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-214Б-23

Студент: Кузин А.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 15.02.25

Москва, 2024

Постановка задачи

Вариант 1.

Требуется создать две динамические библиотеки, реализующие два аллокатора: списки свободных блоков (первое подходящее) и блоки по 2^n.

Общий метод и алгоритм решения

Использованные системные вызовы:

1. ***int munmap(void addr, size_t length); -** Удаляет все отображения из заданной области памяти.
2. ***int dlclose(void handle);** - Закрывает динамическую библиотеку, открытую с помощью dlopen, и освобождает ресурсы, связанные с этим дескриптором.
3. ****void dlopen(const char filename, int flag);** - Открывает динамическую библиотеку и возвращает дескриптор для последующего использования.
4. ****void mmap(void addr, size_t length, int prot, int flags, int fd, off_t offset);** – создает новое отображение памяти или изменяет существующее**.**
5. **int write(int _Filehandle, const void *_Buf, unsigned int _MaxCharCount) –** выводит информацию в файл с указанным дескриптором.

Описание программы

1. main.c
   Открывает динамические библиотеки и получает нужные функции. Если в библиотеке не нашлось нужных функций, то вместо них будут использоваться аварийные оберточные функции. Далее как пример функция выделяет и освобождает память массива.
2. 2n_degree_blocks.c
   Файл в котором реализована логика работы аллокатора блоками по 2^n.
   1) Вся память при инициализации разбивается на блоки которые равны степени двойки.
   1) Все блоки хранятся в списке свободных элементов.
   2) Каждый блок хранит указатель на следующий свободный блок.
   3) При освобождении нужно добавить этот блок в список свободных элементов в нужную позицию.
   4) Для выделения памяти выбираем блок N[log2(size)] и возвращаем указатель на первый элемент, помечая блок занятым.
3. free_list_blocks.c
   Файл в котором реализована логика работы аллокатора на спискач свободных блоков.
   1) Все свободные блоки организованы в список
   2) В блоке хранится его размер и указатель на следующий свободный блок
   3) Для выделения памяти проходимся по всему списку свободных блоков и выбираем минимальный блок, который больше или равен по размеру нужного блока. Помечаем блок, как занятый и убираем из списка.
   4) При освобождении памяти возвращаем блок в список свободных элементов. И при возможности сливаем рядом стоящие блоки.

Код программы

main.c

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

typedef struct AllocatorAPI {
    void* (*create)(void* addr, size_t
size);
    void* (*alloc)(void* allocator, size_t
        size);
    void (*free)(void* allocator, void*
        ptr);
void (*destroy)(void* allocator);
} AllocatorAPI;

void* default_create(void* memory,
        size_t size) {
(void)size; return
    memory;
}

void* default_alloc(void* allocator,
        size_t size) {
(void)allocator;
    uint32_t* block = mmap(NULL,
        size + sizeof(uint32_t),
        PROT_READ | PROT_WRITE,
        MAP_SHARED |
        MAP_ANONYMOUS, -1, 0);
    if (block == MAP_FAILED) {
      return NULL;
    }
    *block = (uint32_t)(size +
        sizeof(uint32_t));
return block + 1;
}

void default_free(void* allocator, void*
        memory) {
(void)allocator;
if (!memory) return;
    uint32_t* block =
        (uint32_t*)memory - 1;
   munmap(block, *block);
}
```

```
void default_destroy(void* allocator) {
    (void)allocator;
}

void load_allocator(const char*
        lib_path, AllocatorAPI* api) {
    void* lib_handle = dlopen(lib_path,
        RTLD_LOCAL | RTLD_NOW);
    if (!lib_path || !lib_path[0] ||
        !lib_handle) {
        write(STDERR_FILENO,
        "WARNING: Using default
        allocator\n", 34);
        api->create = default_create;
        api->alloc = default_alloc;
        api->free = default_free;
        api->destroy = default_destroy;
        return;
    }

    api->create = dlsym(lib_handle,
        "create_allocator");
    api->alloc = dlsym(lib_handle,
        "allocate_memory");
    api->free = dlsym(lib_handle,
        "free_memory");
    api->destroy = dlsym(lib_handle,
        "destroy_allocator");

    if (!api->create || !api->alloc || !api-
        >free || !api->destroy) {
        write(STDERR_FILENO,
        "ERROR: Failed loading allocator
        functions\n", 43);
        dlclose(lib_handle);
        api->create = default_create;
        api->alloc = default_alloc;
        api->free = default_free;
        api->destroy = default_destroy;
    }
}

void print_message(const char* msg) {
    write(STDOUT_FILENO, msg,
        strlen(msg));
}

void print_address(const char* label,
        int index, void* address) {
    char buffer[64];
```

