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: Лапин Д.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 09.01.25

Москва, 2024

Постановка задачи

Вариант 2.

Списки свободных блоков (первое подходящее) и алгоритм Мак-КьюзиКэрелса;

Общий метод и алгоритм решения

1. Введение

В данной работе реализовано и протестировано два варианта аллокатора памяти:

1. Аллокатор на основе списка свободных блоков (**First Fit**).
2. Аллокатор на основе алгоритма **Мак-Кьюзи–Кэрелса**.

Основная цель — сравнить их по следующим характеристикам:

- Фактор использования памяти.
- Скорость выделения и освобождения блоков.
- Простота использования.

---

2. Описание программы

2.1. Основной модуль: main.c

1. Программа загружает динамические библиотеки с аллокаторами.
2. Если загрузка не удалась, используется fallback-реализация, оборачивающая вызовы `mmap`/`munmap`.
3. Тестирует работу аллокаторов, производя выделение и освобождение памяти.

2.2. Аллокатор на базе First Fit: allocator_first_fit.c

1. **Инициализация**
   - Вся память рассматривается как один большой свободный блок.
2. **Выделение памяти**
   - Применяется принцип *First Fit*: ищется первый свободный блок достаточного размера.
   - Если блок больше, чем требуется, он разделяется на два:
     - один блок отдаётся под запрос,
     - остаток возвращается в список свободных блоков.
3. **Освобождение памяти**
   - Освобождённый блок добавляется обратно в список свободных.
   - Если рядом оказались свободные блоки, они объединяются.

2.3. Аллокатор на базе Мак-Кьюзи–Кэрелса: allocator_mckusick.c

1. **Инициализация**
   - Вся память рассматривается как один большой блок.
2. **Выделение памяти**
   - Ищется свободный блок подходящего размера.
   - Если найденный блок больше требуемого, он разделяется.
3. **Освобождение памяти**
   - Освобождённый блок помечается как свободный.
   - Если соседние блоки свободны, они объединяются (*coalescing*).

3. Используемые системные вызовы

1. **`void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)`**
   - Выделение памяти (применяется в данной работе для обхода ограничений `malloc`).
   - **Параметры:**
     - **start**: Начальный адрес (обычно `NULL`).
     - **length**: Размер выделяемой области.
     - **prot**: Права доступа (например, `PROT_READ | PROT_WRITE`).
     - **flags**: Флаги (`MAP_PRIVATE | MAP_ANONYMOUS`).
     - **fd**: Файловый дескриптор (в данной работе не используется, `-1`).
     - **offset**: Смещение (обычно `0`).
2. **`int munmap(void *start, size_t length)`**
   - Освобождает память, выделенную ранее функцией `mmap`.
   - **Параметры:**
     - **start**: Начальный адрес памяти.
     - **length**: Размер освобождаемой области.
3. **`void *dlopen(const char *filename, int flag)`**
   - Загружает динамическую библиотеку.
   - **Параметры:**
     - **filename**: Путь к библиотеке.
     - **flag**: Флаги загрузки (например, `RTLD_NOW` для немедленной загрузки).
4. **`void *dlsym(void *handle, const char *symbol)`**
   - Получает адрес функции или переменной в загруженной библиотеке.
   - **Параметры:**
     - **handle**: Дескриптор загруженной библиотеки.
     - **symbol**: Имя требуемого символа (функции/переменной).
5. **`int dlclose(void *handle)`**
   - Закрывает ранее загруженную библиотеку.
   - **Параметры:**
     - **handle**: Дескриптор библиотеки.

