Лабораторная работа № 7
«Аллокатор памяти»

Текст исходной программы:

```
#include "mem.h"
#include <stdio.h>
#include <errno.h>
#include <string.h>

static uint8_t* HEAP_START = NULL;
static const size_t MMAP_PREF_SIZE = 0x8000; /*With great power comes great responsobility*/
static const uint8_t FREE_FLAG = 0x1;
static const size_t TINY_MEMBLOCK_SIZE = sizeof(memblock_t);

static int is_last(const memblock_t* const ptr) { return ptr->prev_size == 0; }

static void set_in_use(memblock_t* const ptr)
{
        ptr->size &= ~FREE_FLAG;
}

static size_t align_data_size(const size_t query)
{
        return query + ((query % TINY_MEMBLOCK_SIZE) ? TINY_MEMBLOCK_SIZE - (query % TINY_MEMBLOCK_SIZE) :
0);
}
/* We take memory for header, user data and size_t for prev_size*/
static void init_memblock(memblock_t *ptr, size_t size, size_t prev_size)
{
        ptr->size = align_data_size(size);
        set_in_use(ptr);
        ptr->prev_size = prev_size;
}
/* It exists to get right size, because 0 bit of size is FLAG which shows if block is free/in-use*/
static size_t get_memblock_size(const memblock_t * const ptr)
{
        return ptr->size & ~FREE_FLAG;
}

static memblock_t* get_next_memblock(const memblock_t * const ptr)
{
        return (memblock_t*)((uint8_t*)ptr + sizeof(memblock_t) + get_memblock_size(ptr));
}

static memblock_t* get_prev_memblock(const memblock_t * const ptr)
{
        if(ptr == (memblock_t*)HEAP_START) return NULL;
        return (memblock_t*)((uint8_t*)ptr - ptr->prev_size - sizeof(memblock_t));
}

static int is_free(const memblock_t* const ptr)
{
        return ptr->size & FREE_FLAG;
}

static memblock_t* calculate_next_memblock(const memblock_t* const ptr, const size_t query)
{
        uint8_t* result = (uint8_t*)ptr + sizeof(memblock_t) + query + sizeof(size_t);
        return (memblock_t*)result;
}
/*
           memblock -> +--------------+
                       | prev_size    |
                       +--------------+
                       | size     | f |
               mem  -> +--------------+
                       | user data ...|
           memblock -> +--------------+
                       | size         |
                       +--------------+
                       ~ nextsize | f ~
                       *~~~~~~~~~~~~~~*
                       ~ next data    ~
                       *~~~~~~~~~~~~~~*
                       ~
                       *
```

```
*/
void print_memory()
{
        memblock_t *i = (memblock_t*)(HEAP_START);

        while(i->prev_size != 0)
        {
                printf("+- PTR %p ----------------+\n", (void*)i);
                printf("| prev size %lu\n", i->prev_size);
                printf("| size %lu\n", get_memblock_size(i));
                if(is_free(i)) puts("| free"); else puts("| in use ");
                i = get_next_memblock(i);
        }
}

static uint8_t* get_memptr(memblock_t* ptr)
{
        return (uint8_t*)ptr + sizeof(memblock_t);
}

static void set_free(memblock_t* const ptr);

static void join_memblocks(memblock_t* const dist, memblock_t* const src)
{
        int flag = is_free(dist);
        memblock_t* next;
        size_t newsize = get_memblock_size(dist) + get_memblock_size(src) + sizeof(memblock_t);

        dist->size = align_data_size(newsize); /* just in case */
        next = get_next_memblock(src);

        if(next->prev_size != 0) next->prev_size = newsize;
        if(flag) set_free(dist);
}

static void set_free(memblock_t* const ptr)
{
        memblock_t* i = get_next_memblock(ptr);
        memblock_t* p = get_prev_memblock(ptr);
        memblock_t *end;
        uint8_t* ptr_to_unmap, *next_reg;
        size_t mmap_region_start, mmap_region_prev, mmap_region_end, mmap_region_begin, next_size,
newsize;

        ptr->size |= FREE_FLAG;
        /* Check if we're freeing the last block which spans over mmaped regions */
        if(is_last(i))
        {
                /* we have to keep the prev_size header */
                ptr_to_unmap = (uint8_t*)ptr + sizeof(size_t);
                mmap_region_prev = (size_t)(ptr_to_unmap - 1) / MMAP_PREF_SIZE;
                mmap_region_start = (size_t)(ptr_to_unmap) / MMAP_PREF_SIZE;
                mmap_region_end = (size_t)(i+sizeof(size_t)) / MMAP_PREF_SIZE;

        printf("2");
                if(mmap_region_start < mmap_region_end || mmap_region_prev < mmap_region_start)
                {
                        mmap_region_begin = mmap_region_start + (mmap_region_start == mmap_region_prev ?
