  allocator->bitmap = (uint8_t *)((char *)memory + sizeof(BuddyAllocator));

  // Обнуляем битовую карту
  memset(allocator->bitmap, 0, bitmap_size);
  allocator->total_size = size - bitmap_size - sizeof(BuddyAllocator);

  LOG("Buddys готовы\n");

  return (Allocator *)allocator;
}

EXPORT void allocator_destroy(Allocator *const allocator) {
  if (allocator == NULL) return;
  BuddyAllocator *b_allocator = (BuddyAllocator *)allocator;

  size_t bitmap_size = (b_allocator->num_blocks + 7) / 8;

  b_allocator->bitmap = NULL;
  b_allocator->block_size = 0;
  b_allocator->num_blocks = 0;
  b_allocator->total_size = 0;
  b_allocator->memory = NULL;
  b_allocator->in_use_mem = 0;
  return;
}

EXPORT void *allocator_alloc(Allocator *const allocator, const size_t size) {
  if (allocator == NULL || size == 0) {
```

```
  return NULL;
}
BuddyAllocator *b_allocator = (BuddyAllocator *)allocator;

// Вычисляем минимальный размер блока, кратного степени двойки
size_t block_size = b_allocator->block_size;
size_t block_index = 0;
while (block_size < size + sizeof(block_t)) {
  block_size *= 2;
  block_index++;
}

if (block_size > b_allocator->total_size) {
  return NULL;
}

// Поиск подходящего блока
size_t i = block_index;
while (i < b_allocator->num_blocks &&
       (b_allocator->bitmap[i / 8] & (1 << (i % 8)))) {
  i++;
}

if (i >= b_allocator->num_blocks) {
  ERROR("Не найден свободный блок подходящего размера\n");
  return NULL;
}

// Разделение блоков на меньшие, если требуется
while (i > block_index) {
  i--;
  size_t buddy_index = i ^ 1;  // Находим индекс соседа
  b_allocator->bitmap[buddy_index / 8] &= ~(1 << (buddy_index % 8));
```

```
  }

  // Помечаем найденный блок как занятый
  b_allocator->bitmap[i / 8] |= (1 << (i % 8));

  // Вычисляем адрес блока
  uintptr_t block_address = (uintptr_t)b_allocator->memory + i * block_size;

  // Сохраняем заголовок
  block_t *header = (block_t *)block_address;
  header->size = block_size;

  // Обновляем статистику
  b_allocator->in_use_mem += block_size + sizeof(block_t);

  // Возвращаем указатель на память сразу после заголовка
  return (void *)(block_address + sizeof(block_t));
}

EXPORT void allocator_free(Allocator *const allocator, void *const memory) {
  if (memory == NULL || allocator == NULL) {
    return;
  }
  BuddyAllocator *b_allocator = (BuddyAllocator *)allocator;

  // Получаем адрес заголовка
  uintptr_t block_address = (uintptr_t)memory - sizeof(block_t);
  block_t *header = (block_t *)block_address;

  size_t block_size = header->size;
  size_t block_index =
      (block_address - (uintptr_t)b_allocator->memory) / block_size;
```

```
  // Освобождаем блок в битовой карте
  size_t byte_index = block_index / 8;
  size_t bit_index = block_index % 8;
  b_allocator->bitmap[byte_index] &= ~(1 << bit_index);

  // Обновляем статистику
  b_allocator->in_use_mem -= (block_size + sizeof(block_t));

  // Пытаемся объединить блоки
  while (block_size < b_allocator->total_size) {
    size_t buddy_index = block_index ^ 1;  // Находим индекс соседа
    if (!(b_allocator->bitmap[buddy_index / 8] & (1 << (buddy_index % 8)))) {
      // Если соседний блок свободен, объединяем
      b_allocator->bitmap[buddy_index / 8] &= ~(1 << (buddy_index % 8));
      block_index /= 2;  // Переходим к объединенному блоку
      block_size *= 2;
    } else {
      break;  // Если сосед занят, прекращаем объединение
    }
  }

  // memset(header, 0, block_size);
  return;
}
```

blocks2n.c

```
#include "main.h"


// размер страницы
#define MIN_BLOCK_SIZE 1
// самый большой блок (не факт даже что будет и нужен)
#define MAX_BLOCK_SIZE (INT32_MAX)
// кол-во списков пусть всего 2^31
#define MAX_BLOCK_CNT 31


typedef struct block_t {
  struct block_t *next;  // следующий свободный
  size_t size;           // размер блока
} block_t;


typedef struct p2Alloc {
  size_t total_size;        // общий размер
  void *memory;             // указатель на память
  long long in_use_mem;     // занятая память
  long long requested_mem;  // запрашиваемая память
  block_t *free_lists[MAX_BLOCK_CNT];  // массив списков свободных блоков
} p2Alloc;