Код программы

main.c

```c
#include <stdio.h>

#include <stdlib.h>

#include <dlfcn.h>

#include <unistd.h>

#include <sys/mman.h>

#include <time.h>

#include <string.h>
```

```c
typedef struct CustomAllocator CustomAllocator;



typedef CustomAllocator *(*create_t)(void *, size_t);

typedef void (*destroy_t)(CustomAllocator *);

typedef void *(*alloc_t)(CustomAllocator *, size_t);

typedef void (*free_t)(CustomAllocator *, void *);



void *fallback_alloc(CustomAllocator *dummy, size_t sz) {
   (void) dummy;
   return mmap(NULL, sz, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
}

void fallback_free(CustomAllocator *dummy, void *ptr) {
   (void) dummy;
   munmap(ptr, malloc_usable_size(ptr));
}

// Упрощённая функция вывода
static void print_msg(const char *txt) {
   write(STDOUT_FILENO, txt, strlen(txt));
}

void alloc_test(CustomAllocator *alloc, alloc_t allocate, free_t
release) {
   void *chunks[50];
   for (int i = 0; i < 50; i++) {
```

```
        chunks[i] = allocate(alloc, 2048);
        if (!chunks[i]) {
            print_msg("ALLOC TEST: allocation failed at index ");
            char tmp[32];
            snprintf(tmp, sizeof(tmp), "%d\n", i);
            print_msg(tmp);
            return;
        }
    }
    // Освобождаем
    for (int i = 0; i < 50; i++) {
        release(alloc, chunks[i]);
    }
    print_msg("ALLOC TEST: success\n");
}


void merge_test(CustomAllocator *alloc, alloc_t allocate, free_t
release) {
    void *parts[8];
    for (int i = 0; i < 8; i++) {
        parts[i] = allocate(alloc, 1024);
        if (!parts[i]) {
            print_msg("MERGE TEST: allocation failed at index ");
            char buf[16];
            snprintf(buf, sizeof(buf), "%d\n", i);
            print_msg(buf);
            return;
        }
    }
    // Освобождаем в разном порядке
    release(alloc, parts[2]);
    release(alloc, parts[5]);
    release(alloc, parts[0]);
```

```
    release(alloc, parts[6]);

    // Пробуем взять большой блок
    void *large = allocate(alloc, 4096);
    if (large) {
        print_msg("MERGE TEST: passed\n");
        release(alloc, large);
    } else {
        print_msg("MERGE TEST: failed\n");
    }
    // Освободим оставшиеся
    for (int i = 1; i < 8; i++) {
        if (parts[i] != NULL) {
            release(alloc, parts[i]);
        }
    }
}

void speed_test(CustomAllocator *alloc, alloc_t allocate, free_t
release) {
    clock_t t1 = clock();
    for (int i = 0; i < 2000; i++) {
        void *mem = allocate(alloc, 256);
        if (mem) {
            release(alloc, mem);
        }
    }
    clock_t t2 = clock();
    double total_time = (double) (t2 - t1) / CLOCKS_PER_SEC;

    char msg[64];
    snprintf(msg, sizeof(msg), "SPEED TEST: done in %.6f s\n",
total_time);
```

```
    print_msg(msg);
}


void fragmentation_test(CustomAllocator *alloc, alloc_t allocate,
free_t release) {
    void *segments[12];
    for (int i = 0; i < 12; i++) {
        segments[i] = allocate(alloc, 512);
        if (!segments[i]) {
            print_msg("FRAGMENTATION TEST: allocation failed\n");
            return;
        }
    }
    // Освобождаем выборочно
    release(alloc, segments[3]);
    release(alloc, segments[7]);
    release(alloc, segments[10]);

    // Проверим выделение более крупного куска
    void *big = allocate(alloc, 1536);
    if (big) {
        print_msg("FRAGMENTATION TEST: passed\n");
        release(alloc, big);
    } else {
        print_msg("FRAGMENTATION TEST: failed\n");
    }
    // Освободим остальные
    for (int i = 0; i < 12; i++) {
        if (segments[i] != NULL) {
            release(alloc, segments[i]);
        }
    }
}
```

```
int main(int argc, char **argv) {
   void *lib_handle = NULL;
   create_t fn_create = NULL;
   destroy_t fn_destroy = NULL;
   alloc_t fn_alloc = NULL;
   free_t fn_free = NULL;

   // Если в аргументах указан .so/.dll — пробуем открыть
   if (argc > 1) {
       lib_handle = dlopen(argv[1], RTLD_NOW);
       if (lib_handle) {
           fn_create = (create_t) dlsym(lib_handle,
"allocator_create");
           fn_destroy = (destroy_t) dlsym(lib_handle,
"allocator_destroy");
           fn_alloc = (alloc_t) dlsym(lib_handle,
"allocator_alloc");
           fn_free = (free_t) dlsym(lib_handle, "allocator_free");
       }
   }

   // Если не удалось — используем fallback-аллокатор
   if (!lib_handle || !fn_create || !fn_destroy || !fn_alloc ||
!fn_free) {
       print_msg("No custom library. Using fallback
allocator.\n");
       fn_create = (create_t) malloc;
       fn_destroy = free;
       fn_alloc = fallback_alloc;
       fn_free = fallback_free;
   }

   // Выделим область под наш «CustomAllocator» (1 МБ)
   void *region = mmap(NULL, 1024 * 1024,
```

```
                        PROT_READ | PROT_WRITE,
                        MAP_PRIVATE | MAP_ANONYMOUS,
                        -1, 0);

    CustomAllocator *my_allocator = fn_create(region, 1024 * 1024);

    // Запускаем тесты
    alloc_test(my_allocator, fn_alloc, fn_free);
    merge_test(my_allocator, fn_alloc, fn_free);
    speed_test(my_allocator, fn_alloc, fn_free);
    fragmentation_test(my_allocator, fn_alloc, fn_free);

    // Завершаем работу
    fn_destroy(my_allocator);
    munmap(region, 1024 * 1024);

    if (lib_handle) {
        dlclose(lib_handle);
    }

    return 0;
}
```