1 : 0);
                        next_reg = (uint8_t*)(mmap_region_begin * MMAP_PREF_SIZE);
                        next_size = (mmap_region_end - mmap_region_begin + 1) * MMAP_PREF_SIZE;

        printf("3");
                        munmap(next_reg, next_size);/*unmapping region of size*/

        printf("4");
                        if(mmap_region_start == mmap_region_prev)
                        {
                                newsize = (next_reg - ((uint8_t*)ptr + sizeof(memblock_t) +
sizeof(size_t)));
                                newsize -= newsize % sizeof(memblock_t);
                                ptr->size = newsize;

                                if((ptr->size < TINY_MEMBLOCK_SIZE) && p)
                                {
                                        join_memblocks(p, ptr);
                                        return;
                                } else
                                {
                                        /* if we're first and only block (none previous), that means we can
get some free size, profits! */
                                        if(p == NULL && ptr->size < TINY_MEMBLOCK_SIZE) ptr->size =
TINY_MEMBLOCK_SIZE;
                                        ptr->size |= FREE_FLAG;
                                        end = get_next_memblock((memblock_t*)ptr);
                                        end->prev_size = 0;
                                }
                        }
                }
        } else if(!is_last(i) && is_free(i)) join_memblocks(ptr, i);

        if(p && is_free(p)) join_memblocks(p, ptr);
}

 /* Splits memblock in two, creating new one with specified size and a second one with the rest of 
available space */
static void split_memblock(memblock_t* const ptr, const size_t split_size, const size_t prev)
{
        /* split_size is already aligned */
        size_t all = get_memblock_size(ptr);
        size_t piece;
        memblock_t* next;

        if(all <= split_size) return;
        if((all - split_size) < sizeof(memblock_t)) return;

        piece = all - split_size - sizeof(memblock_t);
        if(piece < TINY_MEMBLOCK_SIZE) return;

        init_memblock(ptr, split_size, prev);

        next = get_next_memblock(ptr);

        init_memblock(next, piece, split_size);
        set_free(next);
}

void* _malloc(size_t query)
{
        if (query <= 0){
                return NULL;
        }
        else{

        memblock_t *i = (memblock_t*)(HEAP_START);
        void *prevp, *next, *result;
        /* Holds the size of previous memblock, since we end the loop at the one that has 0 in prev_size 
*/
        size_t prev = 0, mmap_region_1, mmap_region_2;

        /* If it's too small or misaligned, we allocate more bytes to align it*/
        query = align_data_size(query);
        if(HEAP_START == NULL) return heap_init(query);

        /* Trying to find a fresh free block instead of actually mmaping it or increasing fragmentation*/
        while(!is_last(i))
        {
                prev = get_memblock_size(i);

                if(is_free(i))
                {
                        if(i->size >= query)
                        {
                                /* Split the rest that we don't need */
                                split_memblock(i, query, i->prev_size);
                                set_in_use(i);
                                return get_memptr(i);
                                /* if that free block is too small but is the last one, we forcibly rewrite 
it */
                        } else if (is_last(get_next_memblock(i)))
                        {
                                prev = i->prev_size;
                                break;
                        }
                }

                i = get_next_memblock(i);
        }

        /* Didn't find a free block
         * Now we might have to mmap a memory region
         * Since this is an expensive operation we want to use it as rarely as possible*/

        /* We start by checking the mmap 'regions' of the previous memblock and the end of our memblock */
