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-213Б-23

Студент: Иванов В. М.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 18.10.24

Москва, 2024

Постановка задачи

Вариант 9.

Требуется создать две динамические библиотеки, реализующие два аллокатора, соответственно. Библиотеки загружаются в память с помощью интерфейса ОС (dlopen / LoadLibrary) для работы с динамическими библиотеками. Выбор библиотеки, реализующей аллокатор, осуществляется чтением первого аргумента при запуске программы (argv[1]). Этот аргумент должен содержать путь до динамической библиотеки (относительный или абсолютный). Если аргумент не передан или по переданному пути библиотеки не оказалось, то указатели на функции, реализующие API аллокатора ниже, должны быть присвоены функциям, которые оборачивают системный аллокатор ОС (mmap / VirtualAlloc) в этот API. Эти аварийные оберточные функции должны быть реализованы внутри программы, которая загружает динамические библиотеки. Каждый аллокатор памяти должен иметь функции аналогичные стандартным функциям malloc и free (realloc, опционально). Перед работой каждый аллокатор инициализируется свободными страницами памяти, выделенными стандартными средствами ядра (mmap / VirtualAlloc). Необходимо самостоятельно разработать стратегию тестирования для определения ключевых характеристик аллокаторов памяти. При тестировании нужно свести к минимуму потери точности из-за накладных расходов при измерении ключевых характеристик, описанных выше.

Алгоритмы аллокаторов: списки свободных блоков (наиболее подходящее) и алгоритм двойников.

Общий метод и алгоритм решения

Кратко опишите системные вызовы, которые вы использовали в лабораторной работе.

Использованные системные вызовы:

- void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t offset) – выделение памяти
- int munmap(void *addr, size_t len) – освобождение памяти

Аллоктор на списке свободных блоков:

В этом аллокаторе память выделяется блоками, в начале и конце каждого из которых содержится HEADER из 4 байт. В HEADER записаны размер блока в байтах – четное число, и является ли блок свободным – последний бит. Функцию allocator_create выделяет память с помощью mmap и записывает ее размер и указатель на начало в структуру Allocator. Функция allocator_destroy освобождает память через munmap. В функции allocator_alloc создается переменная ptr = NULL, p = allocator.memory. Программа в цикле проходит по всем блокам, увеличивая p на количество байт в HEADER и присваивая ptr значение p, если блок свободен, размер блока больше или равен необходимому и размер блока минимален. Если по завершении цикла ptr == NULL, то возвращается NULL: аллокация не удалась. В противном случае размер в HEADER меняется. Если новый размер меньше размера блока, то создается новый блок, в его HEADER записываются его размеры. Функция возвращает указатель на ptr со сдвигом в 4 байта. В функции allocator_free в HEADER последний бит меняется на 0 – блок свободен. Если соседние блоки свободны, происходит объединение с ними – в их HEADER меняется размер блока.

Аллокатор на алгоритме двойников:

В этом аллокаторе память выделяется блоками, размером 2^n. Каждый блок содержит HEADER, в котором записана его степень n, его путь от корневого: является ли он левым или правым блоком, указатель на то, является ли он свободным и два указателя на следующий и предыдущий блок. Функция allocator_create выделяет память с помощью mmap и записывает ее размер и указатель на начало в структуру Allocator. Также в список free аллокатора записывается указатель на первый блок. Функция allocator_destroy освобождает память через munmap. В функции allocator_alloc находится степень n необходимая для хранения информации. Далее программа проходит по связному списку свободных блоков и выбирает из них блок с минимальной степенью большей чем необходимая. Если такого блока нет, то возвращается NULL. Найденный

блок удаляется из списка free. Если степень найденного блока больше необходимой, то он в цикле делится надвое пока его степень не станет равной необходимой. Все созданные правые блоки добавляются в список free. Функция возвращает указатель на выделенный блок со сдвигом в sizeof(Block) байт. Функция allocator_free пытается в цикле объединить освобождаемый блок с двойниками. Если двойник существует (не верхний блок) и он свободен, то двойник удаляется из списка free. Полученный после цикла объединенный блок помечается как свободный и добавляется в список free.