allocator_first_fit.c

```
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
```

```c
typedef struct Chunk {
    size_t length;
    struct Chunk* next;
} Chunk;

typedef struct {
    Chunk* free_list;
} AllocFF;

// Внешний интерфейс
AllocFF* allocator_create(void* mem, size_t sz) {
    AllocFF* a = (AllocFF*)mem;
    a->free_list = (Chunk*)((char*)mem + sizeof(AllocFF));
    a->free_list->length = sz - sizeof(AllocFF) - sizeof(Chunk);
    a->free_list->next = NULL;
    return a;
}

void allocator_destroy(AllocFF* a) {
    (void)a; // Ничего не делаем
}

void* allocator_alloc(AllocFF* a, size_t needed) {
    Chunk* prev = NULL;
    Chunk* cur = a->free_list;

    while (cur) {
        if (cur->length >= needed) {
            // Можно разместить
            if (cur->length - needed > sizeof(Chunk)) {
                // Разделяем блок
                char* split_addr = (char*)cur + sizeof(Chunk) +
needed;
```

```
                Chunk* new_chunk = (Chunk*)split_addr;
                new_chunk->length = cur->length - needed -
sizeof(Chunk);
                new_chunk->next = cur->next;

                // Текущему уменьшаем length
                cur->length = needed;

                // Вставляем new_chunk в список
                if (prev) {
                    prev->next = new_chunk;
                } else {
                    a->free_list = new_chunk;
                }
            } else {
                // Берём целиком
                if (prev) {
                    prev->next = cur->next;
                } else {
                    a->free_list = cur->next;
                }
            }
            return (char*)cur + sizeof(Chunk);
        }
        // Двигаемся дальше
        prev = cur;
        cur  = cur->next;
    }
    // Не нашли подходящий блок
    return NULL;
}

void allocator_free(AllocFF* a, void* mem) {
```

```
    if (!mem) return;
    Chunk* freed = (Chunk*)((char*)mem - sizeof(Chunk));

    // Кладём освобождённый блок в начало списка
    freed->next = a->free_list;
    a->free_list = freed;
}
```

