Федеральное государственное автономное образовательное учреждение высшего образования
Университет ИТМО

Кафедра Вычислительной Техники

Дисциплина: Низкоуровневое программирование

Лабораторная работа №7

Выполнил: **Доморацкий Эридан Алексеевич**

Группа: **P33113**

Преподаватель: **Логинов Иван Павлович**

2020г

Задание

13.11 Assignment: Custom Memory Allocator

In this assignment, we are going to implement our own version of `malloc` and `free` based on the memory mapping system call `mmap` and a linked list of chunks of arbitrary sizes. It can be viewed as a simplified version of a memory manager typical for the standard C library and shares most of its weaknesses.

For this assignment, the usage of `malloc`/`calloc`, `free` and `realloc` is forbidden.

Выполнение

```c
// mem.h

#pragma once

#include <stdbool.h>
#include <stddef.h>

#pragma pack(push, 1)
struct mem {
   struct mem * next;
   size_t capacity;
   bool is_free;
};
#pragma pack(pop)

void * malloc(size_t query);
void free(void * mem);

// mem.c

#include "mem.h"

#include <stdint.h>
#include <sys/mman.h>

#define HEAP_START ((void *) 0x04040000)
#define HEAP_SIZE ((size_t) 4 * 1024)

#define BLOCK_MIN_SIZE ((size_t) 128 - sizeof(struct mem))

#ifndef MAP_ANONYMOUS
# define MAP_ANONYMOUS 0x20
#endif

void * heap_mmap(void * addr, bool * strict) {
   void * heap = mmap(addr, HEAP_SIZE, PROT_READ | PROT_WRITE,
       MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED, -1, 0);

   if (strict) {
       *strict = (heap != MAP_FAILED);
   }

   if (heap != MAP_FAILED) {
       return heap;
   }

   return mmap(NULL, HEAP_SIZE, PROT_READ | PROT_WRITE,
       MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
}

struct mem mem_create(size_t capacity) {
   struct mem mem;

   mem.next = NULL;
   mem.capacity = capacity;
   mem.is_free = true;

   return mem;
}
```

```c
void * heap_init(void * addr) {
   struct mem * heap = heap_mmap(addr, NULL);

   if (heap == MAP_FAILED) {
       return NULL;
   }

   *heap = mem_create(HEAP_SIZE - sizeof(struct mem));
   return heap;
}

void heap_blocks_merge(struct mem * heap) {
   while (heap->next && heap->next->is_free) {
       if ((uint8_t *) heap->next == (uint8_t *) heap + sizeof(struct mem) + heap->capacity) {
           heap->capacity += sizeof(struct mem) + heap->next->capacity;
           heap->next = heap->next->next;
       }
   }
}

void * fetch_free_block(struct mem * heap) {
   while (!heap->is_free) {
       if (!heap->next) {
           if (!(heap->next = heap_init((uint8_t *) heap + sizeof(struct mem) + heap->capacity)))
{
               return NULL;
           }
       }

       heap = heap->next;
   }

   heap_blocks_merge(heap);
   return heap;
}

void * malloc(size_t query) {
   static struct mem * heap = NULL;
   struct mem * current;
   struct mem * new;
   bool mmap_strict;

   if (!heap) {
       if (!(heap = heap_init(HEAP_START))) {
           return NULL;
       }
   }

   if (query < BLOCK_MIN_SIZE) {
       query = BLOCK_MIN_SIZE;
   }

   /* fetch first at least capacity block */

   current = heap;
   while ((current = fetch_free_block(current)) && current->capacity < query) {
       if (!current->next) {
           new = heap_mmap(((uint8_t *) current) + sizeof(struct mem) + current->capacity,
&mmap_strict);

           if (new == MAP_FAILED) {
               return NULL;
           }

           if (mmap_strict) {
               current->capacity += HEAP_SIZE;
           } else {
               *new = mem_create(HEAP_SIZE - sizeof(struct mem));
               current->next = new;
               current = new;
           }
       } else {
           current = current->next;
       }
   }
```

```
    if (!current) {
        return NULL;
    }

    /* split if can */
    if (query + sizeof(struct mem) + BLOCK_MIN_SIZE <= current->capacity) {
        new = (struct mem *) (((uint8_t *) current) + sizeof(struct mem) + query);
        *new = mem_create(current->capacity - query - sizeof(struct mem));

        current->capacity = query;
        current->next = new;
    }

    current->is_free = false;
    return (uint8_t *) current + sizeof(struct mem);
}

void free(void * mem) {
    struct mem * heap = (struct mem *) ((uint8_t *) mem - sizeof(struct mem));

    heap->is_free = true;
    heap_blocks_merge(heap);
}

// mem_debug.h

#pragma once

#include <stdio.h>

#include "mem.h"

#define DEBUG_FIRST_BYTES 4

void memalloc_debug_struct_info(FILE * f, struct mem const * const address);
void memalloc_debug_heap(FILE * f, struct mem const * ptr);

// mem_debug.c

#include "mem_debug.h"

#include <stdint.h>

void memalloc_debug_struct_info(FILE * f, struct mem const * const address) {
    size_t i;

    fprintf(f, "start: %p\nsize: %lu\nis_free: %d\n",
            (void *) address,
            address->capacity,
            address->is_free);

    for (i = 0; i < DEBUG_FIRST_BYTES && i < address->capacity; ++i) {
        fprintf(f, "%X", (int) ((char *) address)[sizeof(struct mem) + i]);
    }

    putc('\n', f);
}

void memalloc_debug_heap(FILE * f, struct mem const * ptr) {
    for(; ptr; ptr = ptr->next) {
        memalloc_debug_struct_info(f, ptr);
    }
}

int main() {
    void * mem1 = malloc(4 * 1024 - sizeof(struct mem));

    uint32_t * mem;
    uint32_t ** mems;
    uint32_t i;

    memalloc_debug_struct_info(stderr, (const struct mem *) ((uint8_t *) mem1 - sizeof(struct
mem)));
    mem = malloc(sizeof(uint32_t));

    memalloc_debug_struct_info(stderr, (const struct mem *) ((uint8_t *) mem1 - sizeof(struct
mem)));
```

```
    scanf("%u", mem);
    mems = malloc(sizeof(uint32_t *) * *mem);

    memalloc_debug_struct_info(stderr, (const struct mem *) ((uint8_t *) mem1 - sizeof(struct
mem)));

    for (i = 0; i < *mem; ++i) {
        mems[i] = malloc(sizeof(uint32_t));
        *(mems[i]) = i;

        printf("%u\n", *(mems[i]));
        memalloc_debug_heap(stderr, (const struct mem *) ((uint8_t *) mem - sizeof(struct mem)));
    }

    scanf("%*c");

    for (i = 0; i < *mem; ++i) {
        free(mems[i]);
    }

    free(mems);
    free(mem);

    memalloc_debug_heap(stderr, (const struct mem *) ((uint8_t *) mem - sizeof(struct mem)));
    return 0;
}
```

Вывод

В результате выполнения лабораторной работы были реализованы собственные реализации стандартный функций C для работы с кучей malloc и free. В ходе выполнения был применён механизм ОС для выделения памяти mmap и статические переменные для хранения начала цепочки блоков кучи.