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: Юрченко А.Н.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 08.01.25

Москва, 2025

Постановка задачи

Вариант 9.

Исследовать два аллокатора памяти: необходимо реализовать два алгоритма аллокации памяти (списки свободных блоков (наиболее подходящее и алгоритм двойников) и сравнить их по следующим характеристикам:

- фактор использования

- скорость выделения блоков

- скорость освобождения блоков

- простота использования аллокатора

Общий метод и алгоритм решения

Использованные системные вызовы:

- void *memset(void *buf, int ch, size_t count) - копирует младший байт ch в первые count символов массива, на который указывает buf.
- void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset) - функция mmap отражает length байтов, начиная со смещения offset файла, определенного файловым описателем fd, в память, начиная с адреса start.
- int munmap(void *start, size_t length) - системный вызов munmap удаляет все отражения из заданной области памяти, после чего все ссылки на данную область будут вызывать ошибку "неправильное обращение к памяти" (invalid memory reference).
- void *dlopen(const char *filename, int flag) - загружает динамическую библиотеку, имя которой указано в строке filename, и возвращает прямой указатель на начало динамической библиотеки.
- void *dlsym(void *handle, char *symbol) - использует указатель на динамическую библиотеку, возвращаемую dlopen, и оканчивающееся нулем символьное имя, а затем возвращает адрес, указывающий, откуда загружается этот символ.
- int dlclose(void *handle) - уменьшает на единицу счетчик ссылок на указатель динамической библиотеки handle.

Правила со списками свободных блоков (наиболее подходящее)

1. при создании вся память - это один большой свободный блок

2. размер блоков произвольный

3. ищем наиболее подходящий по размеру, и если он больше необходимого, то можно его разбить еще

4. при освобождении памяти свободные соседнии блоки "складываем"

5. если размер блока меньше минимального, выделяется минимально допустимый блок

Правила с алгоритмом двойников

1. все выделяемые блоки имеют размер в степень двойки;

2. размер изначального единого свободного блока (при старте программы) тоже равен степени 2;

Когда требуется выделить блок размера S:

находим минимальную степень 2 больше S;

проверяем наличие свободных страниц такого размера, отдаём если есть;

если нет, находим минимальный блок большего размера и рекурсивно расщепляем его до нужного размера.

Получившиеся “обрезки” пополняют списки свободных блоков.

При освобождении блока:

находим размер блока;

проверяем, свободен ли второй блок (buddy) и если свободен, объединяем их и рекурсивно проверяем

вышестоящий (уже объединённый) блок.

Код программы

library.h

```
#ifndef ALLOCATOR_H
#define ALLOCATOR_H

#include <dlfcn.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <string.h>
#include <stdbool.h>

typedef struct Allocator Allocator;
typedef struct Block Block;

typedef Allocator *allocator_create_f(void *const memory, const size_t size);
typedef void allocator_destroy_f(Allocator *const allocator);
typedef void *allocator_alloc_f(Allocator *const allocator, const size_t size);
typedef void allocator_free_f(Allocator *const allocator, void *const memory);

#endif
```

