НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. І. СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ ТА СПЕЦІАЛІЗОВАНИХ КОМП’ЮТЕРНИХ СИСТЕМ

**Лабораторна робота №1  
з дисципліни «Системне програмне забезпечення»**

Виконав  
студент 4-го курсу  
групи КВ-41  
Курач Віктор

Київ – 2017

**Постановка задачі**

Розробити алокатор загального призначення, беручи до уваги наступні умови:

* області пам’яті можна виділяти будь-яким доступним способом;
* функції мають відповідати наступним прототипам:

void \*mem\_alloc(size\_t size);

void \*mem\_realloc(void \*addr, size\_t size);

void mem\_free(void \*addr);

* адреси пам’яті, що повертаються функціями, мають бути вирівняні на границю в 4 байти;
* спробувати зменшити час пошуку вільного блоку пам’яті та час звільнення зайнятого блоку;
* спробувати зменшити фрагментацію пам’яті.
* написати функцію mem\_dump(), яка має виводити на консоль стан областей пам’яті.

**Опис алгоритму**

Заголовок блоку пам’яті (8 байт) містить ознаку зайнятості блоку (старший біт), зміщення в області пам’яті попереднього блоку (4 байти мінус 1 біт) та зміщення наступного блоку (4 байти). На початку роботи алокатора в перші 4 байти області пам’яті записано нуль, а в наступні 4 байти – розмір області пам’яті.

Блок пам’яті вважається придатним для алокації, якщо його ознака зайнятості рівна нулю, а доступний об’єм пам’яті блоку як мінімум на 8 біт більший за розмір пам’яті, яку треба виділити. Якщо даний блок не придатний для алокації, відбувається перехід по зміщенню на наступний блок. Якщо зміщення дорівнює об’єму області пам’яті (даний блок є останнім), повертається ознака невдалої алокації (NULL).

Коли блок знайдено, відбувається його розділення: у нього записується заголовок (перші 8 біт), після якого розміщуються дані. Після даних розміщується 8 біт заголовку блоку, який утворюється з вільної пам’яті, що залишилась після розділення. Саме для цього необхідні додаткові 8 біт при пошуку блоку, придатного для алокації.

Наступний крок – модифікація заголовків сусідніх блоків таким чином, щоби їх зміщення вказувало на перший біт заголовка виділеного блоку. Останній крок – обчислення адреси виділеного блоку (адреса області пам’яті + зміщення).

Звільнення пам’яті відбувається наступним чином. За адресою пам’яті, яку треба звільнити, знаходиться її зміщення в області пам’яті. Далі відбувається занулення даних. Після цього проводиться перевірка, чи є попередній блок пам’яті вільним. Якщо так – відбувається об’єднання блоків пам’яті шляхом модифікації зміщень у попередньому блоці та видаленням зміщень зі щойно звільненого блока. Після цього за зміщення звільненого блока приймається зміщення попереднього. Далі аналогічно відбувається об’єднання з наступним блоком, якщо він вільний.

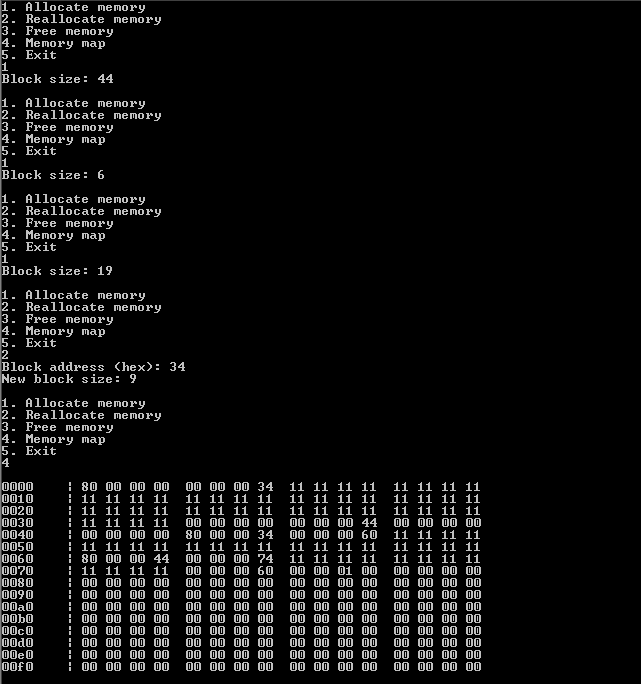
Реалокація пам’яті відбувається так:

1. Знайти зміщення блоку пам’яті за його адресою.
2. Спробувати виділити нову пам'ять заданого розміру за вищеописаним алгоритмом.
3. Якщо виділення завершилося успішно, очистити «стару» пам'ять, інакше повернути NULL.
4. Повернути адресу «нової» пам’яті.