// узнаем размер блока, который можно забить под нужный размер
size_t get_block_size(size_t size) {
  if (size == 1) {
    return 1;
  }
  size_t block_size = 2;
  while (block_size < size) {  // + sizeof(block_t)
    block_size <<= 1;
  }
  return block_size;
```

```
}

// получаем индекс списка блоков свободных необходимого размера
int get_list_index(size_t block_size) {
  int index = 0;
  while (block_size > MIN_BLOCK_SIZE) {
    block_size >>= 1;
    index++;
  }
  return index - 1;
}

EXPORT Allocator *allocator_create(void *const memory, const size_t size) {
  if (size < sizeof(p2Alloc) || memory == NULL) {
    return NULL;
  }

  // Выделение памяти для структуры аллокатора
  p2Alloc *allocator = (p2Alloc *)memory;
  allocator->memory = (void *)((uintptr_t)memory + sizeof(p2Alloc));
  allocator->total_size = size - sizeof(p2Alloc);
  allocator->in_use_mem = 0;
  allocator->requested_mem = 0;

  // Инициализация списков свободных блоков
  memset(allocator->free_lists, 0, sizeof(allocator->free_lists));

  if (allocator->total_size < MIN_BLOCK_SIZE) {
    return (Allocator *)allocator;
  }

  // Начинаем разбиение памяти на блоки
  size_t min_block_size = get_block_size(sizeof(block_t));
```

```
  size_t cnt = 0;
  while (min_block_size <= size) {
    cnt++;
    min_block_size <<= 1;
  }

  // LOG("cnt = %zu\n", cnt);
  size_t remaining_size = allocator->total_size;
  size_t cur = sizeof(p2Alloc);

  for (long long i = cnt - 1; i >= 0; i--) {
    if (remaining_size < MIN_BLOCK_SIZE) break;

    size_t block_size = MIN_BLOCK_SIZE << i;
    if (cur < size && block_size <= remaining_size) {
      block_t *block = (block_t *)((uint8_t *)memory + cur);
      block->size = block_size;
      block->next = NULL;

      allocator->free_lists[i] = block;
      cur += block_size;
      remaining_size -= block_size;
    }
  }

  return (Allocator *)allocator;
}

EXPORT void allocator_destroy(Allocator *const allocator) {
  if (allocator == NULL) {
    return;
  }
  p2Alloc *alloc = (p2Alloc *)allocator;
```

```
  memset(alloc->free_lists, 0, sizeof(alloc->free_lists));
  alloc->total_size = 0;
  alloc->in_use_mem = 0;
  alloc->requested_mem = 0;
  alloc->memory = NULL;
  return;
}

EXPORT void *allocator_alloc(Allocator *const allocator, const size_t size) {
  if (NULL == allocator || size == 0) {
    return NULL;
  }

  p2Alloc *alloc = (p2Alloc *)allocator;

  if (alloc->total_size + sizeof(Allocator) <= size) {
    return NULL;
  }

  size_t block_size = get_block_size(size);
  int index = get_list_index(block_size);

  if (index < 0 || index >= MAX_BLOCK_CNT) {
    return NULL;
  }

  // Ищем свободный блок подходящего размера или больше
  for (int i = index; i < MAX_BLOCK_CNT; ++i) {
    if (alloc->free_lists[i] != NULL) {
      block_t *block = alloc->free_lists[i];
      alloc->free_lists[i] = block->next;  // Удаляем его из списка

      // Разбиваем блок на меньшие, если он больше нужного размера
```

```
    while (i > index) {
      i--;
      size_t smaller_block_size = 1 << i;  // Размер меньшего блока
      uintptr_t split_addr = (uintptr_t)block + smaller_block_size;

      // Создаём новый меньший блок
      block_t *split_block = (block_t *)split_addr;
      split_block->size = smaller_block_size;
      split_block->next = alloc->free_lists[i];
      alloc->free_lists[i] = split_block;
    }

    block->size = block_size;
    alloc->in_use_mem += block_size;
    alloc->requested_mem += size;

    return (void *)((uintptr_t)block + sizeof(block_t));
  }
}

// Попытка выделить память за пределами списка свободных блоков
if (alloc->total_size - alloc->in_use_mem >= block_size) {
  uintptr_t overflow_addr = (uintptr_t)alloc->memory + alloc->in_use_mem;
  block_t *block = (block_t *)overflow_addr;
  block->size = block_size;

  alloc->in_use_mem += block_size + sizeof(block_t);
  alloc->requested_mem += size;

