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

```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);
```

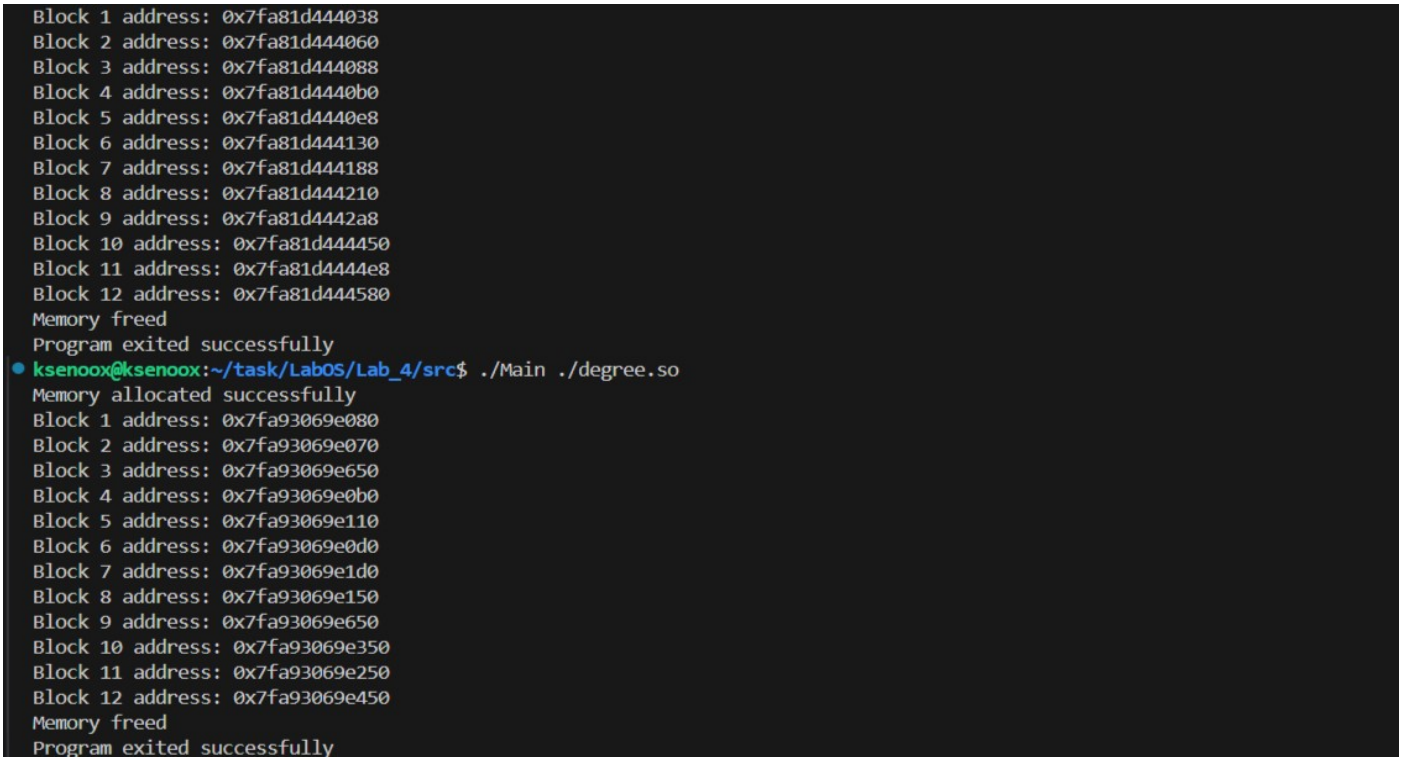
```
        allocator->allocator_create = standard_allocator_create;
        allocator->allocator_alloc = standard_allocator_alloc;
        allocator->allocator_free = standard_allocator_free;
        allocator->allocator_destroy = standard_allocator_destroy;
        return;
    }

    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

```
ksenoox@ksenoox:~/task/LabOS/Lab_4/src$ cc -o degree.so -fPIC -shared degree.c -lm
ksenoox@ksenoox:~/task/LabOS/Lab_4/src$ cc -o allocator.so -fPIC -shared allocator.c -lm
ksenoox@ksenoox:~/task/LabOS/Lab_4/src$ cc -o Main -ldl main.c
ksenoox@ksenoox:~/task/LabOS/Lab_4/src$ ./Main ./allocator.so
Memory allocated successfully
```



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.