Перевагами даного алгоритму є відсутність додаткових структур даних крім області пам’яті та підвищена швидкодія за рахунок використання зміщень блоків пам’яті замість їх повних адрес.

Головним недоліком є можлива велика надлишковість службової інформації. Наприклад, якщо необхідно виділити 4 байти пам’яті і знайдено вільний блок розміром 20 байт, то в перші його 8 байт буде записано заголовок, наступні 4 байти залишаються для даних, а останні 8 байт – для заголовку блоку, розмір якого 0 байт, оскільки далі знаходиться заголовок наступного (раніше виділеного) блоку.

**Приклад роботи**

****

**Лістинг**

1. Allocator.h

#include <stdio.h>

#include <malloc.h>

#define MEM\_SIZE 256

unsigned char\* MEMORY;

void \*mem\_alloc(size\_t size);

void \*mem\_realloc(void \*addr, size\_t size);

void mem\_free(void\* addr);

void mem\_dump();

1. Allocator.c

#include "allocator.h"

void \*mem\_alloc(size\_t size) {

int i = 0, k, div;

int prev = MEMORY[i+3] + (MEMORY[i+2] << 8) + (MEMORY[i+1] << 16) + (MEMORY[i] << 24);

int next = MEMORY[i+7] + (MEMORY[i+6] << 8) + (MEMORY[i+5] << 16) + (MEMORY[i+4] << 24);

size\_t avail\_memory = next - i - 8;

if (size == 0) return NULL;

// Searching for available memory block

while (size + 8 > avail\_memory || (MEMORY[i] & 0x80) != 0) {

prev = i;

i = next;

if (i >= MEM\_SIZE) return NULL;

next = MEMORY[i+7] + (MEMORY[i+6] << 8) + (MEMORY[i+5] << 16) + (MEMORY[i+4] << 24);

avail\_memory = next - i - 8;

}

div = i + size + 8;

while (div % 4 != 0) ++div;

// Updating new block caption

MEMORY[i] = (prev & 0xff000000) >> 24;

MEMORY[i] = MEMORY[i] | 0x80;

MEMORY[i+1] = (prev & 0xff0000) >> 16;

MEMORY[i+2] = (prev & 0xff00) >> 8;

MEMORY[i+3] = prev & 0xff;

MEMORY[i+4] = (div & 0xff000000) >> 24;

MEMORY[i+5] = (div & 0xff0000) >> 16;

MEMORY[i+6] = (div & 0xff00) >> 8;

MEMORY[i+7] = div & 0xff;

// Updating data

for (k = i + 8; k < div; ++k) MEMORY[k] = 0x11;

// Division of memory

MEMORY[div] = (i & 0xff000000) >> 24;

MEMORY[div+1] = (i & 0xff0000) >> 16;

MEMORY[div+2] = (i & 0xff00) >> 8;

MEMORY[div+3] = i & 0xff;

MEMORY[div+4] = (next & 0xff000000) >> 24;

MEMORY[div+5] = (next & 0xff0000) >> 16;

MEMORY[div+6] = (next & 0xff00) >> 8;

MEMORY[div+7] = next & 0xff;

// Updating prev block caption

if (prev != 0) {

MEMORY[prev+4] = (i & 0xff000000) >> 24;

MEMORY[prev+5] = (i & 0xff0000) >> 16;

MEMORY[prev+6] = (i & 0xff00) >> 8;

MEMORY[prev+7] = i & 0xff;

}

// Updating next block caption

if (next + 8 <= MEM\_SIZE) {

MEMORY[next] = (div & 0xff000000) >> 24;

MEMORY[next+1] = (div & 0xff0000) >> 16;

MEMORY[next+2] = (div & 0xff00) >> 8;

MEMORY[next+3] = div & 0xff;

}

return MEMORY + i;