list.c

```
#include "library.h"
#define MIN_BLOCK_SIZE 32

typedef struct Block {
    size_t size;
```

```
    struct Block *next;
    bool is_free;
} Block;

typedef struct Allocator {
    Block *free_list_start_from;
    void *memory_start_from;
    size_t general_size;
} Allocator;

Allocator *allocator_create(void *memory, size_t size) {
    if (!memory || size < sizeof(Allocator)) {
        return NULL;
    }

    Allocator *allocator = (Allocator *)memory;

    allocator->memory_start_from = (char *)memory + sizeof(Allocator);
    allocator->general_size = size - sizeof(Allocator);
    allocator->free_list_start_from = (Block *)allocator->memory_start_from;

    allocator->free_list_start_from->size = allocator->general_size - sizeof(Block);
    allocator->free_list_start_from->next = NULL;
    allocator->free_list_start_from->is_free = true;

    return allocator;
}
```

```
void allocator_destroy(Allocator *allocator) {
    if (allocator) {
        memset(allocator, 0, allocator->general_size);
    }
}

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;
    Block *best = NULL;
    Block *prev_best = NULL;
    Block *current = allocator->free_list_start_from;
    Block *prev = NULL;

    while (current) {
        if (current->is_free && current->size >= size) {
            if (best == NULL || current->size < best->size) {
                best = current;
                prev_best = prev;
            }
        }
        prev = current;
        current = current->next;
```

```
    }

    if (best) {
        size_t remain_size = best->size - size;
        if (remain_size >= sizeof(Block) + MIN_BLOCK_SIZE) {
                Block *new_block = (Block *)((char *)best + sizeof(Block) + size);
                new_block->size = remain_size - sizeof(Block);
                new_block->is_free = true;
                new_block->next = best->next;

                best->next = new_block;
                best->size = size;
            }

        best->is_free = false;

        if (prev_best == NULL) {
            allocator->free_list_start_from = best->next;
        } else {
            prev_best->next = best->next;
        }

        return (void *)((char *)best + sizeof(Block));
    }

    return NULL;
}
```

```
void allocator_free(Allocator *allocator, void *ptr_to_memory) {
    if (!allocator || !ptr_to_memory) {
        return;
    }

    Block *head = (Block *)((char *)ptr_to_memory - sizeof(Block));

    head->next = allocator->free_list_start_from;
    head->is_free = true;
    allocator->free_list_start_from = head;

    Block *current = allocator->free_list_start_from;

    while (current && current->next) {
        if (((char *)current + sizeof(Block) + current->size) == (char *)current->next) {
            current->size += current->next->size + sizeof(Block);
            current->next = current->next->next;
        } else {
            current = current->next;
        }
    }
}
```

twins.c

```
#include "library.h"

#define MIN_BLOCK_SIZE 32
```

```
typedef struct Block {
    size_t size;
    struct Block *next;
    bool is_free;
} Block;

typedef struct Allocator {
    Block *free_list_start_from;
    void *memory_start_from;
    size_t general_size;
} Allocator;

Allocator *allocator_create(void *memory, size_t size) {
    if (!memory || size < sizeof(Allocator)) {
        return NULL;
    }

    Allocator *allocator = (Allocator *)memory;

    allocator->memory_start_from = (char *)memory + sizeof(Allocator);
    allocator->general_size = size - sizeof(Allocator);
    allocator->free_list_start_from = (Block *)allocator->memory_start_from;

    allocator->free_list_start_from->size = allocator->general_size - sizeof(Block);
    allocator->free_list_start_from->next = NULL;
    allocator->free_list_start_from->is_free = true;
```

```
    return allocator;
}

void allocator_destroy(Allocator *allocator) {
    if (allocator) {
        memset(allocator, 0, allocator->general_size);
    }
}

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;
    Block *current = allocator->free_list_start_from;

    while (current) {
        if (current->is_free && current->size >= size) {
            if (current->size >= size + sizeof(Block) + MIN_BLOCK_SIZE) {
                Block *new_block = (Block *)((char *)current + sizeof(Block) + size);
                new_block->size = current->size - size - sizeof(Block);
                new_block->is_free = true;
                new_block->next = current->next;

                current->next = new_block;
```

```
            current->size = size;
        }

        current->is_free = false;
        if (current == allocator->free_list_start_from) {
            allocator->free_list_start_from = current->next;
        } else {
            Block *prev = allocator->free_list_start_from;
            while (prev && prev->next != current) {
                prev = prev->next;
            }
            if (prev) {
                prev->next = current->next;
            }
        }

        return (void *)((char *)current + sizeof(Block));
    }
    current = current->next;
  }

  return NULL;
}

void allocator_free(Allocator *allocator, void *ptr_to_memory) {
  if (!allocator || !ptr_to_memory) {
    return;
```

```
}

Block *block_to_free = (Block *)((char *)ptr_to_memory - sizeof(Block));
block_to_free->is_free = true;

Block *current = allocator->free_list_start_from;

if (block_to_free < current) {
    block_to_free->next = current;
    allocator->free_list_start_from = block_to_free;
    current = block_to_free;
} else {
    while (current && current->next && current->next < block_to_free) {
        current = current->next;
    }
    block_to_free->next = current->next;
    current->next = block_to_free;
}

if (current && current->is_free) {
    current->size += block_to_free->size + sizeof(Block);
    current->next = block_to_free->next;
    block_to_free = current;
}

if (block_to_free->next && block_to_free->next->is_free) {
    block_to_free->size += block_to_free->next->size + sizeof(Block);
```

```
            block_to_free->next = block_to_free->next->next;
        }
    }
```

