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: Кудаева В.А.

Преподаватель: Бахарев В.Д.

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

Дата: 26.12.24

Москва, 2024

**Постановка задачи**

**Вариант 7:**

Требуется создать две динамические библиотеки, реализующие два аллокатора. Первый аллокатор использует список свободных блоков (наиболее подходящие), второй – блоки по 2^n.

**Общий метод и алгоритм решения**

Использованные системные вызовы:

* void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset) - отображает объект разделяемой памяти в адресное пространство процесса;
* int munmap(void \*addr, size\_t length) - освобождает область памяти, которая была ранее выделена с помощью mmap;
* int write(int fd, const void\* buff, int count); – записывает по дескриптору fd count байт из buff;
* void exit(int number); – вызывает нормальное завершение программы с кодом number;
* void \*dlopen(const char \*filename, int flag) – загружает динамическую библиотеку. Возвращает указатель на дескриптор этой библиотеки;
* int dlclose(void \*handle) - используется для закрытия динамически загруженной библиотеки, освобождая ресурсы, связанные с её загрузкой.

### Алгоритм работы аллокатора с использованием списка свободных блоков:

### 1) **Инициализация аллокатора**:

### При создании аллокатора выделяется область памяти фиксированного размера. В начале всей памяти создается один большой свободный блок, который записывается в список свободных блоков (free list). Этот список содержит указатели на свободные блоки памяти, упорядоченные по возрастанию адресов.

### 2) **Выделение памяти**:

* + Просматривается список свободных блоков, чтобы найти наиболее подходящий блок (то есть минимальный блок, который больше или равен запрашиваемому размеру).
  + Если такой блок найден:
    - Если размер найденного блока больше запрашиваемого, блок разбивается на две части: первая часть выделяется пользователю, а оставшаяся часть возвращается в список свободных блоков.
    - Если блок точно соответствует размеру, он удаляется из списка свободных блоков и передается пользователю.
  + Если подходящего блока нет в списке, функция возвращает NULL, сигнализируя о невозможности выделения памяти.

3) **Освобождение памяти**:

* + Когда пользователь освобождает блок, аллокатор получает указатель на него. Сначала он преобразует этот указатель, чтобы найти метаданные блока.
  + Освобожденный блок добавляется обратно в список свободных блоков. Аллокатор проверяет, есть ли соседние блоки в памяти, которые также свободны. Если такие блоки есть, они сливаются в один большой блок, чтобы уменьшить фрагментацию памяти.

4) **Уничтожение аллокатора**:

* + Аллокатор уничтожается, освобождая всю выделенную память. При этом списки свободных блоков очищаются, а вся область памяти освобождается обратно операционной системе.

Алгоритм работы аллокатора с использованием списка свободных блоков, организованных по степеням двойки (2^n):

1) **Инициализация аллокатора:**

* + При создании аллокатора выделяется большая область памяти фиксированного размера.
  + Список свободных блоков (free lists) инициализируется как массив указателей, где каждый индекс соответствует степени двойки. Вначале вся память представлена одним большим свободным блоком, который помещается в соответствующий список.

2) **Выделение памяти:**

* + Аллокатор находит минимальный размер блока из степеней двойки, который может вместить запрашиваемый размер. Например, если пользователь запрашивает 20 байт, выбирается блок размером 32 байта (2^5).
  + Аллокатор проверяет список свободных блоков для найденного размера:
    - Если свободный блок нужного размера доступен, он извлекается из списка, а его указатель возвращается пользователю.
    - Если блок нужного размера отсутствует, аллокатор ищет в списках для больших размеров ближайший больший блок.
  + Если ни одного подходящего блока не найдено, выделение памяти завершается с ошибкой (NULL).

3) **Освобождение памяти:**

* + Аллокатор определяет размер освобождаемого блока и помещает его обратно в соответствующий список (по степени двойки).

4) **Уничтожение аллокатора:**

* + При уничтожении аллокатора все списки очищаются, а память освобождается обратно операционной системе.