}

void \*mem\_realloc(void \*addr, size\_t size) {

int i = 0;

void\* new\_addr;

while (MEMORY + i != addr) {

++i;

if (i == MEM\_SIZE) return NULL;

}

new\_addr = mem\_alloc(size);

if (new\_addr != NULL) mem\_free(addr);

return new\_addr;

}

void mem\_free(void\* addr) {

int i = 0, k, prev, next, nextnext;

unsigned char alloc\_flag;

// Checking whether addr is memory block address

while (MEMORY + i != addr) {

++i;

if (i == MEM\_SIZE) return;

}

// Freeing memory

MEMORY[i] = MEMORY[i] & 0x7f;

prev = MEMORY[i+3] + (MEMORY[i+2] << 8) + (MEMORY[i+1] << 16) + (MEMORY[i] << 24);

next = MEMORY[i+7] + (MEMORY[i+6] << 8) + (MEMORY[i+5] << 16) + (MEMORY[i+4] << 24);

for (k = i + 8; k < next; ++k) MEMORY[k] = 0;

// Merging with previous free memory block

if ((MEMORY[prev] & 0x80) == 0) {

for (k = 4; k < 8; ++k) MEMORY[prev+k] = MEMORY[i+k];

for (k = 0; k < 8; ++k) MEMORY[i+k] = 0;

alloc\_flag = MEMORY[next];

MEMORY[next] = ((prev & 0xff000000) >> 24) | (alloc\_flag & 0x80);

MEMORY[next+1] = (prev & 0xff0000) >> 16;

MEMORY[next+2] = (prev & 0xff00) >> 8;

MEMORY[next+3] = prev & 0xff;

i = prev;

}

// Merging with next free memory block

if ((MEMORY[next] & 0x80) == 0) {

nextnext = MEMORY[next+7] + (MEMORY[next+6] << 8) + (MEMORY[next+5] << 16) + (MEMORY[next+4] << 24);

for (k = 4; k < 8; ++k) MEMORY[i+k] = MEMORY[next+k];

alloc\_flag = MEMORY[nextnext];

for (k = 0; k < 4; ++k) MEMORY[nextnext+k] = MEMORY[next+k];

MEMORY[nextnext] = MEMORY[nextnext] | (alloc\_flag & 0x80);

for (k = 0; k < 8; ++k) MEMORY[next+k] = 0;

}

}

void mem\_dump() {

int i;

for (i = 0; i < MEM\_SIZE; ++i) {

if (i % 16 == 0) printf("\n%0\*x\t|", 4, i);

if (i % 4 == 0) printf(" ");

printf("%0\*x ", 2, MEMORY[i]);

}

printf("\n");

}

1. Main.c

#include "allocator.h"

void init() {

int i;

MEMORY = (unsigned char\*)malloc((MEM\_SIZE + 4) \* sizeof(unsigned char));

for (i = 0; i < MEM\_SIZE; ++i) MEMORY[i] = 0;

MEMORY[7] = MEM\_SIZE & 0xff;

MEMORY[6] = (MEM\_SIZE & 0xff00) >> 8;

MEMORY[5] = (MEM\_SIZE & 0xff0000) >> 16;

MEMORY[4] = (MEM\_SIZE & 0xff000000) >> 24;

}

void menu() {

int command = 0, block\_size = 0, block\_address = 0;

while (command != 5) {

printf("1. Allocate memory\n2. Reallocate memory\n3. Free memory\n4. Memory map\n5. Exit\n");

scanf\_s("%d", &command);

switch (command) {

case 1: {

printf("Block size: ");

scanf\_s("%d", &block\_size);

if (mem\_alloc(block\_size) == NULL)

printf("Allocation failed\n");

break;

} case 2: {

printf("Block address (hex): ");

scanf\_s("%x", &block\_address);

printf("New block size: ");

scanf\_s("%d", &block\_size);

if (mem\_realloc(MEMORY + block\_address, block\_size) == NULL)

printf("Allocation failed\n");

break;

} case 3: {

printf("Block address (hex): ");

scanf\_s("%x", &block\_address);

mem\_free(MEMORY + block\_address);

break;

} case 4: {

mem\_dump();

break;

} case 5: break;

}

printf("\n");

}

}

void end() {

free(MEMORY);

}

main () {

init();

menu();

end();

}