Код программы

main.c

```
#include <stdint.h>
#include <dlfcn.h>
#include <unistd.h>

#include <string.h>

#include <sys/mman.h>

#include "allocator.h"



static allocator_create_func *allocator_create;
static allocator_destroy_func *allocator_destroy;
static allocator_alloc_func *allocator_alloc;
static allocator_free_func *allocator_free;

struct Allocator
{
    size_t size;
    uint32_t *memory;
};

Allocator *allocator_create_def(void *const memory, const size_t size)
{
    (void)size;
    (void)memory;
    return NULL;
```

```
}

void allocator_destroy_def(Allocator *const allocator)
{
    (void)allocator;
}

void *allocator_alloc_def(Allocator *const allocator, const size_t size)
{
    (void)allocator;
    uint32_t *memory = mmap(NULL, size + 1, PROT_READ | PROT_WRITE,
                            MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (memory == MAP_FAILED)
    {
        return NULL;
    }
    *memory = size + 1;
    return memory + 1;
}

void allocator_free_def(Allocator *const allocator, void *const memory)
{
    (void)allocator;
    if (memory == NULL)
        return;
    uint32_t *mem = memory;
    mem--;
    munmap(mem, *mem);
}

#define LOAD_FUNCTION(name)                                          \
    {                                                       \
        name = dlsym(library, #name);                               \
        if (allocator_create == NULL)                               \
        {                                                   \
            char message[] =                                        \
```

```
            "WARNING: failed to find " #name " function implementation\n"; \
        write(STDERR_FILENO, message, strlen(message));                    \
        name = name##_def;                                                \
    }                                                                     \
}

void load_dynamic(const char *path)
{
    void *library = dlopen(path, RTLD_LOCAL | RTLD_NOW);
    if (path && library)
    {
        LOAD_FUNCTION(allocator_create);
        LOAD_FUNCTION(allocator_destroy);
        LOAD_FUNCTION(allocator_alloc);
        LOAD_FUNCTION(allocator_free);
    }
    else
    {
        char *message = "failed to load shared library. Using mmap implementation\n";
        write(STDERR_FILENO, message, strlen(message));
        allocator_create = allocator_create_def;
        allocator_destroy = allocator_destroy_def;
        allocator_alloc = allocator_alloc_def;
        allocator_free = allocator_free_def;
    }
}

int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        write(STDERR_FILENO, "No path is Given!: ", 19);
        load_dynamic(NULL); // load default implementation
    }
    else {
        load_dynamic(argv[1]);
```

```
    }
    char mem[1024];
    Allocator *memAllocator = allocator_create(mem, sizeof(mem));
    char *a = allocator_alloc(memAllocator, 12 * sizeof(char));
    char *b = allocator_alloc(memAllocator, 12 * sizeof(char));
    for (int i = 0; i < 10; i++)
    {
        a[i] = '0' + i;
        b[10 - i - 1] = '0' + i;
    }
    b[10] = 0;
    a[10] = 0;

    write(STDOUT_FILENO, "contents of a: ", 15);
    write(STDOUT_FILENO, a, strlen(a));
    write(STDOUT_FILENO, "\n", 1);
    write(STDOUT_FILENO, "contents of b: ", 15);
    write(STDOUT_FILENO, b, strlen(b));
    write(STDOUT_FILENO, "\n", 1);
    strcpy(a, b);
    write(STDOUT_FILENO, "contents of a after copy from b: ", 33);
    write(STDOUT_FILENO, b, strlen(b));
    write(STDOUT_FILENO, "\n", 1);
    allocator_free(memAllocator, a);
    allocator_free(memAllocator, b);
    allocator_destroy(memAllocator);
}
```

buddy.c

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

#include "allocator.h"

#ifndef BLOCK_COUNT
```

```
#define BLOCK_COUNT 128
#endif

typedef struct Block Block;
struct Block
{
    bool used;
    unsigned int path;
    unsigned char size_class;
    Block *next;
    Block *prev;
};