main.c

```
#include "library.h"

#define MY_MEMORY_SIZE 1024

static Allocator *allocator_create_other(void *const memory, const size_t size) {
    const char mes[] = "We use mmap\n";
    write(STDERR_FILENO, mes, sizeof(mes) - 1);

    void *new_memory = mmap(memory, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS | MAP_FIXED, -1, 0);
    if (new_memory == MAP_FAILED) {
        const char err[] = "mmap failed 1\n";
        write(STDERR_FILENO, err, sizeof(err) - 1);
        return NULL;
    }

    return (Allocator *)new_memory;
}

static void allocator_destroy_other(Allocator *const allocator) {
    const char mes[] = "We use munmap\n";
    write(STDERR_FILENO, mes, sizeof(mes) - 1);

    if (allocator) {
```

```
		if (munmap(allocator, MY_MEMORY_SIZE) == -1) {
			const char err[] = "munmap failed 2\n";
			write(STDERR_FILENO, err, sizeof(err) - 1);
		}
	}
}

static void *allocator_alloc_other(Allocator *const allocator, const size_t size) {
	const char mes[] = "We use mmap\n";
	write(STDERR_FILENO, mes, sizeof(mes) - 1);

	void *new_memory = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
	if (new_memory == MAP_FAILED) {
		const char err[] = "mmap failed 3\n";
		write(STDERR_FILENO, err, sizeof(err) - 1);
		return NULL;
	}

	return new_memory;
}

static void allocator_free_other(Allocator *const allocator, void *const memory) {
	const char mes[] = "We use munmap\n";
	write(STDERR_FILENO, mes, sizeof(mes) - 1);

	if (memory && munmap(memory, sizeof(memory)) == -1) {
		const char err[] = "munmap failed 4\n";
		write(STDERR_FILENO, err, sizeof(err) - 1);
```

```
	}
}

static allocator_create_f *allocator_create;
static allocator_destroy_f *allocator_destroy;
static allocator_alloc_f *allocator_alloc;
static allocator_free_f *allocator_free;

int main(int argc, char **argv) {
	if (argc < 2) {
		const char mes[] = "./a.out(main) <library_path>\n";
		write(STDERR_FILENO, mes, sizeof(mes));
		return 1;
	}

	void *library = dlopen(argv[1], RTLD_LOCAL | RTLD_NOW);

	if (library) {
		allocator_create = dlsym(library, "allocator_create");
		allocator_destroy = dlsym(library, "allocator_destroy");
		allocator_alloc = dlsym(library, "allocator_alloc");
		allocator_free = dlsym(library, "allocator_free");

		if (!allocator_create || !allocator_destroy || !allocator_alloc || !allocator_free) {
			allocator_create = allocator_create_other;
			allocator_destroy = allocator_destroy_other;
			allocator_alloc = allocator_alloc_other;
			allocator_free = allocator_free_other;
```

```
                }
        } else {
                const char mes[] = "failed to open custom library\n";
                write(STDERR_FILENO, mes, sizeof(mes));
                return 1;
        }

        size_t size = MY_MEMORY_SIZE;
    void *new_addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);
    if (new_addr == MAP_FAILED) {
                dlclose(library);
        char mes[] = "mmap failed\n";
        write(STDERR_FILENO, mes, sizeof(mes) - 1);
        return 1;
    }

        Allocator *allocator = allocator_create(new_addr, MY_MEMORY_SIZE);

        if (!allocator) {
                const char mes[] = "Failed to initialize allocator\n";
                write(STDERR_FILENO, mes, sizeof(mes));
                munmap(new_addr, size);
                dlclose(library);

                return 1;
        }

        int *num = (int *)allocator_alloc(allocator, sizeof(int));
```

```
    if (num) {
        *num = 36;
        const char mes[] = "Allocated block with int value 36\n";
        write(STDOUT_FILENO, mes, sizeof(mes));
    } else {
        const char mes[] = "Failed to allocate memory\n";
        write(STDERR_FILENO, mes, sizeof(mes));
    }

    if (num) {
        allocator_free(allocator, num);
        const char mes[] = "Free 36\n";
        write(STDOUT_FILENO, mes, sizeof(mes));
    }

    allocator_destroy(allocator);
    const char mes[] = "Allocator destroyed\n";
    write(STDOUT_FILENO, mes, sizeof(mes));

    if (library) dlclose(library);
    munmap(new_addr, size);

    return EXIT_SUCCESS;
}
```

Сравнение аллокаторов

При тестировании использовался подход с выделением n количества блоков со случайным размером. После случайным образом выбиралось n/2 блока для освобождения и на их место выделялись другие.