allocator_mckusick.c

```
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct {
    size_t length;
    int free_flag;
} Chunk;

typedef struct {
    Chunk* first_chunk;
    size_t total_capacity;
} AllocatorMcKusick;

AllocatorMcKusick* allocator_create(void* region, size_t
region_size) {
    AllocatorMcKusick* handle = (AllocatorMcKusick*)region;
    handle->first_chunk = (Chunk*)((char*)region +
sizeof(AllocatorMcKusick));
    handle->first_chunk->length = region_size -
sizeof(AllocatorMcKusick) - sizeof(Chunk);
    handle->first_chunk->free_flag = 1;
    handle->total_capacity = region_size;
```

```
    return handle;
}


void allocator_destroy(AllocatorMcKusick* handle) {
    (void)handle;
}


void* allocator_alloc(AllocatorMcKusick* handle, size_t user_size)
{
    Chunk* curr = handle->first_chunk;
    while ((char*)curr < (char*)handle->first_chunk +
handle->total_capacity) {
        if (curr->free_flag && curr->length >= user_size) {
            if (curr->length - user_size > sizeof(Chunk)) {
                char* split_addr = (char*)curr + sizeof(Chunk) +
user_size;
                Chunk* new_chunk = (Chunk*)split_addr;


                new_chunk->length = curr->length - user_size -
sizeof(Chunk);
                new_chunk->free_flag = 1;
                curr->length = user_size;
            }
            curr->free_flag = 0;
            return (char*)curr + sizeof(Chunk);
        }
        curr = (Chunk*)((char*)curr + sizeof(Chunk) +
curr->length);
    }
    return NULL;
}


void allocator_free(AllocatorMcKusick* handle, void* ptr) {
    if (!ptr) return;
```

```
    Chunk* blk = (Chunk*)((char*)ptr - sizeof(Chunk));
    blk->free_flag = 1;


    Chunk* curr = handle->first_chunk;
    while ((char*)curr < (char*)handle->first_chunk +
handle->total_capacity) {
        char* next_addr = (char*)curr + sizeof(Chunk) +
curr->length;
        if (next_addr >= (char*)handle->first_chunk +
handle->total_capacity) {
            break;
        }
        Chunk* nxt = (Chunk*)next_addr;
        if (curr->free_flag && nxt->free_flag) {
            curr->length += sizeof(Chunk) + nxt->length;
            continue;
        }
        curr = nxt;
    }
}
```

Протокол работы программы

1. **Массовое выделение и освобождение памяти**:

   Выделение памяти разного размера с последующим освобождением.

   Цель: Проверить, как аллокатор справляется с множественными запросами на выделение и освобождение памяти.

2. **Проверка объединения блоков**:

   Выделение нескольких блоков и освобождение их в произвольном порядке.

   Цель: Проверить корректность объединения свободных блоков.

3. **Измерение производительности**:

   Сравнение времени выполнения операций выделения и освобождения памяти.

   Цель: Оценить скорость работы аллокаторов.

4. **Измерение фрагментации**:

Оценка степени использования памяти.

Цель: Определить, насколько эффективно аллокатор использует память.

Обоснование подхода тестирования

Тесты разработаны для проверки следующих характеристик:

1. **Эффективность выделения памяти**:

Важно для приложений, интенсивно использующих динамическую память.

Проверяется путем массового выделения и освобождения памяти.

2. **Корректность работы**:

Объединение блоков и освобождение должны работать без ошибок.

Проверяется путем освобождения блоков в произвольном порядке.

3. **Производительность**:

Аллокатор должен минимизировать накладные расходы.

Проверяется путем измерения времени выполнения операций.

4. **Фрагментация**:

Важно для долгосрочной работы без истощения памяти.

Проверяется путем оценки степени использования памяти.

Результаты тестирования

Тест 1: Массовое выделение и освобождение памяти

· **Цель**: Проверить, как аллокатор справляется с множественными запросами.

· **Шаги**:

1. Выделить 100 блоков памяти разного размера.
2. Освободить все блоки.

· **Ожидаемый результат**: Все блоки успешно выделяются и освобождаются.

Тест 2: Проверка объединения блоков

· **Цель**: Проверить корректность объединения свободных блоков.

· **Шаги**:

1. Выделить 10 блоков памяти.
2. Освободить блоки в произвольном порядке.
3. Попытаться выделить новый блок, который требует объединения свободных блоков.

· **Ожидаемый результат**: Аллокатор корректно объединяет свободные блоки и выделяет память.

Тест 3: Измерение производительности

· **Цель**: Оценить скорость работы аллокаторов.

· **Шаги**:

1. Запустить цикл, в котором многократно выделяется и освобождается память.
2. Замерить время выполнения операций.

