Національний технічний університет України

«Київський політехнічний інститут»

Факультет інформатики і обчислювальної техніки

Кафедра обчислювальної техніки

**Лабораторна робота №1**

**з курсу: «Системне програмування»**

*Виконав:*

студент групи ІО-12

Васильєв О.В.

Залікова книжка №1206

*Перевірив:*

Дифучин А.Ю.

Київ, 2020 р.

**Тема**: аллокатор пам’яті загального призначення (частина 1).

**Мета**: розробити аллокатор загального призначення.

**Опис розроблених алгоритмів**:

1. Ініціалізація загального пулу. Пул блоків пам’яті виділяється за допомогою стандартної функції виділення пам’яті malloc:

Кожен блок виділеної пам’яті містить заголовок:

unsigned is\_free: 1; // bit which shows is this block

// store any information

unsigned data\_size: 15; // block size in bytes [0 - 32767]

mem\_block \*prev\_block; // pointer on next block

mem\_block \*next\_block; // pointer on previous block

};

Відповідно до опису заголовок має займати 1 + 1 + 4 + 4 = 10 байт

Під час ініціалізації виділяється 3 блоки:

* Початковий зайнятий блок-бар’єр нульового розміру.
* Кінцевий зайнятий блок-бар’єр нульового розміру.
* Проміжний вільний блок.

1. Звільнення загального пулу. Виконується за допомогою стандартної функції allocSearchForFreeBlock
2. Виділення пам’яті заданого розміру mem\_alloc(size\_t size). Передбачені 2 алгоритми пошуку потрібного блоку для виділення пам’яті:

* За необхідний вибирається перший знайдений підходящий блок, тобто такий, розмір якого не менший за size.
* За необхідний вибирається підходящий блок з мінімальним розміром.

Якщо блок не знайдений, то повертається NULL, якщо знайдений – то виконується функція виділення пам’яті в заданому блоці.

1. Функція виділення пам’яті в заданому блоці allocFreeBlock (Header \*headerPtr, size\_t size).
2. Якщо розмір блоку дещо більший за size, то він ділиться на 2 блоки.
3. Якщо розмір блоку майже дорівнює size, то блок залишається неподільним. Повертається вказівник на заголовок блоку.
4. Перевиділення пам’яті void \*allocRealloc(allocator \*alloc, void \*data, size\_t size)
5. Якщо addr = NULL, то виконується виділення пам’яті заданого розміру mem\_alloc(size).
6. Якщо розмір блоку, що виділений під данні вказівника addr, більший за size, то перевиділяємо пам’ять блоку за допомогою freeBlockAlloc(headerPtr, size).
7. У протилежному випадку намагаємося об’єднати даний блок з вільним правим (без перенесення даних) і/або лівим (з перенесенням даних) й перевиділити пам’ять блоку.
8. Якщо цього виявилося недостатньо, то знаходимо вільне місце для даних з допомогою mem\_alloc(), переносимо дані й звільняємо даний блок.
9. Звільнення пам’яті mem\_free(void \*addr). Помічаємо блок як вільний і об’єднуємо даний блок с сусідніми вільними блоками.

**Оцінка часу пошуку вільного блоку пам’яті**:

* Алгоритм «перший підходящий блок»: O(N).
* Алгоритм «підходящий з мінімальним розміром»: O(2 \* N).

N – кількість блоків в пулі.

**Оцінка часу звільнення занятого блоку**: O(1).

**Оцінка витрати пам'яті для зберігання службовох інформації:** 12 \* N байт.

N – кількість блоків в пулі.

**Переваги аллокатора:**

1. Невеликий об’єм службової інформації.
2. Малий час виконання звільнення блоку.

**Недоліки аллокатора:**

1. Лінійниний час пошуку вільного блоку (повільно).
2. Можлива передчасна фрагментація пам’яті.

**Лістинг заготовочного файлу mem\_alloc.h:**

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

typedef unsigned char byte;

// struct for header of mem block

typedef struct \_mem\_block mem\_block;

struct \_mem\_block

{

unsigned is\_free: 1; // bit which shows is this block

// store any information

unsigned data\_size: 15; // block size in bytes [0 - 32767]

mem\_block \*prev\_block; // pointer on next block

mem\_block \*next\_block; // pointer on previous block

};

// default length of buffer for allocator

const size\_t INIT\_ALLOCATOR\_DATA\_SIZE = 128;

const size\_t MAX\_ALLOCATOR\_DATA\_SIZE = 32767;

// struct to store information about allocator

typedef struct \_allocator allocator;

struct \_allocator

{

// bool is\_init; // is initializated allocator

mem\_block \*mem\_start; // pointer on first block of data

mem\_block \*mem\_end; // pointer on last block of data

size\_t last\_alloc\_size; // the size of data allocated last time

};

allocator \*ALLOC;

// function that allocates memory for new block and init it

mem\_block \*newBlock(size\_t size, mem\_block \*prev)

{

// create and allocate memory for new block

mem\_block \*new\_block;

new\_block = (mem\_block \*) malloc(sizeof(mem\_block) + size);

// init block with standart values

new\_block->is\_free = true;

new\_block->data\_size = size;

new\_block->prev\_block = prev;

new\_block->next\_block = NULL;

return new\_block;

}

allocator \*allocatorInit()