**Код программы**

**allocator.c**

#include <math.h>

#include <stdbool.h>

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/mman.h>

#include <unistd.h>

#define MIN\_BLOCK\_SIZE 16

typedef struct BlockHeader {

    size\_t size;

    struct BlockHeader \*next;

    bool is\_free;

} BlockHeader;

typedef struct Allocator {

    BlockHeader \*free\_list;

    void \*memory\_start;

    size\_t total\_size;

    void \*base\_addr;

} Allocator;

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

    if (!memory || size < sizeof(Allocator)) {

        return NULL;

    }

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

    allocator->base\_addr = memory;

    allocator->memory\_start = (char \*)memory + sizeof(Allocator);

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

    allocator->free\_list = (BlockHeader \*)allocator->memory\_start;

    allocator->free\_list->size = allocator->total\_size - sizeof(BlockHeader);

    allocator->free\_list->next = NULL;

    allocator->free\_list->is\_free = true;

    return allocator;

}

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

    if (!allocator || size == 0) {

        return NULL;

    }

    size = (size + MIN\_BLOCK\_SIZE - 1) / MIN\_BLOCK\_SIZE \*

           MIN\_BLOCK\_SIZE;  // Округление вверх

    BlockHeader \*best\_fit = NULL;

    BlockHeader \*prev\_best = NULL;

    BlockHeader \*current = allocator->free\_list;

    BlockHeader \*prev = NULL;

    while (current) {

        if (current->is\_free && current->size >= size) {

            if (best\_fit == NULL || current->size < best\_fit->size) {

                best\_fit = current;

                prev\_best = prev;

            }

        }

        prev = current;

        current = current->next;

    }

    if (best\_fit) {

        size\_t remaining\_size = best\_fit->size - size;

        if (remaining\_size >= sizeof(BlockHeader) + MIN\_BLOCK\_SIZE) {

            BlockHeader \*new\_block =

                (BlockHeader \*)((char \*)best\_fit + sizeof(BlockHeader) + size);

            new\_block->size = remaining\_size - sizeof(BlockHeader);

            new\_block->is\_free = true;

            new\_block->next = best\_fit->next;

            best\_fit->next = new\_block;

            best\_fit->size = size;

        }

        best\_fit->is\_free = false;

        if (prev\_best == NULL) {

            allocator->free\_list = best\_fit->next;

        } else {

            prev\_best->next = best\_fit->next;

        }

        return (void \*)((char \*)best\_fit + sizeof(BlockHeader));

    }

    return NULL;

}

void allocator\_free(Allocator \*allocator, void \*ptr) {

    if (!allocator || !ptr) {

        return;

    }

    BlockHeader \*header = (BlockHeader \*)((char \*)ptr - sizeof(BlockHeader));

    if (!header) return;

    header->is\_free = true;

    header->next = allocator->free\_list;

    allocator->free\_list = header;

    BlockHeader \*current = allocator->free\_list;

    BlockHeader \*prev = NULL;

    while (current && current->next) {

        BlockHeader \*next = current->next;

        if (((char \*)current + sizeof(BlockHeader) + current->size) ==

            (char \*)next) {

            current->size += next->size + sizeof(BlockHeader);

            current->next = next->next;

            continue;

        }

        if (prev && ((char \*)prev + sizeof(BlockHeader) + prev->size) ==

                        (char \*)current) {

            prev->size += current->size + sizeof(BlockHeader);

            prev->next = current->next;

            current = prev;

            if (allocator->free\_list == current) allocator->free\_list = prev;

            continue;

        }

        prev = current;

        current = current->next;

    }

}

void allocator\_destroy(Allocator \*allocator) {

    if (allocator) {

        munmap(allocator->base\_addr, allocator->total\_size + sizeof(Allocator));

    }

}

**degree.c**

#include <math.h>

#include <stdbool.h>

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/mman.h>

#include <unistd.h>

#define MIN\_BLOCK\_SIZE 16

int log2s(int n) {

    if (n == 0) {

        return -1;

    }

    int result = 0;

    while (n > 1) {

        n >>= 1;

        result++;

    }

    return result;

}

typedef struct BlockHeader {

    struct BlockHeader \*next;

