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: Григорян А.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 22.02.25

Москва, 2025

Постановка задачи

Вариант 6.

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

Блоки по 2^n и алгоритм двойников;

Общий метод и алгоритм решения

Блоки по 2^n

Алгоритм аллокации памяти с фиксированными блоками (2n2^n) разделяет память на пулы блоков заранее определённых размеров (16, 32, 64, 128 байт и т. д.). При запросе выделяется ближайший по размеру блок, округляя объём вверх до степени двойки. Освобождённые блоки возвращаются в соответствующий пул для повторного использования. Этот метод обеспечивает быструю аллокацию и минимизирует фрагментацию, но страдает от перерасхода памяти (внутренняя фрагментация) и требует предварительного резервирования под разные пулы.

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

2n-alloc.c

```
#include "alloc.h"

Allocator* allocator_create(void *const memory, const size_t size) {
    Allocator *allocator = (Allocator *)memory;
    memset(allocator, 0, sizeof(Allocator));
```

```
    allocator->total_size = size - sizeof(Allocator);
    allocator->used_size = 0;

    Block *first_block = (Block *)(memory + sizeof(Allocator));
    first_block->next = NULL;
    allocator->free_lists[0] = first_block;

    return allocator;
}

void allocator_destroy(Allocator *const allocator) {
    memset(allocator, 0, sizeof(Allocator));
}

size_t round_up_to_power_of_two(size_t size) {
    size_t power = 1;
    while (power < size) power <<= 1;
    return power;
}

void* allocator_alloc(Allocator *const allocator, const size_t size) {
    size_t block_size = round_up_to_power_of_two(size);
    int index = 0;
    while ((1 << index) < block_size) index++;

    if (!allocator->free_lists[index]) return NULL;

    Block *block = allocator->free_lists[index];
    allocator->free_lists[index] = block->next;
```

```
    allocator->used_size += block_size;
    return (void *)block;
}

void allocator_free(Allocator *const allocator, void *const memory) {
    if (!memory) return;

    size_t block_size = 0;
    for (int i = 0; i < 32; i++) {
        if (allocator->free_lists[i]) {
            block_size = 1 << i;
            break;
        }
    }

    Block *block = (Block *)memory;
    block->next = allocator->free_lists[block_size];
    allocator->free_lists[block_size] = block;

    allocator->used_size -= block_size;
}
```