  return (void *)((uintptr_t)block + sizeof(block_t));
}

// Если подходящего блока нет и переполнение невозможно
```

```
  return NULL;
}

EXPORT void allocator_free(Allocator *const allocator, void *const memory) {
  if (NULL == allocator || NULL == memory) {
    return;
  }

  p2Alloc *alloc = (p2Alloc *)allocator;

  if (alloc->total_size == 0) {
    return;
  }

  uintptr_t memory_addr = (uintptr_t)memory - sizeof(block_t);
  uintptr_t alloc_start = (uintptr_t)alloc->memory;
  uintptr_t alloc_end = alloc_start + alloc->total_size;

  if (memory_addr < alloc_start || memory_addr >= alloc_end) {
    return;
  }

  block_t *block = (block_t *)((uintptr_t)memory - sizeof(block_t));

  size_t block_size = block->size;

  if (block_size > MAX_BLOCK_SIZE || block_size < MIN_BLOCK_SIZE) {
    return;
  }

  int index = get_list_index(block_size);

  if (index < 0 || index >= MAX_BLOCK_CNT) {
```

```
    return;
  }

  memset(block, 0, block_size);
  block->next = alloc->free_lists[index];
  alloc->free_lists[index] = block;
  alloc->in_use_mem -= (block_size + sizeof(block_t));
  alloc->requested_mem -= block_size - sizeof(block_t);

  // LOG("meow meow %lld __ %lld\n", alloc->requested_mem / alloc->in_use_mem);
}
```

Протокол работы программы

Тестирование:

lemito@lemito:~/Desktop/OSi/lab4$ ./main.out ./blocks2n.so
Создаем аллокатор
Аллоцируем
Аллокация заняла 0.000002
Фактор использования == 0.712054 [319 / 448]
Алоцированный блок 1 живет по адресу 0x71703397d428 и там есть 27022003
Алоцированный блок 2 живет по адресу 0x71703397d4e8
block2 == Meow meow meow ^_^
Happy New Year!!!
Чистка блока заняла 0.000002
Заполнилось всё - вернуло NULL
Массовое выделение и освобождение заняло 0.002453
Очищено
lemito@lemito:~/Desktop/OSi/lab4$ ./main.out ./buddys.so
Создаем аллокатор
Buddys готовы
Аллоцируем
Аллокация заняла 0.000002
Фактор использования == 0.675847 [319 / 472]
Алоцированный блок 1 живет по адресу 0x713bc4208c40 и там есть 27022003
Алоцированный блок 2 живет по адресу 0x713bc42085c0
block2 == Meow meow meow ^_^
Happy New Year!!!
Чистка блока заняла 0.000001
Заполнилось всё - вернуло NULL
Массовое выделение и освобождение заняло 0.004372
Очищено

Strace:

lemito@lemito:~/Desktop/OSi/lab4$ strace -f ./main.out ./buddys.so

execve("./main.out", ["./main.out", "./buddys.so"], 0x7ffcabfc9110 /* 89 vars */) = 0

brk(NULL)   = 0x5a37e6eed000

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7be4d3b1d000

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

fstat(3, {st_mode=S_IFREG|0644, st_size=98319, ...}) = 0

mmap(NULL, 98319, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7be4d3b04000

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\220\243\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

fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 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, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7be4d3800000

mmap(0x7be4d3828000, 1605632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|
MAP_DENYWRITE, 3, 0x28000) = 0x7be4d3828000

mmap(0x7be4d39b0000, 323584, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000)
= 0x7be4d39b0000

mmap(0x7be4d39ff000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
3, 0x1fe000) = 0x7be4d39ff000

mmap(0x7be4d3a05000, 52624, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
MAP_ANONYMOUS, -1, 0) = 0x7be4d3a05000

close(3)                                = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7be4d3b01000

arch_prctl(ARCH_SET_FS, 0x7be4d3b01740) = 0

set_tid_address(0x7be4d3b01a10)         = 19335

set_robust_list(0x7be4d3b01a20, 24)     = 0

rseq(0x7be4d3b02060, 0x20, 0, 0x53053053) = 0

mprotect(0x7be4d39ff000, 16384, PROT_READ) = 0

mprotect(0x5a37be72e000, 4096, PROT_READ) = 0

mprotect(0x7be4d3b55000, 8192, PROT_READ) = 0

prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0

munmap(0x7be4d3b04000, 98319)           = 0

getrandom("\x14\x0f\xa4\xed\x43\xee\x87\xbc", 8, GRND_NONBLOCK) = 8

brk(NULL)                               = 0x5a37e6eed000

brk(0x5a37e6f0e000)                     = 0x5a37e6f0e000

openat(AT_FDCWD, "./buddys.so", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\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

fstat(3, {st_mode=S_IFREG|0775, st_size=15760, ...}) = 0

getcwd("/home/lemito/Desktop/OSi/lab4", 128) = 30

mmap(NULL, 16440, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7be4d3b18000

mmap(0x7be4d3b19000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x1000) = 0x7be4d3b19000
```

mmap(0x7be4d3b1a000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7be4d3b1a000

mmap(0x7be4d3b1b000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7be4d3b1b000

close(3) = 0

openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

fstat(3, {st_mode=S_IFREG|0644, st_size=98319, ...}) = 0

mmap(NULL, 98319, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7be4d3ae8000