Итоги:

1. Фактор использования

- **Метод двойников**:
    - **Оценка**: Средний.
    - **Объяснение**: Этот метод использует фиксированные размеры блоков, что может привести к неэффективному использованию памяти, особенно если запрашиваемые размеры блоков сильно варьируются. Однако, если размеры блоков хорошо подобраны, фактор использования может быть приемлемым.
- **Метод свободных блоков**:
    - **Оценка**: Высокий.
    - **Объяснение**: Этот метод позволяет более эффективно использовать память, поскольку блоки могут быть любого размера. Это снижает фрагментацию и позволяет более гибко управлять памятью.

2. Скорость выделения блоков

- **Метод двойников**:
    - **Оценка**: Высокая.
    - **Объяснение**: Выделение блоков происходит быстро, так как аллокатор просто выбирает из заранее определенного размера блоков. Это минимизирует время поиска и выделения.
- **Метод свободных блоков**:
    - **Оценка**: Средняя.
    - **Объяснение**: Выделение может занять больше времени, так как необходимо искать подходящий свободный блок.

3. Скорость освобождения блоков

- **Метод двойников**:
    - **Оценка**: Средняя.

- **Объяснение**: Освобождение может быть медленнее, так как необходимо объединять соседние свободные блоки, что требует дополнительного времени для проверки и изменения структуры данных.

- **Метод свободных блоков**:
  - **Оценка**: Средняя.
  - **Объяснение**: Освобождение может быть медленнее, так как необходимо объединять соседние свободные блоки, что требует дополнительного времени для проверки и изменения структуры данных.

4. Простота использования аллокатора

- **Метод двойников**:
  - **Оценка**: Высокая.
  - **Объяснение**: Аллокатор с фиксированными размерами блоков проще в использовании, так как пользователю не нужно беспокоиться о фрагментации и размерах блоков.
- **Метод свободных блоков**:
  - **Оценка**: Средняя.
  - **Объяснение**: Хотя этот метод предлагает большую гибкость, он может быть сложнее в использовании из-за необходимости управления фрагментацией и более сложных операций выделения и освобождения.

Протокол работы программы

Тестирование:

Strace:

list.c

```
execve("./main", ["./main", "./liballocator.so"], 0x7ffd371a7a28 /* 70 vars */) = 0
brk(NULL)                               = 0x62eb780dc000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffee5a92600) = -1 EINVAL (Недопустимый аргумент)

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7777900a4000
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
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=66559, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 66559, PROT_READ, MAP_PRIVATE, 3, 0) = 0x777790093000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/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\0P\237\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
pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0I\17\357\204\3$\f\221\2039x\324\224\323\236S"..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2220400, ...}, AT_EMPTY_PATH) = 0
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, 2264656, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x77778fe00000
mprotect(0x77778fe28000, 2023424, PROT_NONE) = 0
mmap(0x77778fe28000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x77778fe28000
mmap(0x77778ffbd000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x77778ffbd000
mmap(0x777790016000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x777790016000
mmap(0x77779001c000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x77779001c000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x777790090000
arch_prctl(ARCH_SET_FS, 0x777790090740) = 0
set_tid_address(0x777790090a10)         = 7414
```

```
set_robust_list(0x777790090a20, 24)     = 0
rseq(0x7777900910e0, 0x20, 0, 0x53053053) = 0
mprotect(0x777790016000, 16384, PROT_READ) = 0
mprotect(0x62eb64aae000, 4096, PROT_READ) = 0
mprotect(0x7777900de000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,
 rlim_max=RLIM64_INFINITY}) = 0
munmap(0x777790093000, 66559)           = 0
getrandom("\x55\xcd\x59\x04\xdd\x8e\x63\x6d", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x62eb780dc000
brk(0x62eb780fd000)                     = 0x62eb780fd000
openat(AT_FDCWD, "./liballocator.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|0775, st_size=15632, ...}, AT_EMPTY_PATH) = 0
getcwd("/home/ann/Desktop/osi/4", 128)  = 24
mmap(NULL, 16432, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
 0x77779009f000
mmap(0x7777900a0000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|
MAP_DENYWRITE, 3, 0x1000) = 0x7777900a0000
mmap(0x7777900a1000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|
MAP_DENYWRITE, 3, 0x2000) = 0x7777900a1000
mmap(0x7777900a2000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x7777900a2000
close(3)                                = 0
mprotect(0x7777900a2000, 4096, PROT_READ) = 0
mmap(NULL, 1024, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
 -1, 0) = 0x7777900dd000
write(1, "Allocated block with int value 3"..., 35Allocated block with int value 36
) = 35
write(1, "Free 36\n\0", 9Free 36
)            = 9
write(1, "Allocator destroyed\n\0", 21Allocator destroyed
) = 21
munmap(0x77779009f000, 16432)           = 0
munmap(0x7777900dd000, 1024)            = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

twins.c