buddy-alloc.c

```
#include "alloc.h"
#include <math.h>
#include <stdio.h> // Для отладочной информации
#include <stdbool.h>
#include <sys/mman.h> // Для функции munmap

void allocator_init(struct Allocator* allocator) {
    for (int i = 0; i < NUM_FREE_LISTS; i++) {
```

```
        allocator->freelist[i] = NULL;
    }
}

size_t round_up_to_power_of_two(size_t size) {
    if (size < MIN_BLOCK_SIZE) {
        return MIN_BLOCK_SIZE;
    }
    return (size_t)pow(2, ceil(log2(size)));
}

struct Allocator* allocator_create(void* memory, size_t size) {
    // Округляем размер памяти до ближайшей степени двойки
    size_t rounded_size = round_up_to_power_of_two(size);

    if (rounded_size < sizeof(struct Allocator)) {
        return NULL;
    }

    struct Allocator* allocator = (struct Allocator*)memory;
    allocator->base_memory = memory;
    allocator->current_memory = (void*)((char*)memory + sizeof(struct Allocator));
    allocator->initial_size = rounded_size - sizeof(struct Allocator);
    allocator->remaining_size = allocator->initial_size;
    allocator_init(allocator);

    if (allocator->remaining_size >= MIN_BLOCK_SIZE) {
        struct FreeBlock* block = (struct FreeBlock*)allocator->current_memory;
        block->size = allocator->remaining_size;
        block->next = NULL;
```

```
        int index = get_index(block->size);
        allocator->freelist[index] = block;
    }

    return allocator;
}

void allocator_destroy(struct Allocator* allocator) {
    allocator->current_memory = allocator->base_memory;
    allocator->remaining_size = allocator->initial_size;
    for (int i = 0; i < NUM_FREE_LISTS; i++) {
        allocator->freelist[i] = NULL;
    }
    munmap(allocator->base_memory, allocator->initial_size + sizeof(struct Allocator));
}

int get_index(size_t size) {
    int index = 0;
    while ((1 << index) < size) {
        index++;
    }
    return index;
}

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

    size_t alloc_size = sizeof(struct FreeBlock) + size;
```

```
int index = get_index(alloc_size);

if (index >= NUM_FREE_LISTS) {
    return NULL;
}

struct FreeBlock* block = allocator->freelist[index];

if (block != NULL) {
    allocator->freelist[index] = block->next;
    block->size = size;
    return (void*)(block + 1);
} else {
    if (index < NUM_FREE_LISTS - 1) {
        void* ptr = allocator_alloc(allocator, (1 << (index + 1)) - sizeof(struct FreeBlock));
        if (ptr == NULL) {
            return NULL;
        }
        block = (struct FreeBlock*)ptr - 1;
        block->size = size;
        struct FreeBlock* buddy = (struct FreeBlock*)((char*)ptr + (1 << index));
        buddy->size = (1 << index) - sizeof(struct FreeBlock);
        buddy->next = allocator->freelist[index];
        allocator->freelist[index] = buddy;
        return ptr;
    } else {
        if (allocator->remaining_size < alloc_size) {
            return NULL;
        }
        block = (struct FreeBlock*)allocator->current_memory;
```

```
            block->size = size;
            allocator->current_memory = (void*)((char*)allocator->current_memory + alloc_size);
            allocator->remaining_size -= alloc_size;
            return (void*)(block + 1);
        }
    }
}

void allocator_free(struct Allocator* allocator, void* ptr) {
    if (ptr == NULL) {
        return;
    }

    struct FreeBlock* block = (struct FreeBlock*)ptr - 1;
    size_t size = block->size;
    int index = get_index(size + sizeof(struct FreeBlock));

    block->next = allocator->freelist[index];
    allocator->freelist[index] = block;

    // printf("Освобожден блок размером %zu на уровне %d\n", block->size, index); // Отладочная информация

    // Пытаемся объединить блок с его близнецом
    while (index < NUM_FREE_LISTS) {
        size_t block_size = 1 << index;
        size_t buddy_address = ((size_t)block - (size_t)allocator->base_memory) ^ block_size;
        struct FreeBlock* buddy = (struct FreeBlock*)((char*)allocator->base_memory + buddy_address);

        struct FreeBlock** list = &allocator->freelist[index];
```

```
struct FreeBlock* prev = NULL;
struct FreeBlock* curr = *list;

bool buddy_found = false;
while (curr) {
    if (curr == buddy) {
        if (prev) {
            prev->next = curr->next;
        } else {
            *list = curr->next;
        }

        if ((size_t)buddy < (size_t)block) {
            block = buddy;
        }

        block->size = block_size * 2;
        index++;
        block->next = allocator->freelist[index];
        allocator->freelist[index] = block;
        printf("Объединены блоки до размера %zu на уровне %d\n", block->size, index);
        buddy_found = true;
        break;
    }
    prev = curr;
    curr = curr->next;
}

if (!buddy_found) {
    break; // Близнец не найден, прекращаем объединение
```

```
    }
  }
}
```

test.c

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <time.h>

typedef struct Allocator Allocator;

typedef Allocator* (*create_func)(void *const memory, const size_t size);
typedef void (*destroy_func)(Allocator *const allocator);
typedef void* (*alloc_func)(Allocator *const allocator, const size_t size);
typedef void (*free_func)(Allocator *const allocator, void *const memory);

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <path_to_library>\n", argv[0]);
        return 1;
    }

    void *handle = dlopen(argv[1], RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        return 1;
    }

    create_func allocator_create = (create_func)dlsym(handle, "allocator_create");
    destroy_func allocator_destroy = (destroy_func)dlsym(handle, "allocator_destroy");
```

```
    alloc_func allocator_alloc = (alloc_func)dlsym(handle, "allocator_alloc");
    free_func allocator_free = (free_func)dlsym(handle, "allocator_free");

    void *memory = malloc(1 << 20); // 1 MB
    Allocator *allocator = allocator_create(memory, 1 << 20);

    // Тестирование
    clock_t start = clock();
    for (int i = 0; i < 10000; i++) {
        void *ptr = allocator_alloc(allocator, 64);
        allocator_free(allocator, ptr);
    }
    clock_t end = clock();