        if((uint8_t*)i != HEAP_START) prevp = (uint8_t*)i - 1; else prevp = HEAP_START;
        next = calculate_next_memblock(i, query);

        mmap_region_1 = (size_t)(prevp) / MMAP_PREF_SIZE;
        mmap_region_2 = (size_t)(next) / MMAP_PREF_SIZE;

        /* If they differ, we have to mmap more */
        if(mmap_region_1 < mmap_region_2)
        {
                result = (memblock_t*)mmap((uint8_t*)((mmap_region_1+1)*MMAP_PREF_SIZE), (mmap_region_2 -
mmap_region_1) * MMAP_PREF_SIZE, PROT_READ | PROT_WRITE, MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
                if(result == MAP_FAILED) return NULL;
        }
        init_memblock(i, query, prev);

        /* In theory, linux kernel mmap syscall should zero out the bytes, but just in case */
        ((memblock_t*)next)->prev_size = 0;
        return get_memptr(i);

        }
}

/*
 * User's free(), the actual work is performed in set_free()
 */
void _free(void* ptr)
{
        memblock_t *mptr = (memblock_t*)((uint8_t*)ptr - sizeof(memblock_t));
        set_free(mptr);
}

void* _realloc(void* ptr, size_t query)
{
        memblock_t *next;
        memblock_t *i;
        size_t curr_size;
        size_t available;
        void* result = ptr;

        if(ptr == NULL) return _malloc(query);
        if(query == 0 && ptr != NULL)
        {
                _free(ptr);
                return ptr;
        }

        i = (memblock_t*)((uint8_t*)ptr - sizeof(memblock_t));
        next = get_next_memblock(ptr);
        curr_size = get_memblock_size(i);
        query = align_data_size(query);

        if(query == curr_size) return result;
        if(query > curr_size)
        {
                if(!is_last(next) && is_free(next))
                {
                        available = get_memblock_size(next) + sizeof(memblock_t);
                        if(available + curr_size >= query)
                        {
                                join_memblocks(i, next);
                                split_memblock(i, query, i->prev_size);
                                return result;
                        }
                }

                result = _malloc(query);
                memcpy(result, ptr, curr_size);
                set_free(i);
                return result;
        }
        split_memblock(i, query, i->prev_size);
        return result;
}

/*
 * Initialize the heap, mmaping the first region and setting HEAP_START ptr
 * Uses 0xDEADBEEF as dummy previous size, since prev_size of 0 indicates the end of memblock chain
 * */
void* heap_init(size_t initial_size)
{
        memblock_t *ptr;
        HEAP_START = (uint8_t*)mmap((uint8_t*)0x04040000, MMAP_PREF_SIZE, PROT_READ | PROT_WRITE,
MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
        if(HEAP_START == MAP_FAILED) return NULL;

        ptr = (memblock_t*)(HEAP_START);
        init_memblock(ptr, initial_size, 0xDEADBEEF);

        return get_memptr(ptr);
}
```

Лабораторная работа № 8
«Сепия-фильтр для изображения»

Текст исходной программы:

```
extern  _GLOBAL_OFFSET_TABLE_

section .data
      align 16
      byte2float:
      %assign i 0
      %rep  256
            dd %[i].
      %assign i i+1
      %endrep
      align 16
       cn1:
            dd 0.0, 0.393, 0.349, 0.272
      cn2:
            dd 0.0, 0.769, 0.686, 0.543
      cn3:
            dd 0.0, 0.189, 0.168, 0.131

      %macro firstfill 3
            lea rax, [rdi + %2 + 3*%3]
            xor rcx, rcx
            mov byte cl, [rax]
            lea rax, [rel byte2float]
            lea rax, [rax + rcx*4]
            ;lea rax, [rcx*4 + rel byte2float wrt ..gotoff]
            movss %1, [rax]
            shufps %1, %1, 00h

            lea rax, [rdi + %2 + 3]
            xor rcx, rcx
            mov byte cl, [rax]
            lea rax, [rel byte2float]
            lea rax, [rax + rcx*4]
            ;lea rax, [rcx*4 + rel byte2float wrt ..gotoff]
            movss xmm7, [rax]
            movss %1, xmm7
      %endmacro

      %macro fill_cn 3
            lea rax, [rel cn%2]
            movaps %1, [rax]
