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.

**Тестирование**

Тестирование происходит по пунктам, указанным в задании: замеряется время выделение блоков памяти и время их очистки. Берутся блоки по 32 байта, операции происходят 10000 раз

**allocator блоки по 2^n**

Allocation time for 10000 allocations: 0.003283 seconds

Free time for 10000 deallocations: 0.000433 seconds

**allocator двойников**

Allocation time for 10000 allocations: 0.001474 seconds

Free time for 10000 deallocations: 0.190526 seconds

Результаты подтверждают теоретическое предположение: 2n allocator дольше выделяет память, т к происходит поиск нужного блока в списке. Buddy allocator же выделяет память быстрее, т к свободные блоки путем разбиения памяти по схеме бинарного дерева. Из-за операции слияния свободных блоков этот тип аллокатора тратит на очистку памяти больше времени, относительно другого алгоритма аллокации.

**Код программы**

**2n-alloc.c**

#ifndef ALLOC\_H

#define ALLOC\_H

#include <stdint.h>

#include <stddef.h>

#include <string.h>

typedef struct Block {

size\_t size; // Размер блока

struct Block \*next; // Следующий свободный блок

} Block;

typedef struct Allocator {

Block \*free\_lists[32]; // Списки свободных блоков для разных размеров

size\_t total\_size; // Общий размер памяти

size\_t used\_size; // Использованный размер

} Allocator;

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

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

size\_t round\_up\_to\_power\_of\_two(size\_t size);

void\* allocator\_alloc(Allocator \*const allocator, const size\_t size);

void allocator\_free(Allocator \*const allocator, void \*const memory);

#endif

#include "alloc.h"

// Создание аллокатора

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

if (!memory || size < sizeof(Allocator) + sizeof(Block)) return NULL;

Allocator \*allocator = (Allocator \*)memory;

memset(allocator, 0, sizeof(Allocator));

allocator->total\_size = size - sizeof(Allocator);

allocator->used\_size = 0;

// Инициализация первого свободного блока (вся доступная память)

Block \*first\_block = (Block \*)((char\*)memory + sizeof(Allocator));

first\_block->size = allocator->total\_size;

first\_block->next = NULL;

int index = 0;

while ((1 << index) < first\_block->size) index++;

allocator->free\_lists[index] = first\_block;

return allocator;

}

// Уничтожение аллокатора

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

memset(allocator, 0, sizeof(Allocator));

}

// Округление до степени двойки

size\_t round\_up\_to\_power\_of\_two(size\_t size) {

size\_t power = 1;

while (power < size) power <<= 1;

return power;

}

// Разбиение блоков

Block\* split\_block(Block \*block, size\_t required\_size, int index) {

size\_t remaining\_size = block->size - required\_size;

if (remaining\_size >= sizeof(Block)) {

Block \*new\_block = (Block \*)((char \*)block + required\_size);

new\_block->size = remaining\_size;

new\_block->next = block->next; // Устанавливаем указатель next

block->size = required\_size;

return new\_block;

}

return NULL;

}

// Выделение памяти

void\* allocator\_alloc(Allocator \*const allocator, const size\_t size) {

size\_t block\_size = round\_up\_to\_power\_of\_two(size + sizeof(Block));

int index = 0;

while ((1 << index) < block\_size) index++;

if (index >= 32) return NULL; // Проверка выхода за границы массива

// Поиск подходящего блока

while (index < 32 && !allocator->free\_lists[index]) index++;

if (index >= 32) return NULL;

Block \*block = allocator->free\_lists[index];

allocator->free\_lists[index] = block->next;

// Разбиение блока

Block \*remaining\_block = split\_block(block, block\_size, index);

if (remaining\_block) {

int new\_index = 0;

while ((1 << new\_index) < remaining\_block->size) new\_index++;

remaining\_block->next = allocator->free\_lists[new\_index];

if (allocator->free\_lists[new\_index] != remaining\_block) {

allocator->free\_lists[new\_index] = remaining\_block;

}

}

allocator->used\_size += block\_size;

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

}

// Слияние блоков

void merge\_blocks(Allocator \*allocator, Block \*block) {

int index = 0;

while ((1 << index) < block->size) index++;

Block \*prev = NULL;

Block \*current = allocator->free\_lists[index];

while (current) {

if ((char \*)current + current->size == (char \*)block) {

current->size += block->size;

return;

} else if ((char \*)block + block->size == (char \*)current) {

block->size += current->size;

block->next = current->next;

if (prev) {

prev->next = block;

} else {

allocator->free\_lists[index] = block;

}

return;

}

prev = current;

current = current->next;

}

block->next = allocator->free\_lists[index];

allocator->free\_lists[index] = block;

}

// Освобождение памяти

void allocator\_free(Allocator \*const allocator, void \*const memory) {

if (!memory) return;

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

allocator->used\_size -= block->size;

merge\_blocks(allocator, block);

}

**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);

#define ALLOC\_COUNT 10000

#define BLOCK\_SIZE 32

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

if (memory == NULL){

printf("Error alloc memory");