    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Time spent: %.6f seconds\n", time_spent);

    allocator_destroy(allocator);
    free(memory);
    dlclose(handle);

    return 0;
}
```

Протокол работы программы

Методика тестирования:

Для сравнения аллокаторов мы замерим время очищения памяти и ее выделения. Для этого программа в цикле 10000 раз запрашивает блок памяти и освобождает это. На основе этой метрики произведем сравнение работы двух динамических библиотек.

Тестирование:

Buddy:

Time spent: 0.000244 seconds

Блоки по 2^n:

Time spent: 0.000151 seconds

Strace

[arcsenius@ars-nbdewxx9 src]$ strace ./test ./2n-alloc/alloc.so

execve("./test", ["./test", "./2n-alloc/alloc.so"], 0x7ffd7792e768 /* 66 vars */) = 0

brk(NULL) = 0x62a2e9e60000

access("/etc/ld.so.preload", R_OK) = -1 ENOENT (Нет такого файла или каталога)

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

fstat(3, {st_mode=S_IFREG|0644, st_size=241615, ...}) = 0

mmap(NULL, 241615, PROT_READ, MAP_PRIVATE, 3, 0) = 0x795940789000

close(3) = 0

openat(AT_FDCWD, "/usr/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\340_\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=2014520, ...}) = 0

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x795940787000

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, 2034616, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x795940596000

mmap(0x7959405ba000, 1511424, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x24000) = 0x7959405ba000

mmap(0x79594072b000, 319488, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x195000) = 0x79594072b000

mmap(0x795940779000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e3000) = 0x795940779000

mmap(0x79594077f000, 31672, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x79594077f000

close(3) = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x795940593000

arch_prctl(ARCH_SET_FS, 0x795940593740) = 0

```
set_tid_address(0x795940593a10)         = 15668

set_robust_list(0x795940593a20, 24)     = 0

rseq(0x795940594060, 0x20, 0, 0x53053053) = 0

mprotect(0x795940779000, 16384, PROT_READ) = 0

mprotect(0x62a2bf332000, 4096, PROT_READ) = 0

mprotect(0x7959407fe000, 8192, PROT_READ) = 0

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

munmap(0x795940789000, 241615)          = 0

getrandom("\x84\xaf\x34\xd5\xbf\xb1\x37\x29", 8, GRND_NONBLOCK) = 8

brk(NULL)                               = 0x62a2e9e60000

brk(0x62a2e9e81000)                     = 0x62a2e9e81000

openat(AT_FDCWD, "./2n-alloc/alloc.so", 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

fstat(3, {st_mode=S_IFREG|0755, st_size=15104, ...}) = 0

getcwd("/home/arcsenius/CLionProjects/OS_Labs/lab_4/src", 128) = 48

mmap(NULL, 16416, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7959407bf000

mmap(0x7959407c0000, 4096, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7959407c0000

mmap(0x7959407c1000, 4096, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7959407c1000

mmap(0x7959407c2000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7959407c2000

close(3)                                = 0

mprotect(0x7959407c2000, 4096, PROT_READ) = 0

mmap(NULL, 1052672, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -
1, 0) = 0x795940492000

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=1519194}) = 0

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=2058315}) = 0

fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0x88, 0x3), ...}) = 0

write(1, "Time spent: 0.000539 seconds\n", 29Time spent: 0.000539 seconds

) = 29

munmap(0x795940492000, 1052672)         = 0

munmap(0x7959407bf000, 16416)           = 0
```

```
exit_group(0)                   = ?
+++ exited with 0 +++
```

Вывод

Результаты показали, что аллокатор с фиксированными блоками по 2n2^n2n работает быстрее, так как он использует заранее подготовленные пулы памяти, где выделение и освобождение блоков происходит за постоянное время O(1)O(1)O(1), без необходимости сложных операций объединения и разделения. В отличие от него, buddy-аллокатор требует дополнительных вычислений для поиска подходящего блока, а также потенциального слияния фрагментов при освобождении, что приводит к увеличению времени работы. Таким образом, тест подтвердил, что для частых и небольших выделений памяти фиксированные блоки по 2n2^n2n обеспечивают лучшую производительность.\