```
execve("./main", ["./main", "./liballocator.so"], 0x7fff7e721938 /* 70 vars */) = 0
brk(NULL)                               = 0x5779ccf8f000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffe34a5b030) = -1 EINVAL (Недопустимый аргумент)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
 0) = 0x76d9a2be6000
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
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=66559, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 66559, PROT_READ, MAP_PRIVATE, 3, 0) = 0x76d9a2bd5000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/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\0P\237\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
pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0I\17\357\204\3$\f\221\2039x\324\224\323\236S"...,
 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2220400, ...}, AT_EMPTY_PATH) = 0
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, 2264656, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
 0x76d9a2800000
mprotect(0x76d9a2828000, 2023424, PROT_NONE) = 0
mmap(0x76d9a2828000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|
MAP_DENYWRITE, 3, 0x28000) = 0x76d9a2828000
mmap(0x76d9a29bd000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|
MAP_DENYWRITE, 3, 0x1bd000) = 0x76d9a29bd000
mmap(0x76d9a2a16000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
MAP_DENYWRITE, 3, 0x215000) = 0x76d9a2a16000
mmap(0x76d9a2a1c000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
MAP_ANONYMOUS, -1, 0) = 0x76d9a2a1c000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
 0) = 0x76d9a2bd2000
arch_prctl(ARCH_SET_FS, 0x76d9a2bd2740) = 0
set_tid_address(0x76d9a2bd2a10)         = 7340
set_robust_list(0x76d9a2bd2a20, 24)     = 0
rseq(0x76d9a2bd30e0, 0x20, 0, 0x53053053) = 0
mprotect(0x76d9a2a16000, 16384, PROT_READ) = 0
mprotect(0x57799f2a6000, 4096, PROT_READ) = 0
mprotect(0x76d9a2c20000, 8192, PROT_READ) = 0
```

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

munmap(0x76d9a2bd5000, 66559) = 0

getrandom("\xe2\x4d\x46\x6f\x65\x6f\x5e\x14", 8, GRND_NONBLOCK) = 8

brk(NULL) = 0x5779ccf8f000

brk(0x5779ccfb0000) = 0x5779ccfb0000

openat(AT_FDCWD, "./liballocator.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|0775, st_size=15632, ...}, AT_EMPTY_PATH) = 0

getcwd("/home/ann/Desktop/osi/4", 128) = 24

mmap(NULL, 16432, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x76d9a2be1000

mmap(0x76d9a2be2000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x76d9a2be2000

mmap(0x76d9a2be3000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x76d9a2be3000

mmap(0x76d9a2be4000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x76d9a2be4000

close(3) = 0

mprotect(0x76d9a2be4000, 4096, PROT_READ) = 0

-1, mmap(NULL, 1024, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, 0) = 0x76d9a2c1f000

write(1, "Allocated block with int value 3"..., 35Allocated block with int value 36

) = 35

write(1, "Free 36\n\0", 9Free 36

) = 9

write(1, "Allocator destroyed\n\0", 21Allocator destroyed

) = 21

munmap(0x76d9a2be1000, 16432) = 0

munmap(0x76d9a2c1f000, 1024) = 0

exit_group(0) = ?

+++ exited with 0 +++

Вывод

В процессе выполнения этой лабораторной работы я освоила работу с динамисескими библиотеками в Си, реализовала два алгоритма аллокации памяти (списки свободных блоков (наиболее подходящее и алгоритм двойников) и сравнила их. Было сложно и интересно разбираться в этой теме.