}

Allocator \*allocator = allocator\_create(memory, 1 << 20);

if (allocator == NULL){

printf("Error alloc memory");

}

void \*pointers[ALLOC\_COUNT];

// Измерение времени выделения памяти

clock\_t start\_alloc = clock();

for (int i = 0; i < ALLOC\_COUNT; i++) {

pointers[i] = allocator\_alloc(allocator, BLOCK\_SIZE);

}

clock\_t end\_alloc = clock();

double alloc\_time = (double)(end\_alloc - start\_alloc) / CLOCKS\_PER\_SEC;

printf("Allocation time for %d allocations: %.6f seconds\n", ALLOC\_COUNT, alloc\_time);

// Измерение времени освобождения памяти

clock\_t start\_free = clock();

for (int i = 0; i < ALLOC\_COUNT; i++) {

allocator\_free(allocator, pointers[i]);

}

clock\_t end\_free = clock();

double free\_time = (double)(end\_free - start\_free) / CLOCKS\_PER\_SEC;

printf("Free time for %d deallocations: %.6f seconds\n", ALLOC\_COUNT, free\_time);

allocator\_destroy(allocator);

free(memory);

dlclose(handle);

return 0;

}

**Протокол работы программы**

execve("./test", ["./test", "./buddy-alloc/alloc.so"], 0x7fff3270eed8 /\* 66 vars \*/) = 0

brk(NULL) = 0x5bb37f5c9000

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) = 0x7138d40c9000

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\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) = 0x7138d40c7000

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) = 0x7138d3ed6000

mmap(0x7138d3efa000, 1511424, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x24000) = 0x7138d3efa000

mmap(0x7138d406b000, 319488, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x195000) = 0x7138d406b000

mmap(0x7138d40b9000, 24576, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1e3000) = 0x7138d40b9000

mmap(0x7138d40bf000, 31672, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_ANONYMOUS, -1, 0) = 0x7138d40bf000

close(3) = 0

mmap(NULL, 12288, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7138d3ed3000

arch\_prctl(ARCH\_SET\_FS, 0x7138d3ed3740) = 0

set\_tid\_address(0x7138d3ed3a10) = 11295

set\_robust\_list(0x7138d3ed3a20, 24) = 0

rseq(0x7138d3ed4060, 0x20, 0, 0x53053053) = 0

mprotect(0x7138d40b9000, 16384, PROT\_READ) = 0

mprotect(0x5bb372933000, 4096, PROT\_READ) = 0

mprotect(0x7138d413e000, 8192, PROT\_READ) = 0

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

munmap(0x7138d40c9000, 241615) = 0

getrandom("\x18\xbd\x14\x1d\x83\x89\x89\x20", 8, GRND\_NONBLOCK) = 8

brk(NULL) = 0x5bb37f5c9000

brk(0x5bb37f5ea000) = 0x5bb37f5ea000

openat(AT\_FDCWD, "./buddy-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=15288, ...}) = 0

getcwd("/home/arcsenius/CLionProjects/OS\_Labs/lab\_4/src", 128) = 48

mmap(NULL, 16440, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7138d40ff000

mmap(0x7138d4100000, 4096, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1000) = 0x7138d4100000

mmap(0x7138d4101000, 4096, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x2000) = 0x7138d4101000

mmap(0x7138d4102000, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x2000) = 0x7138d4102000

close(3) = 0

mprotect(0x7138d4102000, 4096, PROT\_READ) = 0

mmap(NULL, 1052672, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7138d3dd2000

clock\_gettime(CLOCK\_PROCESS\_CPUTIME\_ID, {tv\_sec=0, tv\_nsec=1799235}) = 0

clock\_gettime(CLOCK\_PROCESS\_CPUTIME\_ID, {tv\_sec=0, tv\_nsec=2953657}) = 0

fstat(1, {st\_mode=S\_IFCHR|0600, st\_rdev=makedev(0x88, 0x4), ...}) = 0

mmap(NULL, 1052672, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7134d31d7300

write(1, "Allocation time for 10000 alloca"..., 56Allocation time for 10000 allocations: 0.001154 seconds

) = 56

clock\_gettime(CLOCK\_PROCESS\_CPUTIME\_ID, {tv\_sec=0, tv\_nsec=2995635}) = 0

clock\_gettime(CLOCK\_PROCESS\_CPUTIME\_ID, {tv\_sec=0, tv\_nsec=185671099}) = 0

write(1, "Free time for 10000 deallocation"..., 52Free time for 10000 deallocations: 0.182676 seconds

) = 52

munmap(0x7138d3dd2000, 1052672) = 0

munmap(0x7138d40ff000, 16440) = 0

exit\_group(0) = ?

+++ exited with 0 +++

**Вывод**

Результаты показали, что аллокатор с фиксированными блоками по 2^n работает быстрее, так как он использует заранее подготовленные пулы памяти, где выделение и освобождение блоков происходит за постоянноевремя 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 фиксированныеблокипо2^n обеспечивают лучшую производительность.