```
    int len = 0;

    while (*label) {
        buffer[len++] = *label++;
    }

    if (index < 10) {
        buffer[len++] = '0' + index;
    } else {
        buffer[len++] = '0' + (index / 10);
        buffer[len++] = '0' + (index % 10);
    }

    char* ad = " address: ";
    while (*ad) {
        buffer[len++] = *ad++;
    }

    uintptr_t addr = (uintptr_t)address;
    for (int i = (sizeof(uintptr_t) * 2) - 1; i
        >= 0; --i) {
        int nibble = (addr >> (i * 4)) &
        0xF;
        buffer[len++] = (nibble < 10) ? ('0'
        + nibble) : ('a' + (nibble - 10));
    }

    buffer[len++] = '\n';
    write(STDOUT_FILENO, buffer,
        len);
}

int main(int argc, char** argv) {
    const char* lib_path = (argc > 1) ?
        argv[1] : NULL;
    AllocatorAPI allocator_api;
    load_allocator(lib_path,
        &allocator_api);

    size_t pool_size = 4096;
    void* pool = mmap(NULL,
        pool_size, PROT_READ |
        PROT_WRITE,
        MAP_PRIVATE |
        MAP_ANONYMOUS, -1, 0);
    if (pool == MAP_FAILED) {
        print_message("ERROR: Memory
        pool allocation failed\n");
        return EXIT_FAILURE;
    }
```

```
    void* allocator =
        allocator_api.create(pool,
        pool_size);
    if (!allocator) {
        print_message("ERROR:
        Allocator initialization failed\n");
        munmap(pool, pool_size);
        return EXIT_FAILURE;
    }

    size_t block_sizes[] = {12, 13, 24,
        40, 56, 100, 120, 400};
    void* blocks[8];

    for (int i = 0; i < 8; ++i) {
        blocks[i] =
        allocator_api.alloc(allocator,
        block_sizes[i]);
        if (!blocks[i]) {
            print_message("ERROR:
        Memory allocation failed\n");
            break;
        }
        print_address("block №", i + 1,
        blocks[i]);
    }
    print_message("INFO: Memory
        allocation - SUCCESS\n");

    for (int i = 0; i < 8; ++i) {
        if (blocks[i]) {
            allocator_api.free(allocator,
        blocks[i]);
        }
    }
    print_message("INFO: Memory
        freed\n");

    allocator_api.destroy(allocator);
    print_message("INFO: Allocator
        destroyed\n");

    return EXIT_SUCCESS;
}
```

free_list_blocks.c

```
#include <stddef.h>

typedef struct Allocator {
        void* start;
        size_t total_size;
```

```
	void* free_blocks;
} Allocator;

typedef struct FreeBlock {
	size_t block_size;
	struct FreeBlock* next_block;
} FreeBlock;

Allocator* create_allocator(void* memory_pool, size_t pool_size) {
	if (memory_pool == NULL || pool_size < sizeof(FreeBlock)) {
		return NULL;
	}

	Allocator* allocator = (Allocator*)memory_pool;
	allocator->start = (char*)memory_pool + sizeof(Allocator);
	allocator->total_size = pool_size - sizeof(Allocator);
	allocator->free_blocks = allocator->start;

	FreeBlock* first_block = (FreeBlock*)allocator->start;
	first_block->block_size = allocator->total_size;
	first_block->next_block = NULL;

	return allocator;
}

void destroy_allocator(Allocator* allocator) {
	if (allocator == NULL) {
	return;
	}

	allocator->start = NULL;
	allocator->total_size = 0;
	allocator->free_blocks = NULL;
}

void* allocate_memory(Allocator* allocator, size_t request_size) {
	if (allocator == NULL || request_size == 0) {
		return NULL;
	}

	FreeBlock* previous = NULL;
	FreeBlock* current = (FreeBlock*)allocator->free_blocks;

	while (current != NULL) {
		if (current->block_size >= request_size + sizeof(FreeBlock)) {
			if (current->block_size > request_size + sizeof(FreeBlock)) {
				FreeBlock* remaining_block = (FreeBlock*)((char*)current + sizeof(FreeBlock) +
request_size);
				remaining_block->block_size = current->block_size - request_size - sizeof(FreeBlock);
				remaining_block->next_block = current->next_block;

				current->block_size = request_size;
				current->next_block = remaining_block;
			}
```

```
                        if (previous != NULL) {
                                previous->next_block = current->next_block;
                        } else {
                                allocator->free_blocks = current->next_block;
                        }