· **Ожидаемый результат**: Получить данные о производительности каждого аллокатора.

Тест 4: Измерение фрагментации

· **Цель**: Оценить степень использования памяти.

· **Шаги**:

1. Выделить несколько блоков памяти.
2. Освободить некоторые из них.
3. Оценить, насколько эффективно используется память.

· **Ожидаемый результат**: Определить уровень фрагментации.

```
    void fragmentation_test(CustomAllocator *alloc, alloc_t allocate, free_t release)
{
        void *segments[12];
        for (int i = 0; i < 12; i++) {
            segments[i] = allocate(alloc, 512);
            if (!segments[i]) {
                print_msg("FRAGMENTATION TEST: allocation failed\n");
                return;
            }
        }
        // Освобождаем выборочно
```

```
    release(alloc, segments[3]);
    release(alloc, segments[7]);
    release(alloc, segments[10]);

    // Проверим выделение более крупного куска
    void *big = allocate(alloc, 1536);
    if (big) {
        print_msg("FRAGMENTATION TEST: passed\n");
        release(alloc, big);
    } else {
        print_msg("FRAGMENTATION TEST: failed\n");
    }
    // Освободим остальные
    for (int i = 0; i < 12; i++) {
        if (segments[i] != NULL) {
            release(alloc, segments[i]);
        }
    }
}
```

```
void speed_test(CustomAllocator *alloc, alloc_t allocate, free_t release) {
    clock_t t1 = clock();
    for (int i = 0; i < 2000; i++) {
        void *mem = allocate(alloc, 256);
        if (mem) {
            release(alloc, mem);
        }
    }
    clock_t t2 = clock();
    double total_time = (double) (t2 - t1) / CLOCKS_PER_SEC;

    char msg[64];
    snprintf(msg, sizeof(msg), "SPEED TEST: done in %.6f s\n", total_time);
    print_msg(msg);
}
```

```
void merge_test(CustomAllocator *alloc, alloc_t allocate, free_t release) {
    void *parts[8];
    for (int i = 0; i < 8; i++) {
        parts[i] = allocate(alloc, 1024);
        if (!parts[i]) {
            print_msg("MERGE TEST: allocation failed at index ");
            char buf[16];
            snprintf(buf, sizeof(buf), "%d\n", i);
            print_msg(buf);
            return;
        }
    }
    // Освобождаем в разном порядке
    release(alloc, parts[2]);
    release(alloc, parts[5]);
    release(alloc, parts[0]);
    release(alloc, parts[6]);

    // Пробуем взять большой блок
    void *large = allocate(alloc, 4096);
    if (large) {
        print_msg("MERGE TEST: passed\n");
        release(alloc, large);
    } else {
        print_msg("MERGE TEST: failed\n");
    }
    // Освободим оставшиеся
    for (int i = 1; i < 8; i++) {
        if (parts[i] != NULL) {
            release(alloc, parts[i]);
        }
    }
}
void alloc_test(CustomAllocator *alloc, alloc_t allocate, free_t release) {
    void *chunks[50];
    for (int i = 0; i < 50; i++) {
```

```
            chunks[i] = allocate(alloc, 2048);
            if (!chunks[i]) {
                print_msg("ALLOC TEST: allocation failed at index ");
                char tmp[32];
                snprintf(tmp, sizeof(tmp), "%d\n", i);
                print_msg(tmp);
                return;
            }
        }
        // Освобождаем
        for (int i = 0; i < 50; i++) {
            release(alloc, chunks[i]);
        }
        print_msg("ALLOC TEST: success\n");
    }
```

```
qbzy@QBZstation:/mnt/c/Users/mrbor/CLionProjects/osi/lab4$ ./main
./allocator_mckusick.so

ALLOC TEST: success

MERGE TEST: passed

SPEED TEST: done in 0.000047 s

FRAGMENTATION TEST: passed

qbzy@QBZstation:/mnt/c/Users/mrbor/CLionProjects/osi/lab4$ ./main
./allocator_first_fit.so

ALLOC TEST: success

MERGE TEST: passed

SPEED TEST: done in 0.000022 s

FRAGMENTATION TEST: passed
```