} BlockHeader;

typedef struct Allocator {

    BlockHeader \*\*free\_lists;

    size\_t num\_lists;

    void \*base\_addr;

    size\_t total\_size;

} Allocator;

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

    if (!memory || size < sizeof(Allocator)) {

        return NULL;

    }

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

    allocator->base\_addr = memory;

    allocator->total\_size = size;

    size\_t min\_usable\_size = sizeof(BlockHeader) + MIN\_BLOCK\_SIZE;

    size\_t max\_block\_size = (size < 32) ? 32 : size;

    allocator->num\_lists = (size\_t)floor(log2s(max\_block\_size) / 2) + 3;

    allocator->free\_lists =

        (BlockHeader \*\*)((char \*)memory + sizeof(Allocator));

    for (size\_t i = 0; i < allocator->num\_lists; i++) {

        allocator->free\_lists[i] = NULL;

    }

    void \*current\_block = (char \*)memory + sizeof(Allocator) +

                          allocator->num\_lists \* sizeof(BlockHeader \*);

    size\_t remaining\_size =

        size - sizeof(Allocator) - allocator->num\_lists \* sizeof(BlockHeader \*);

    size\_t block\_size = MIN\_BLOCK\_SIZE;

    while (remaining\_size >= min\_usable\_size) {

        if (block\_size > remaining\_size) {

            break;

        }

        if (block\_size > max\_block\_size) {

            break;

        }

        if (remaining\_size >= (block\_size + sizeof(BlockHeader)) \* 2) {

            for (int i = 0; i < 2; i++) {

                BlockHeader \*header = (BlockHeader \*)current\_block;

                size\_t index = (size == 0) ? 0 : (size\_t)log2s(block\_size);

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

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

                current\_block = (char \*)current\_block + block\_size;

                remaining\_size -= block\_size;

            }

        } else {

            BlockHeader \*header = (BlockHeader \*)current\_block;

            size\_t index = (size == 0) ? 0 : (size\_t)log2s(block\_size);

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

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

            current\_block = (char \*)current\_block + remaining\_size;

            remaining\_size = 0;

        }

        block\_size <<= 1;

    }

    return allocator;

}

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

    if (!allocator || size == 0) {

        return NULL;

    }

    size\_t index = (size == 0) ? 0 : log2s(size) + 1;

    if (index >= allocator->num\_lists) {

        index = allocator->num\_lists;

    }

    bool flag = false;

    if (allocator->free\_lists[index] == NULL) {

        while (index <= allocator->num\_lists) {

            if (allocator->free\_lists[index] != NULL) {

                flag = true;

                break;

            } else {

                ++index;

            }

        }

        if (!flag) return NULL;

    }

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

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

    return (void \*)((char \*)block + sizeof(BlockHeader));

}

void allocator\_free(Allocator \*allocator, void \*ptr) {

    if (!allocator || !ptr) {

        return;

    }

    BlockHeader \*block = (BlockHeader \*)((char \*)ptr - sizeof(BlockHeader));

    size\_t temp\_size =

        (char \*)block + sizeof(BlockHeader) - (char \*)allocator->base\_addr;

    size\_t temp = 32;

    while (temp <= temp\_size) {

        size\_t next\_size = temp << 1;

        if (next\_size > temp\_size) {

            break;

        }

        temp = next\_size;

    }

    size\_t index = (temp\_size == 0) ? 0 : (size\_t)log2s(temp);

    if (index >= allocator->num\_lists) {

        index = allocator->num\_lists - 1;

    }

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

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

}

void allocator\_destroy(Allocator \*allocator) {

    if (allocator) {

        munmap(allocator->base\_addr, allocator->total\_size);

    }

}

**main.c**

#include <dlfcn.h>

#include <math.h>

#include <stddef.h>

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/mman.h>

#include <unistd.h>

typedef struct Allocator {

    void \*(\*allocator\_create)(void \*addr, size\_t size);

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

    void (\*allocator\_free)(void \*allocator, void \*ptr);

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

} Allocator;

void \*standard\_allocator\_create(void \*memory, size\_t size) {

    (void)size;

    (void)memory;

    return memory;