#define MIN_SIZE_CLASS ((size_t)ceil(log2(sizeof(Block) / sizeof(unsigned int) + 1)))
#define MAX_SIZE_CLASS ((size_t)log2(BLOCK_COUNT))

struct Allocator
{
    size_t size;
    char *mem;
    Block *free;
};

Block *buddy(Block *p)
{
    unsigned int *ptr = (unsigned int *)p;
    if (p->path & (1 << p->size_class))
        ptr -= (1 << p->size_class);
    else
        ptr += (1 << p->size_class);
    return (Block *)ptr;
}

Allocator *allocator_create(void *const memory, const size_t size)
{
    if (memory == NULL || size < sizeof(Allocator))
```

```
        return NULL;
    Allocator *buddy_alloc = (Allocator *)memory;
    buddy_alloc->size = 1 << MAX_SIZE_CLASS; // 128 4 byte blocks
    buddy_alloc->mem = mmap(NULL, buddy_alloc->size * sizeof(unsigned int), PROT_READ |
PROT_WRITE,
                    MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (buddy_alloc->mem == MAP_FAILED)
        return NULL;


    buddy_alloc->free = (Block *)buddy_alloc->mem;
    buddy_alloc->free->used = false;
    buddy_alloc->free->size_class = MAX_SIZE_CLASS;
    buddy_alloc->free->next = NULL;
    buddy_alloc->free->prev = NULL;
    buddy_alloc->free->path = 0;
    return buddy_alloc;
}


void allocator_destroy(Allocator *const allocator)
{
    munmap(allocator->mem, allocator->size * sizeof(unsigned int));
}


void *allocator_alloc(Allocator *const buddy_alloc, const size_t size)
{
    if (buddy_alloc == 0 || size == 0)
        return NULL;
    size_t size_in_u32 = (size + sizeof(Block) + 3) / 4;
    size_t size_class = ceil(log2(size_in_u32));
    if (size_class > MAX_SIZE_CLASS)
        return NULL;
    Block *p = buddy_alloc->free;
    for (Block *c = buddy_alloc->free; c; c = c->next)
    {


        if (c->used || (unsigned char)c->size_class < size_class)
```

```
            continue;
        if (p->used)
        {
            p = c;
            continue;
        }
        if (c->size_class < p->size_class)
            p = c;
    }
    if (p == NULL || p->used)
        return NULL;

    if (buddy_alloc->free == p)
        buddy_alloc->free = p->next;
    if (p->next)
        p->next->prev = p->prev;
    if (p->prev)
        p->prev->next = p->next;

    while (p->size_class > size_class)
    {
        p->size_class = p->size_class - 1;
        Block *other = buddy(p);
        other->size_class = p->size_class;
        other->next = buddy_alloc->free;
        other->prev = NULL;
        other->used = false;
        other->path = p->path | 1 << p->size_class;
        if (buddy_alloc->free)
            buddy_alloc->free->prev = other;
        buddy_alloc->free = other;
    }
    p->used = true;
    return (char *)p + sizeof(Block);
}
```

```
void allocator_free(Allocator *const allocator, void *const memory)
{
    if (!allocator || !memory)
        return;
    Block *p = (Block *)((char *)memory - sizeof(Block));
    while (p->size_class < MAX_SIZE_CLASS)
    {
        Block *bud = buddy(p);
        if (bud->used)
            break;
        if (bud->prev)
            bud->prev->next = bud->next;
        if (bud->next)
            bud->next->prev = bud->prev;
        if (allocator->free == bud)
            allocator->free = bud->next;
        p = p->path & (1 << p->size_class) ? bud : p;
        p->size_class++;
    }
    p->used = false;
    p->next = allocator->free;
    p->prev = NULL;
    allocator->free = p;
}
```

freelist.c

```
#include <string.h>
#include <sys/mman.h>
#include <stdlib.h>

#include "allocator.h"

#ifndef BLOCK_COUNT
#define BLOCK_COUNT 128
#endif

struct Allocator
```

```
{
    size_t size;
    unsigned int *memory;
};