                        return (char*)current + sizeof(FreeBlock);
                }

                previous = current;
                current = current->next_block;
        }

        return NULL;
}

void free_memory(Allocator* allocator, void* memory) {
        if (allocator == NULL || memory == NULL) {
                return;
        }

        FreeBlock* block_to_free = (FreeBlock*)((char*)memory - sizeof(FreeBlock));
        block_to_free->next_block = (FreeBlock*)allocator->free_blocks;
        allocator->free_blocks = block_to_free;
}
```

2n_degree_blocks.c

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

#define MIN_BLOCK_SIZE 16

typedef struct BlockHeader {
        struct BlockHeader *next;
} BlockHeader;

typedef struct Allocator {
        BlockHeader **free_lists;
        size_t num_lists;
        void *base_address;
        size_t total_size;
} Allocator;

static int calculate_log2(int value) {
        int result = -1;
        while (value > 0) {
                value >>= 1;
                result++;
        }
```

```
    return result;
}

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

    Allocator *allocator = (Allocator *)memory;
    allocator->base_address = 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)(calculate_log2(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;
        }

        size_t num_blocks = (remaining_size >= (block_size + sizeof(BlockHeader)) * 2) ? 2 : 1;

        for (size_t i = 0; i < num_blocks; i++) {
            BlockHeader *header = (BlockHeader *)current_block;
            size_t index = (block_size == 0) ? 0 : calculate_log2(block_size);
            header->next = allocator->free_lists[index];
            allocator->free_lists[index] = header;

            current_block = (char *)current_block + block_size;
            remaining_size -= block_size;
        }

        block_size <<= 1;
    }

    return allocator;
}

void destroy_allocator(Allocator *allocator) {
    if (allocator != NULL) {
        munmap(allocator->base_address,  allocator->total_size);
    }
}
```

```
void* allocate_memory(Allocator *allocator, size_t size) {
	if (allocator == NULL || size == 0) {
		return NULL;
	}

	size_t index = (size == 0) ? 0 : calculate_log2(size) + 1;
	if (index >= allocator->num_lists) {
		index = allocator->num_lists - 1;
	}

	while (index < allocator->num_lists && allocator->free_lists[index] == NULL) {
		index++;
	}

	if (index >= allocator->num_lists) {
	return NULL;
	}

	BlockHeader *block = allocator->free_lists[index];
	allocator->free_lists[index] = block->next;

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

void free_memory(Allocator *allocator, void *ptr) {
	if (allocator == NULL || ptr == NULL) {
		return;
	}

	BlockHeader *block = (BlockHeader *)((char *)ptr - sizeof(BlockHeader));
	size_t block_offset = (char *)block - (char *)allocator->base_address;

	size_t block_size = MIN_BLOCK_SIZE;
	while ((block_size << 1) <= block_offset) {
		block_size <<= 1;
	}

	size_t index = calculate_log2(block_size);
	if (index >= allocator->num_lists) {
		index = allocator->num_lists - 1;
	}

	block->next = allocator->free_lists[index];
	allocator->free_lists[index] = block;
}
```

Протокол работы программы

```
vboxuser@Linux:~/lab04/src$ ./main
WARNING: Using default allocator
block №1 address: 0000741b45bd2004
block №2 address: 0000741b45bd1004
block №3 address: 0000741b45bd0004
block №4 address: 0000741b45bcf004
block №5 address: 0000741b45bce004
block №6 address: 0000741b45bcd004
block №7 address: 0000741b45bcc004
block №8 address: 0000741b45bcb004
INFO: Memory allocation - SUCCESS
INFO: Memory freed
INFO: Allocator destroyed
vboxuser@Linux:~/lab04/src$ ./main ./2ndegree.so
block №1 address: 000079508bfa0080
block №2 address: 000079508bfa0070
block №3 address: 000079508bfa00b0
block №4 address: 000079508bfa0110
block №5 address: 000079508bfa00d0
block №6 address: 000079508bfa01d0
block №7 address: 000079508bfa0150
block №8 address: 000079508bfa0350
INFO: Memory allocation - SUCCESS
INFO: Memory freed
INFO: Allocator destroyed
vboxuser@Linux:~/lab04/src$ ./main ./flist.so
block №1 address: 00007157dc6c1028
block №2 address: 00007157dc6c1044
block №3 address: 00007157dc6c1061
block №4 address: 00007157dc6c1089
block №5 address: 00007157dc6c10c1
block №6 address: 00007157dc6c1109
block №7 address: 00007157dc6c117d
block №8 address: 00007157dc6c1205
INFO: Memory allocation - SUCCESS
INFO: Memory freed
INFO: Allocator destroyed
```