}

void \*standard\_allocator\_alloc(void \*allocator, size\_t size) {

    (void)allocator;

    uint32\_t \*memory =

        mmap(NULL, size + sizeof(uint32\_t), PROT\_READ | PROT\_WRITE,

             MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

    if (memory == MAP\_FAILED) {

        return NULL;

    }

    \*memory = (uint32\_t)(size + sizeof(uint32\_t));

    return memory + 1;

}

void standard\_allocator\_free(void \*allocator, void \*memory) {

    (void)allocator;

    if (memory == NULL) return;

    uint32\_t \*mem = (uint32\_t \*)memory - 1;

    munmap(mem, \*mem);

}

void standard\_allocator\_destroy(void \*allocator) { (void)allocator; }

void load\_allocator(const char \*library\_path, Allocator \*allocator) {

    void \*library = dlopen(library\_path, RTLD\_LOCAL | RTLD\_NOW);

    if (library\_path == NULL || library\_path[0] == '\0' || !library) {

        char message[] = "WARNING: failed to load shared library\n";

        write(STDERR\_FILENO, message, sizeof(message) - 1);

        allocator->allocator\_create = standard\_allocator\_create;

        allocator->allocator\_alloc = standard\_allocator\_alloc;

        allocator->allocator\_free = standard\_allocator\_free;

        allocator->allocator\_destroy = standard\_allocator\_destroy;

        return;

    }

    allocator->allocator\_create = dlsym(library, "allocator\_create");

    allocator->allocator\_alloc = dlsym(library, "allocator\_alloc");

    allocator->allocator\_free = dlsym(library, "allocator\_free");

    allocator->allocator\_destroy = dlsym(library, "allocator\_destroy");

    if (!allocator->allocator\_create || !allocator->allocator\_alloc ||

        !allocator->allocator\_free || !allocator->allocator\_destroy) {

        const char msg[] = "Error: failed to load all allocator functions\n";

        write(STDERR\_FILENO, msg, sizeof(msg) - 1);

        dlclose(library);

        return;

    }

}

int main(int argc, char \*\*argv) {

    const char \*library\_path = (argc > 1) ? argv[1] : NULL;

    Allocator allocator\_api;

    load\_allocator(library\_path, &allocator\_api);

    size\_t size = 4096;

    void \*addr = mmap(NULL, size, PROT\_READ | PROT\_WRITE,

                      MAP\_PRIVATE | MAP\_ANONYMOUS, -1, 0);

    if (addr == MAP\_FAILED) {

        char message[] = "mmap failed\n";

        write(STDERR\_FILENO, message, sizeof(message) - 1);

        return EXIT\_FAILURE;

    }

    void \*allocator = allocator\_api.allocator\_create(addr, size);

    if (!allocator) {

        char message[] = "Failed to initialize allocator\n";

        write(STDERR\_FILENO, message, sizeof(message) - 1);

        munmap(addr, size);

        return EXIT\_FAILURE;

    }

    void \*blocks[12];

    size\_t block\_sizes[12] = {12,  13, 13, 24, 40, 56, 100, 120, 400, 120, 120, 120};

    int alloc\_failed = 0;

    for (int i = 0; i < 12; ++i) {

        blocks[i] = allocator\_api.allocator\_alloc(allocator, block\_sizes[i]);

        if (blocks[i] == NULL) {

            alloc\_failed = 1;

            char alloc\_fail\_message[] = "Memory allocation failed\n";

            write(STDERR\_FILENO, alloc\_fail\_message,

                  sizeof(alloc\_fail\_message) - 1);

            break;

        }

    }

    if (!alloc\_failed) {

        char alloc\_success\_message[] = "Memory allocated successfully\n";

        write(STDOUT\_FILENO, alloc\_success\_message,

              sizeof(alloc\_success\_message) - 1);

        for (int i = 0; i < 12; ++i) {

            char buffer[64];

            snprintf(buffer, sizeof(buffer), "Block %d address: %p\n", i + 1,

                     blocks[i]);

            write(STDOUT\_FILENO, buffer, strlen(buffer));