Allocator *allocator_create(void *const memory, const size_t size)
{
    if (memory == NULL || size < sizeof(Allocator))
        return NULL;
    Allocator *list_alloc = (Allocator *)memory;
    list_alloc->size = BLOCK_COUNT; // 128 4 byte blocks
    if (list_alloc->size % 2 == 1)
        list_alloc->size++;

    list_alloc->memory =
        mmap(NULL, list_alloc->size * sizeof(unsigned int), PROT_READ | PROT_WRITE,
            MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (list_alloc->memory == MAP_FAILED)
        return NULL;
    *list_alloc->memory = list_alloc->size;
    *(list_alloc->memory + list_alloc->size - 1) = list_alloc->size;
    return list_alloc;
}

void allocator_destroy(Allocator *const allocator)
{
    munmap(allocator->memory, allocator->size * sizeof(unsigned int));
}

void *allocator_alloc(Allocator *const allocator, const size_t size)
{
    size_t size_in_blocks = (size + 3) / 4;
    size_t full_size = size_in_blocks + 2;
    unsigned int *end = allocator->memory + allocator->size;

    size_t diff = -1;
```

```
    unsigned int *ptr = NULL;

    for (unsigned int *p = allocator->memory; p < end; p = p + (*p & ~1))
    {
        if (*p & 1)
            continue;
        if (*p < full_size) // we need memory for headers
            continue;

        if (size - *p < diff)
        {
            diff = size - *p;
            ptr = p;
        }
    }
    if (!ptr)
        return NULL;

    size_t newsize =
        full_size % 2 == 0 ? full_size : full_size + 1; // round to even bytes

    size_t oldsize = *ptr & ~1;
    *ptr = newsize | 1;          // front header
    *(ptr + newsize - 1) = *ptr; // back header

    if (newsize < oldsize)
    {
        *(ptr + newsize) = oldsize - newsize;
        *(ptr + oldsize - 1) = oldsize - newsize;
    }
    return ptr + 1;
}

void allocator_free(Allocator *const allocator, void *const memory)
{
    (void)allocator;
```

```
    unsigned int *ptr = (unsigned int *)memory - 1;
    *ptr = *ptr & ~1;

    unsigned int *front = ptr;
    unsigned int *back = ptr + *ptr - 1;
    size_t size = *ptr;

    unsigned int *next = ptr + *ptr;
    if ((*next & 1) == 0)
    {
        back += *next;
        size += *next;
    }

    if (ptr > allocator->memory)
    {
        unsigned int *prev_back = ptr - 1;
        if (((*prev_back) & 1) == 0)
        {
            front -= *prev_back;
            size += *prev_back;
        }
    }
    *front = size;
    *back = size;
}
```

Протокол работы программы

Тестирование:

```
$./main
No path is Given!: failed to load shared library. Using mmap implementation
contents of a: 0123456789
contents of b: 9876543210
contents of a after copy from b: 9876543210
```

$./main buddy.so

contents of a: 0123456789

contents of b: 9876543210

contents of a after copy from b: 9876543210

$./main freelist.so

contents of a: 0123456789

contents of b: 9876543210

contents of a after copy from b: 9876543210

Strace:

execve("./main", ["./main", "freelist.so"], 0x7ffce4263198 /* 82 vars */) = 0

brk(NULL) = 0x6044bef17000

access("/etc/ld.so.preload", R_OK) = -1 ENOENT (Нет такого файла или каталога)

openat(AT_FDCWD, "./glibc-hwcaps/x86-64-v3/libm.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (Нет такого файла или каталога)

openat(AT_FDCWD, "./glibc-hwcaps/x86-64-v2/libm.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (Нет такого файла или каталога)

openat(AT_FDCWD, "./libm.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (Нет такого файла или каталога)

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

fstat(3, {st_mode=S_IFREG|0644, st_size=248131, ...}) = 0

mmap(NULL, 248131, PROT_READ, MAP_PRIVATE, 3, 0) = 0x751937c42000

close(3) = 0

mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x751937c40000

openat(AT_FDCWD, "/usr/lib/libm.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\0\0\0\0\0\0\0\0\0\0"..., 832) = 832

fstat(3, {st_mode=S_IFREG|0755, st_size=973144, ...}) = 0

mmap(NULL, 975176, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x751937b51000

mmap(0x751937b5f000, 536576, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xe000) = 0x751937b5f000

mmap(0x751937be2000, 376832, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x91000) = 0x751937be2000

mmap(0x751937c3e000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0xec000) = 0x751937c3e000

close(3) = 0

openat(AT_FDCWD, "./glibc-hwcaps/x86-64-v3/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (Нет такого файла или каталога)

openat(AT_FDCWD, "./glibc-hwcaps/x86-64-v2/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (Нет такого файла или каталога)

openat(AT_FDCWD, "./libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (Нет такого файла или каталога)

openat(AT_FDCWD, "/usr/lib/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\0\340_\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

fstat(3, {st_mode=S_IFREG|0755, st_size=2014520, ...}) = 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, 2034616, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x751937960000

mmap(0x751937984000, 1511424, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x24000) = 0x751937984000

mmap(0x751937af5000, 319488, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x195000) = 0x751937af5000

mmap(0x751937b43000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e3000) = 0x751937b43000

mmap(0x751937b49000, 31672, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x751937b49000

close(3) = 0

mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x75193795d000

arch_prctl(ARCH_SET_FS, 0x75193795d740) = 0

set_tid_address(0x75193795da10) = 333756

set_robust_list(0x75193795da20, 24) = 0

rseq(0x75193795e060, 0x20, 0, 0x53053053) = 0

mprotect(0x751937b43000, 16384, PROT_READ) = 0

mprotect(0x751937c3e000, 4096, PROT_READ) = 0

mprotect(0x6044ab512000, 4096, PROT_READ) = 0

mprotect(0x751937cb9000, 8192, PROT_READ) = 0

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

munmap(0x751937c42000, 248131) = 0

openat(AT_FDCWD, "./glibc-hwcaps/x86-64-v3/freelist.so", O_RDONLY|O_CLOEXEC) = -1 ENOENT (Нет такого файла или каталога)

openat(AT_FDCWD, "./glibc-hwcaps/x86-64-v2/freelist.so", O_RDONLY|O_CLOEXEC) = -1 ENOENT (Нет такого файла или каталога)

openat(AT_FDCWD, "./freelist.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

getrandom("\x4a\xb8\x0a\x75\x1d\xba\x0a\xdb", 8, GRND_NONBLOCK) = 8

brk(NULL) = 0x6044bef17000

brk(0x6044bef38000) = 0x6044bef38000

fstat(3, {st_mode=S_IFREG|0755, st_size=17736, ...}) = 0

getcwd("/home/vovan/Documents/MAI_OS/lab04/src", 128) = 39

mmap(NULL, 16416, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x751937c7a000

mmap(0x751937c7b000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1000) = 0x751937c7b000

mmap(0x751937c7c000, 4096, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x751937c7c000

mmap(0x751937c7d000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x2000) = 0x751937c7d000

close(3) = 0

mprotect(0x751937c7d000, 4096, PROT_READ) = 0

mmap(NULL, 512, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0) = 0x751937c79000

write(1, "contents of a: ", 15) = 15

write(1, "0123456789", 10) = 10

write(1, "\n", 1) = 1

write(1, "contents of b: ", 15) = 15

write(1, "9876543210", 10) = 10

write(1, "\n", 1) = 1

write(1, "contents of a after copy from b:"..., 33) = 33

write(1, "9876543210", 10) = 10

write(1, "\n", 1) = 1

munmap(0x751937c79000, 512) = 0

exit_group(0) = ?

+++ exited with 0 +++

Вывод

Я изучил принципы работы аллокаторов памяти на списке свободных блоуов и на алгоритме двойников, реализовал их и протестировал. Так же я изучил принцип работы с динамическими бибилиотеками и применил их загрузку из аргументов командной строки