Strace:

strace -f ./main ./flist.so

execve("./main", ["./main", "./flist.so"], 0x7ffcf157ed60 /* 71 vars */) = 0

brk(NULL) = 0x5a5dd3392000

arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd20d6c8e0) = -1 EINVAL

(Invalid argument)

mmap(NULL, 8192, PROT_READ|PROT_WRITE,

MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7a1b60d6a000

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

directory)

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

newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=59559, ...},
AT_EMPTY_PATH) = 0

mmap(NULL, 59559, PROT_READ, MAP_PRIVATE, 3, 0) =
0x7a1b60d5b000

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"...,
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) = 0x7a1b60a00000

mprotect(0x7a1b60a28000, 2023424, PROT_NONE) = 0

mmap(0x7a1b60a28000, 1658880, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) =
0x7a1b60a28000

mmap(0x7a1b60bbd000, 360448, PROT_READ,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) =
0x7a1b60bbd000

mmap(0x7a1b60c16000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) =
0x7a1b60c16000

mmap(0x7a1b60c1c000, 52816, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) =
0x7a1b60c1c000

close(3) = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7a1b60d58000

arch_prctl(ARCH_SET_FS, 0x7a1b60d58740) = 0

set_tid_address(0x7a1b60d58a10) = 18649

set_robust_list(0x7a1b60d58a20, 24) = 0

rseq(0x7a1b60d590e0, 0x20, 0, 0x53053053) = 0

mprotect(0x7a1b60c16000, 16384, PROT_READ) = 0

mprotect(0x5a5dd2dca000, 4096, PROT_READ) = 0

mprotect(0x7a1b60da4000, 8192, PROT_READ) = 0

prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024,

rlim_max=RLIM64_INFINITY}) = 0

munmap(0x7a1b60d5b000, 59559)   = 0

getrandom("\x36\x09\xa6\x61\x2d\x50\x4b\x86", 8, GRND_NONBLOCK)

= 8

brk(NULL)   = 0x5a5dd3392000

brk(0x5a5dd33b3000)   = 0x5a5dd33b3000

openat(AT_FDCWD, "./flist.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"...,

832) = 832

newfstatat(3, "", {st_mode=S_IFREG|0775, st_size=15272, ...},

AT_EMPTY_PATH) = 0

getcwd("/home/vboxuser/lab04/src", 128) = 25

mmap(NULL, 16424, PROT_READ,

MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7a1b60d65000

mmap(0x7a1b60d66000, 4096, PROT_READ|PROT_EXEC,

MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) =

0x7a1b60d66000

mmap(0x7a1b60d67000, 4096, PROT_READ,

MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) =

0x7a1b60d67000

mmap(0x7a1b60d68000, 8192, PROT_READ|PROT_WRITE,

MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) =

0x7a1b60d68000

close(3)   = 0

mprotect(0x7a1b60d68000, 4096, PROT_READ) = 0

mmap(NULL, 4096, PROT_READ|PROT_WRITE,

MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7a1b60da3000

write(1, "block \342\204\2261 address: 00007a1b60da"..., 37block №1

address: 00007a1b60da3028

) = 37

write(1, "block \342\204\2262 address: 00007a1b60da"..., 37block №2

address: 00007a1b60da3044

) = 37

write(1, "block \342\204\2263 address: 00007a1b60da"..., 37block №3

address: 00007a1b60da3061

) = 37

write(1, "block \342\204\2264 address: 00007a1b60da"..., 37block №4

address: 00007a1b60da3089

) = 37

write(1, "block \342\204\2265 address: 00007a1b60da"..., 37block №5

address: 00007a1b60da30c1

) = 37

write(1, "block \342\204\2266 address: 00007a1b60da"..., 37block №6

address: 00007a1b60da3109

) = 37

write(1, "block \342\204\2267 address: 00007a1b60da"..., 37block №7

address: 00007a1b60da317d

) = 37

write(1, "block \342\204\2268 address: 00007a1b60da"..., 37block №8

address: 00007a1b60da3205

) = 37

write(1, "INFO: Memory allocation - SUCCES"..., 34INFO: Memory

allocation - SUCCESS

) = 34

write(1, "INFO: Memory freed\n", 19INFO: Memory freed

) = 19

write(1, "INFO: Allocator destroyed\n", 26INFO: Allocator destroyed

) = 26

exit_group(0) = ?

+++ exited with 0 +++

Вывод

В рамках лабораторной работы была разработана программа, демонстрирующая работу аллокатора передаваемого в качестве аргумента при вызове программы. Было реализовано 2 аллокатора и проведена работа по сравнению их работоспособности.