        }

    }

    for (int i = 0; i < 12; ++i) {

        if (blocks[i] != NULL)

            allocator\_api.allocator\_free(allocator, blocks[i]);

    }

    char free\_message[] = "Memory freed\n";

    write(STDOUT\_FILENO, free\_message, sizeof(free\_message) - 1);

    allocator\_api.allocator\_destroy(allocator);

    char exit\_message[] = "Program exited successfully\n";

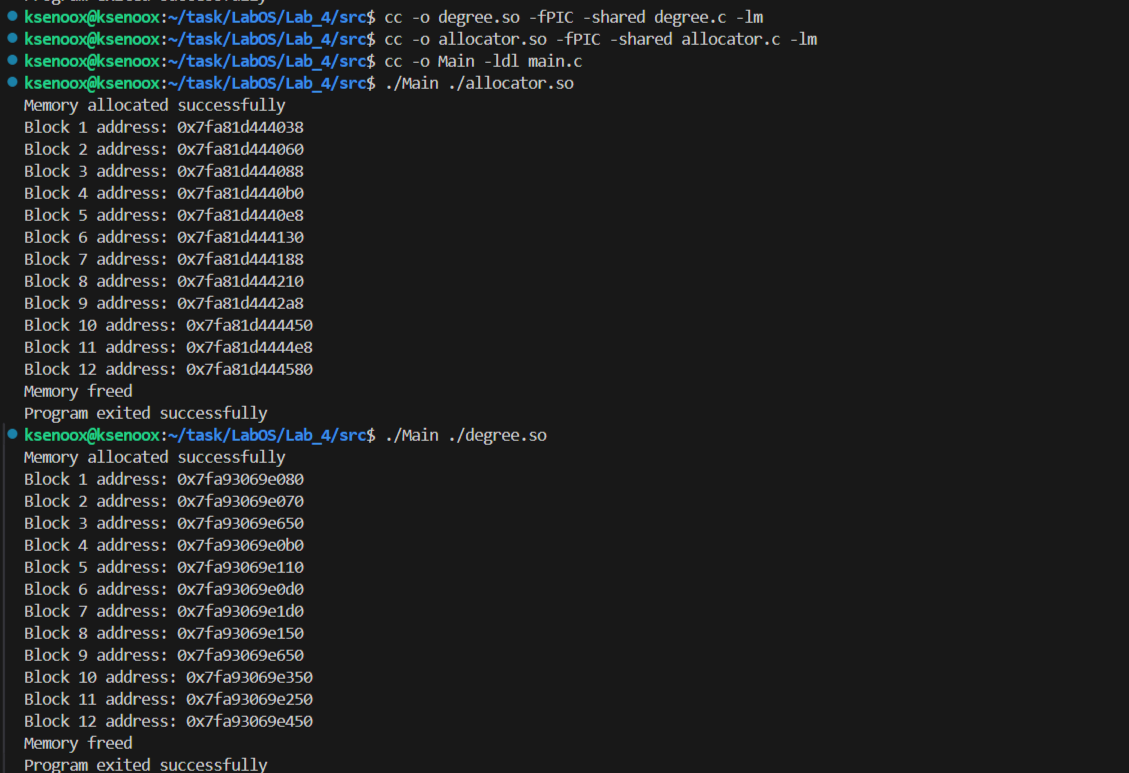
    write(STDOUT\_FILENO, exit\_message, sizeof(exit\_message) - 1);