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"..., 832) = 832

fstat(3, {st_mode=S_IFREG|0644, st_size=952616, ...}) = 0

mmap(NULL, 950296, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7be4d3717000

mmap(0x7be4d3727000, 520192, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x10000) = 0x7be4d3727000

mmap(0x7be4d37a6000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x8f000) = 0x7be4d37a6000

mmap(0x7be4d37fe000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xe7000) = 0x7be4d37fe000

close(3) = 0

mprotect(0x7be4d37fe000, 4096, PROT_READ) = 0

mprotect(0x7be4d3b1b000, 4096, PROT_READ) = 0

munmap(0x7be4d3ae8000, 98319) = 0

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7be4d3b16000

write(1, "\320\241\320\276\320\267\320\264\320\260\320\265\320\274 \
320\260\320\273\320\273\320\276\320\272\320\260\321\202\320\276\321"..., 34Создаем аллокатор

) = 34

write(1, "Buddys \320\263\320\276\321\202\320\276\320\262\321\213\n", 20Buddys готовы

) = 20

write(1, "\320\220\320\273\320\273\320\276\321\206\320\270\321\200\321\203\320\265\320\274\n", 21Аллоцируем

) = 21

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=3527496}) = 0

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=3539656}) = 0

write(1, "\320\220\320\273\320\273\320\276\320\272\320\260\321\206\320\270\321\217 \
320\267\320\260\320\275\321\217\320\273\320\260 "..., 42Аллокация заняла 0.000012

) = 42

```
	write(1, "\320\244\320\260\320\272\321\202\320\276\321\200 \
320\270\321\201\320\277\320\276\320\273\321\214\320\267\320\276\320\262\320"..., 59Фактор использования == 0.675847  [472]

	) = 59

	write(1, "\
320\220\320\273\320\276\321\206\320\270\321\200\320\276\320\262\320\260\320\275\320\275\321\213\320\271 \
320\261\320\273\320"..., 110Алоцированный блок 1 живет по адресу 0x7be4d3b16c40 и там есть 27022003

	) = 110

	write(1, "\
320\220\320\273\320\276\321\206\320\270\321\200\320\276\320\262\320\260\320\275\320\275\321\213\320\271 \
320\261\320\273\320"..., 82Алоцированный блок 2 живет по адресу 0x7be4d3b165c0

	) = 82

	write(1, "block2 == Meow meow meow ^_^\nHap"..., 47block2 == Meow meow meow ^_^

	Happy New Year!!!

	) = 47

	clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=3663653}) = 0

	clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=3675401}) = 0

	write(1, "\320\247\320\270\321\201\321\202\320\272\320\260 \320\261\320\273\320\276\320\272\320\260 \
320\267\320\260\320\275\321\217"..., 47Чистка блока заняла  0.000012

	) = 47

	write(1, "\320\227\320\260\320\277\320\276\320\273\320\275\320\270\320\273\320\276\321\201\321\214 \
320\262\321\201\321\221 - "..., 52Заполнилось всё - вернуло NULL

	) = 52

	clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=5672252}) = 0

	clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=10460427}) = 0

	write(1, "\320\234\320\260\321\201\321\201\320\276\320\262\320\276\320\265 \
320\262\321\213\320\264\320\265\320\273\320\265\320\275\320"..., 87Массовое выделение и освобождение заняло 0.004788

	) = 87

	write(1, "\320\236\321\207\320\270\321\211\320\265\320\275\320\276\n", 15Очищено

	) = 15

	munmap(0x7be4d3b16000, 8192)            = 0

	munmap(0x7be4d3b18000, 16440)           = 0

	munmap(0x7be4d3717000, 950296)          = 0

	exit_group(0)                           = ?

	+++ exited with 0 +++
```

Вывод

В ходе лабораторной работы я приобрел практические навыки в Создании аллокаторов памяти и их анализу и Создании динамических библиотек и программ, использующие динамические библиотеки. Составить и отладить программу на языке Си, реализующую жва алгоритма выделения памяти (аллокатор). Проблем в ходе выполнения не возникло.