Strace:

```
152442 execve("./main", ["./main", "./allocator_first_fit.so"],
0x7ffc15979e90 /* 26 vars */) = 0

152442 brk(NULL)                        = 0x56051b32c000
```

```
    152442 mmap(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fab01dea000

    152442 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such
file or directory)

    152442 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) =
3

    152442 fstat(3, {st_mode=S_IFREG|0644, st_size=21363, ...}) = 0

    152442 mmap(NULL, 21363, PROT_READ, MAP_PRIVATE, 3, 0) =
0x7fab01de4000

    152442 close(3)                         = 0

    152442 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6",
O_RDONLY|O_CLOEXEC) = 3

    152442 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"..
., 832) = 832

    152442 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

    152442 fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0

    152442 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

    152442 mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE,
3, 0) = 0x7fab01bd2000

    152442 mmap(0x7fab01bfa000, 1605632, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7fab01bfa000

    152442 mmap(0x7fab01d82000, 323584, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x7fab01d82000

    152442 mmap(0x7fab01dd1000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x7fab01dd1000

    152442 mmap(0x7fab01dd7000, 52624, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fab01dd7000

    152442 close(3)                         = 0

    152442 mmap(NULL, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fab01bcf000

    152442 arch_prctl(ARCH_SET_FS, 0x7fab01bcf740) = 0

    152442 set_tid_address(0x7fab01bcfa10)  = 152442

    152442 set_robust_list(0x7fab01bcfa20, 24) = 0

    152442 rseq(0x7fab01bd0060, 0x20, 0, 0x53053053) = 0

    152442 mprotect(0x7fab01dd1000, 16384, PROT_READ) = 0
```

```
    152442 mprotect(0x5604fcacc000, 4096, PROT_READ) = 0

    152442 mprotect(0x7fab01e22000, 8192, PROT_READ) = 0

    152442 prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0

    152442 munmap(0x7fab01de4000, 21363)    = 0

    152442 getrandom("\x37\xb8\x20\xf6\xe3\x31\x08\x8f", 8,
GRND_NONBLOCK) = 8

    152442 brk(NULL)                        = 0x56051b32c000

    152442 brk(0x56051b34d000)              = 0x56051b34d000

    152442 openat(AT_FDCWD, "./allocator_first_fit.so",
O_RDONLY|O_CLOEXEC) = 3

    152442 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

    152442 fstat(3, {st_mode=S_IFREG|0777, st_size=15256, ...}) = 0

    152442 getcwd("/mnt/c/Users/mrbor/CLionProjects/osi/lab4", 128) =
42

    152442 mmap(NULL, 16400, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3,
0) = 0x7fab01de5000

    152442 mmap(0x7fab01de6000, 4096, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7fab01de6000

    152442 mmap(0x7fab01de7000, 4096, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7fab01de7000

    152442 mmap(0x7fab01de8000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7fab01de8000

    152442 close(3)                         = 0

    152442 mprotect(0x7fab01de8000, 4096, PROT_READ) = 0

    152442 mmap(NULL, 1048576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fab01acf000

    152442 write(1, "ALLOC TEST: success\n", 20) = 20

    152442 write(1, "MERGE TEST: passed\n", 19) = 19

    152442 clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0,
tv_nsec=3461600}) = 0

    152442 clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0,
tv_nsec=3518600}) = 0

    152442 write(1, "SPEED TEST: done in 0.000057 s\n", 31) = 31

    152442 write(1, "FRAGMENTATION TEST: passed\n", 27) = 27

    152442 munmap(0x7fab01acf000, 1048576)  = 0
```

```
    152442 munmap(0x7fab01de5000, 16400)    = 0

    152442 exit_group(0)                    = ?

    152442 +++ exited with 0 +++
```

```
    152694 execve("./main", ["./main", "./allocator_mckusick.so"],
0x7ffd72be0550 /* 26 vars */) = 0

    152694 brk(NULL)                        = 0x55a03f077000

    152694 mmap(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f4e55af2000

    152694 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such