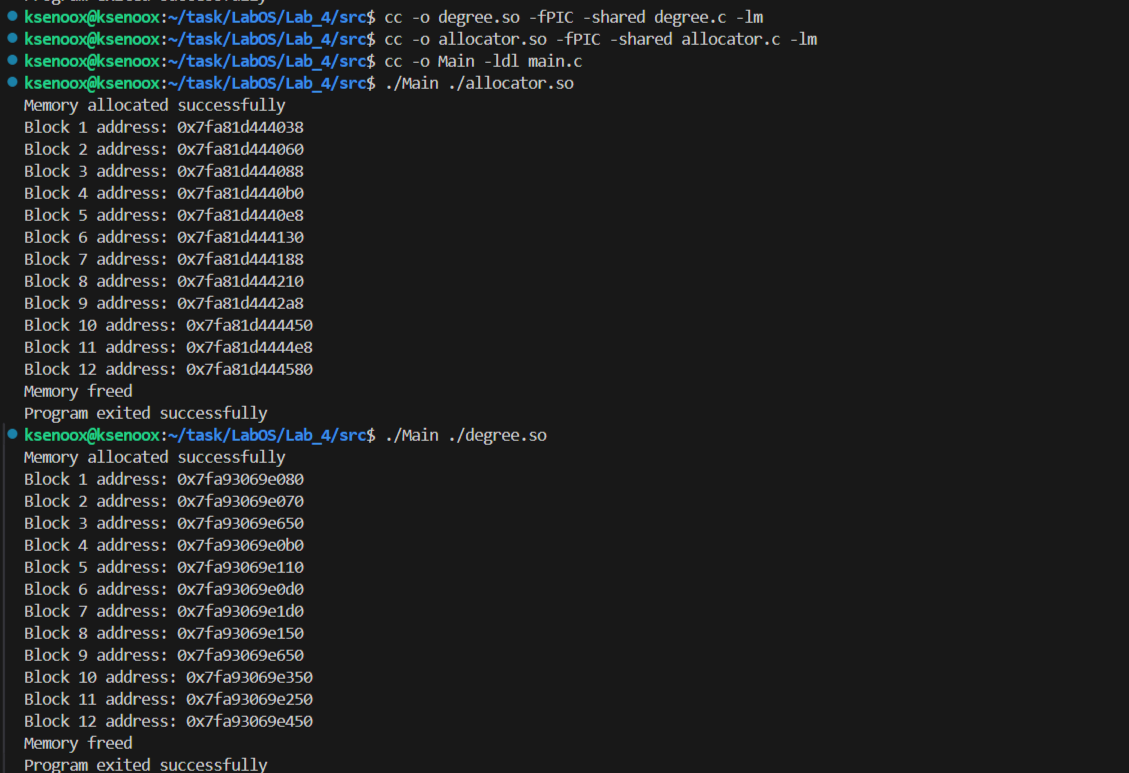
    return EXIT\_SUCCESS;

}

**Протокол работы программы**

**Тестирование**





**Сравнение алгоритмов**

Метод с использованием списков свободных блоков (наиболее подходящий):

1. **Фактор использования памяти:**  
   Высокий, так как выделение памяти происходит строго под необходимый размер, что минимизирует внутреннюю фрагментацию.
2. **Скорость выделения блоков:**  
   Умеренная, поскольку требуется линейный проход по списку свободных блоков для поиска наиболее подходящего блока, особенно при большом количестве фрагментов.
3. **Скорость освобождения блоков:**  
   Быстрая, но дополнительно может потребоваться объединение смежных свободных блоков, что увеличивает сложность операции.
4. **Простота использования аллокатора:**  
   Реализация более сложная из-за необходимости динамического управления списком блоков и их объединения.

Метод с использованием блоков размером 2^n:

1. **Фактор использования памяти:**  
   Средний, так как размеры блоков округляются до ближайшей степени двойки, что приводит к дополнительным потерям памяти из-за внутренней фрагментации.
2. **Скорость выделения блоков:**  
   Высокая, поскольку размер легко сопоставляется с соответствующим списком блоков за O(1).
3. **Скорость освобождения блоков:**  
   Также высокая, из-за предсказуемой структуры и фиксированных размеров.
4. **Простота использования аллокатора:**  
   Реализация проще благодаря фиксированным размерам блоков и предсказуемому управлению памятью.