            lea rax, [rax + %3*4]
            ;lea rax, [%3*4 + rel cn%2 wrt ..gotpc]
            movss xmm7, [rax]
            movss %1, xmm7
      %endmacro
      %macro p1_2byte 0
            cvttss2si rax, xmm3                 ;r1
            cmp rax, 255
            jle %%skip
            mov rax, 255
      %%skip:
            dec rsp
            mov byte [rsp], al
      %endmacro

      %macro float2byte 2
            shufps %1, %1, 39h
            cvttss2si rax, %1
            cmp rax, 255
            jle %%skip
            mov rax, 255
      %%skip:
            lea rdx, [rdi + %2]
            mov byte [rdx], al
      %endmacro

section .text
      global sepia_filter_asm:function
      sepia_filter_asm:
            push rbp
            mov rbp, rsp
            push rbx

            ; Computing the number of pixels
            mov dword eax, [rdi]                      ;width -> rax
            lea rdx, [rdi+4]                          ;height -> rdx
            mov dword edx, [rdx]
            mul edx                                   ;rax *= rdx -> width *= height

            ; Load pixel pointer into rdi and compute the end
            lea rdi, [rdi + 8]
            mov rdi, [rdi]
            lea rsi, [3*rax]
            add rsi, rdi

      .loop:
            cmp rdi, rsi
            jge .end

            firstfill xmm0, 0, 0          ;r0r0r0r1
            firstfill xmm1, 1, 0          ;g0g0g0g1
            firstfill xmm2, 2, 0          ;b0b0b0b1

            fill_cn xmm3, 1, 1            ;c30 c20 c10 c10
            fill_cn xmm4, 2, 1            ;c31 c21 c11 c11
            fill_cn xmm5, 3, 1            ;c32 c22 c12 c12

            mulps xmm3, xmm0              ;r0*c30 r0*c20 r0*c10 r1*c10
            mulps xmm4, xmm1              ;g0*c31 g0*c21 g0*c11 g1*c11
            mulps xmm5, xmm2              ;b0*c32 b0*c22 b0*c12 b1*c12
            addps xmm3, xmm4
            addps xmm3, xmm5              ;b0 g0 r0 r1

            p1_2byte
            float2byte xmm3, 0
            float2byte xmm3, 1
            float2byte xmm3, 2

            firstfill xmm0, 0, 2          ;r2r2r2r1
            firstfill xmm1, 1, 2          ;g2g2g2g1
            firstfill xmm2, 2, 2          ;b2b2b2b1

            fill_cn xmm3, 1, 2            ;c30 c20 c10 c20
            fill_cn xmm4, 2, 2            ;c31 c21 c11 c21
            fill_cn xmm5, 3, 2            ;c32 c22 c12 c22

            mulps xmm3, xmm0              ;r2*c30 r2*c20 r2*c10 r1*c20
            mulps xmm4, xmm1              ;g2*c31 g2*c21 g2*c11 g1*c21
            mulps xmm5, xmm2              ;b2*c32 b2*c22 b2*c12 b1*c22
            addps xmm3, xmm4
            addps xmm3, xmm5              ;b2 g2 r2 g1

            p1_2byte
            float2byte xmm3, 6
            float2byte xmm3, 7
            float2byte xmm3, 8

            firstfill xmm0, 0, 3          ;r3r3r3r1
            firstfill xmm1, 1, 3          ;g3g3g3g1
            firstfill xmm2, 2, 3          ;b3b3b3b1

            fill_cn xmm3, 1, 3            ;c30 c20 c10 c30
            fill_cn xmm4, 2, 3            ;c31 c21 c11 c31
            fill_cn xmm5, 3, 3            ;c32 c22 c12 c32

            mulps xmm3, xmm0              ;r3*c30 r3*c20 r3*c10 r1*c30
            mulps xmm4, xmm1              ;r3*c31 r3*c21 r3*c11 g1*c31
            mulps xmm5, xmm2              ;r3*c31 r3*c22 r3*c12 b1*c32
            addps xmm3, xmm4
            addps xmm3, xmm5              ;b3 g3 r3 b1

            p1_2byte
            float2byte xmm3, 9
            float2byte xmm3, 10
            float2byte xmm3, 11

            ;Unwinding stack to get pixel 1 values r1 g1 b1
            lea rdx, [rdi + 5]
            mov byte al, [rsp]
            mov byte [rdx], al
            inc rsp
            dec rdx
            mov byte al, [rsp]
            mov byte [rdx], al
            inc rsp
            dec rdx
            mov byte al, [rsp]
            mov byte [rdx], al
            inc rsp

            add rdi, 12
            jmp .loop
      .end:
            pop rbx
            leave
            ret
```