file or directory)

    152694 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) =
3

    152694 fstat(3, {st_mode=S_IFREG|0644, st_size=21363, ...}) = 0

    152694 mmap(NULL, 21363, PROT_READ, MAP_PRIVATE, 3, 0) =
0x7f4e55aec000

    152694 close(3)                         = 0

    152694 openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6",
O_RDONLY|O_CLOEXEC) = 3

    152694 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

    152694 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

    152694 fstat(3, {st_mode=S_IFREG|0755, st_size=2125328, ...}) = 0

    152694 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

    152694 mmap(NULL, 2170256, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE,
3, 0) = 0x7f4e558da000

    152694 mmap(0x7f4e55902000, 1605632, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f4e55902000

    152694 mmap(0x7f4e55a8a000, 323584, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0x7f4e55a8a000

    152694 mmap(0x7f4e55ad9000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1fe000) = 0x7f4e55ad9000

    152694 mmap(0x7f4e55adf000, 52624, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f4e55adf000
```

```
    152694 close(3)                          = 0

    152694 mmap(NULL, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f4e558d7000

    152694 arch_prctl(ARCH_SET_FS, 0x7f4e558d7740) = 0

    152694 set_tid_address(0x7f4e558d7a10)  = 152694

    152694 set_robust_list(0x7f4e558d7a20, 24) = 0

    152694 rseq(0x7f4e558d8060, 0x20, 0, 0x53053053) = 0

    152694 mprotect(0x7f4e55ad9000, 16384, PROT_READ) = 0

    152694 mprotect(0x55a039085000, 4096, PROT_READ) = 0

    152694 mprotect(0x7f4e55b2a000, 8192, PROT_READ) = 0

    152694 prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
rlim_max=RLIM64_INFINITY}) = 0

    152694 munmap(0x7f4e55aec000, 21363)    = 0

    152694 getrandom("\x93\xaa\x2e\x18\xc6\x9a\x28\x8d", 8,
GRND_NONBLOCK) = 8

    152694 brk(NULL)                         = 0x55a03f077000

    152694 brk(0x55a03f098000)               = 0x55a03f098000

    152694 openat(AT_FDCWD, "./allocator_mckusick.so",
O_RDONLY|O_CLOEXEC) = 3

    152694 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

    152694 fstat(3, {st_mode=S_IFREG|0777, st_size=15256, ...}) = 0

    152694 getcwd("/mnt/c/Users/mrbor/CLionProjects/osi/lab4", 128) =
42

    152694 mmap(NULL, 16400, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3,
0) = 0x7f4e55aed000

    152694 mmap(0x7f4e55aee000, 4096, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x7f4e55aee000

    152694 mmap(0x7f4e55aef000, 4096, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f4e55aef000

    152694 mmap(0x7f4e55af0000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7f4e55af0000

    152694 close(3)                          = 0

    152694 mprotect(0x7f4e55af0000, 4096, PROT_READ) = 0

    152694 mmap(NULL, 1048576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f4e557d7000

    152694 write(1, "ALLOC TEST: success\n", 20) = 20
```

```
    152694 write(1, "MERGE TEST: passed\n", 19) = 19

    152694 clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0,
tv_nsec=4649300}) = 0

    152694 clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0,
tv_nsec=4741500}) = 0

    152694 write(1, "SPEED TEST: done in 0.000092 s\n", 31) = 31

    152694 write(1, "FRAGMENTATION TEST: passed\n", 27) = 27

    152694 munmap(0x7f4e557d7000, 1048576)  = 0

    152694 munmap(0x7f4e55aed000, 16400)    = 0

    152694 exit_group(0)                    = ?

    152694 +++ exited with 0 +++
```

Вывод

В процессе выполнения этой лабораторной работы я освоил работу с динамическими библиотеками, новыми системными вызовами, предназначенными для работы с динамическими библиотеками, и написанием собственного аллокатора памяти в языке C. Я научился писать собственные динамические библиотеки, подключать, обрабатывать ошибки, связанные с их подключением, и использовать их. Главная сложность работы возникла при написании собственного аллокатора памяти, поскольку материал был новый для меня и информацию про алгоритмы аллокаторов приходилась искать в книгах и интернете.