**Вывод strace**

execve("./Main", ["./Main", "./degree.so"], 0x7ffe9b7052d8 /\* 73 vars \*/) = 0

brk(NULL) = 0x55867fbf8000

arch\_prctl(0x3001 /\* ARCH\_??? \*/, 0x7ffea221e010) = -1 EINVAL (Invalid argument)

mmap(NULL, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fe747341000

access("/etc/ld.so.preload", R\_OK) = -1 ENOENT (No such file or directory)

mmap(NULL, 12288, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fe74710e000

arch\_prctl(ARCH\_SET\_FS, 0x7fe74710e740) = 0

set\_tid\_address(0x7fe74710ea10) = 362413

set\_robust\_list(0x7fe74710ea20, 24) = 0

rseq(0x7fe74710f0e0, 0x20, 0, 0x53053053) = 0

mprotect(0x7fe747327000, 16384, PROT\_READ) = 0

mprotect(0x55866e99b000, 4096, PROT\_READ) = 0

mprotect(0x7fe74737b000, 8192, PROT\_READ) = 0

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

munmap(0x7fe74733a000, 27331) = 0

getrandom("\xc4\xab\x79\xf1\x3e\x52\x42\x4c", 8, GRND\_NONBLOCK) = 8

brk(NULL) = 0x55867fbf8000

brk(0x55867fc19000) = 0x55867fc19000

openat(AT\_FDCWD, "./degree.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

newfstatat(3, "", {st\_mode=S\_IFREG|0755, st\_size=15744, ...}, AT\_EMPTY\_PATH) = 0

getcwd("/home/ksenoox/task/LabOS/Lab\_4/src", 128) = 35

mmap(NULL, 16440, PROT\_READ, MAP\_PRIVATE|MAP\_DENYWRITE, 3, 0) = 0x7fe74733c000

mmap(0x7fe74733d000, 4096, PROT\_READ|PROT\_EXEC, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x1000) = 0x7fe74733d000

mmap(0x7fe74733e000, 4096, PROT\_READ, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x2000) = 0x7fe74733e000

mmap(0x7fe74733f000, 8192, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_FIXED|MAP\_DENYWRITE, 3, 0x2000) = 0x7fe74733f000

close(3) = 0

mprotect(0x7fe74733f000, 4096, PROT\_READ) = 0

mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANONYMOUS, -1, 0) = 0x7fe74737a000

write(1, "Memory allocated successfully\n", 30Memory allocated successfully

) = 30

write(1, "Block 1 address: 0x7fe74737a080\n", 32Block 1 address: 0x7fe74737a080

) = 32

write(1, "Block 2 address: 0x7fe74737a070\n", 32Block 2 address: 0x7fe74737a070

) = 32

write(1, "Block 3 address: 0x7fe74737a650\n", 32Block 3 address: 0x7fe74737a650

) = 32

write(1, "Block 4 address: 0x7fe74737a0b0\n", 32Block 4 address: 0x7fe74737a0b0

) = 32

write(1, "Block 5 address: 0x7fe74737a110\n", 32Block 5 address: 0x7fe74737a110

) = 32

write(1, "Block 6 address: 0x7fe74737a0d0\n", 32Block 6 address: 0x7fe74737a0d0

) = 32

write(1, "Block 7 address: 0x7fe74737a1d0\n", 32Block 7 address: 0x7fe74737a1d0

) = 32

write(1, "Block 8 address: 0x7fe74737a150\n", 32Block 8 address: 0x7fe74737a150

) = 32

write(1, "Block 9 address: 0x7fe74737a650\n", 32Block 9 address: 0x7fe74737a650

) = 32

write(1, "Block 10 address: 0x7fe74737a350"..., 33Block 10 address: 0x7fe74737a350

) = 33

write(1, "Block 11 address: 0x7fe74737a250"..., 33Block 11 address: 0x7fe74737a250

) = 33

write(1, "Block 12 address: 0x7fe74737a450"..., 33Block 12 address: 0x7fe74737a450

) = 33

write(1, "Memory freed\n", 13Memory freed

) = 13

munmap(0x7fe74737a000, 4096) = 0

write(1, "Program exited successfully\n", 28Program exited successfully

) = 28

exit\_group(0) = ?

+++ exited with 0 +++

**Вывод**

В ходе лабораторной работы мне удалось реализовать два аллокатора: один использующий список свободных блоков с выбором наиболее подходящего, а второй – основанный на группировке блоков по степеням двойки. Первый подход обеспечивает более точное использование памяти за счёт минимизации внутренней фрагментации, тогда как второй упрощает управление блоками и ускоряет операции выделения и освобождения.