{

allocator \*alloc = (allocator \*) malloc(sizeof(alloc));

alloc->mem\_start = newBlock(INIT\_ALLOCATOR\_DATA\_SIZE, NULL);

alloc->mem\_end = alloc->mem\_start;

alloc->last\_alloc\_size = INIT\_ALLOCATOR\_DATA\_SIZE;

return alloc;

}

mem\_block \*allocSearchForFreeBlock(allocator \*alloc, size\_t min\_data\_size)

{

// set the start block

mem\_block \*curr\_block = alloc->mem\_start;

while (curr\_block != NULL) { // while we don't reach the end

// if block is free and its size not less than needed

// (block is free and (have the same mem size or size which allow to

// divide block into two blocks))

if ( curr\_block->is\_free &&

((curr\_block->data\_size == min\_data\_size) ||

(curr\_block->data\_size >= min\_data\_size + sizeof(mem\_block) )

)) {

return curr\_block;

}

// move to next block

curr\_block = curr\_block->next\_block;

}

// if we didn't find suitable block then return NULL

return NULL;

}

bool allocIsCanDivideBlock(mem\_block \*block, size\_t size)

{

if (block->data\_size >= sizeof(mem\_block) + size) {

return true;

}

else {

return false;

}

}

// function which divide block into two pieces

// with 'first\_data\_size' data size for first piece

// returns pointer on first block

mem\_block \*allocDivideBlock(mem\_block \*block, size\_t first\_data\_size)

{

if (allocIsCanDivideBlock(block, first\_data\_size)) {

mem\_block \*first\_block = block;

// find position of second block

mem\_block \*second\_block = (mem\_block \*) ((byte \*) block + sizeof(mem\_block) + first\_data\_size);

// init second block

second\_block->is\_free = true;

second\_block->data\_size = first\_block->data\_size - sizeof(mem\_block) - first\_data\_size;

second\_block->prev\_block = first\_block;

second\_block->next\_block = first\_block->next\_block;

// change 'next\_block' field for first block to second block pointer

first\_block->next\_block = second\_block;

first\_block->is\_free = false;

first\_block->data\_size = first\_data\_size;

// return pointer on first block

return first\_block;

}

// if data size not enogh return NULL

return NULL;

}

bool allocIsCanConcateBlocks(mem\_block\* first\_block)

{

if (first\_block == NULL || first\_block->next\_block == NULL) {

return false;

}

mem\_block\* second\_block = first\_block->next\_block;

if (first\_block->is\_free &&

second\_block->is\_free &&

(byte \*) second\_block - (byte \*) first\_block == sizeof(mem\_block) + first\_block->data\_size &&

first\_block->data\_size <= MAX\_ALLOCATOR\_DATA\_SIZE - second\_block->data\_size - sizeof(mem\_block))

{

return true;

}

else {

return false;

}

}

// function that try to concatenate two free blocks which located together

// second block located after first block

mem\_block \*allocTryConcatBlocks(mem\_block\* first\_block)

{

// check are they free, are they located together and is the result date size will be less than max data size

if (allocIsCanConcateBlocks(first\_block))

{

mem\_block\* second\_block = first\_block->next\_block;

// connect block

first\_block->next\_block = second\_block->next\_block;

// change data size

first\_block->data\_size += sizeof(mem\_block) + second\_block->data\_size;

}

else {

// if we can't concatenate blocks, return NULL

return NULL;

}

}

// functon that concat all free blocks which located together

mem\_block \*allocConcatAllPossibleBlocks(mem\_block\* block)

{

mem\_block \*last\_block = block; // last successful block

// try to concat on the left way

block = allocTryConcatBlocks(block->prev\_block);

while (block != NULL) {

last\_block = block;

block = allocTryConcatBlocks(block->prev\_block);

}

// try to concat on the right way

block = allocTryConcatBlocks(last\_block);

while (block != NULL) {

last\_block = block;

block = allocTryConcatBlocks(block);

}

return last\_block; // returns last successful block

}

// function that returns pointer on raw data which located after header of block

void \*memBlockPtrToDataPtr(mem\_block \*block)

{

// move pointer to raw data using size of header

return (void \*) ((byte \*) block + sizeof(mem\_block));

}

// function that calculate pointer on mem\_block using header offset

mem\_block \*dataPtrToMemBlockPtr(void \*data)

{

printf("===%ld===\n", \*((int \*) data));

return (mem\_block \*) ((byte\*) data - sizeof(mem\_block));

}

// returns pointer on new block

void \*allocGetBlockData(allocator \*alloc, size\_t size)

{

// allocated size can't be bigger than max data size in single block

if (size > MAX\_ALLOCATOR\_DATA\_SIZE) {

return NULL;

}

mem\_block \*suit\_pos = allocSearchForFreeBlock(alloc, size);

if (suit\_pos != NULL) {

if (suit\_pos->data\_size == size) {

suit\_pos->is\_free = false;

return memBlockPtrToDataPtr(suit\_pos);

}

else {

mem\_block \*suit\_block = allocDivideBlock(suit\_pos, size);

suit\_block->is\_free = false;

return memBlockPtrToDataPtr(suit\_block);

}

}

else {

size\_t size\_to\_alloc = alloc->last\_alloc\_size \* 2;

if (size\_to\_alloc < size) {

size\_to\_alloc = size;

}

if (size\_to\_alloc > MAX\_ALLOCATOR\_DATA\_SIZE) {

size\_to\_alloc = MAX\_ALLOCATOR\_DATA\_SIZE;

}

alloc->last\_alloc\_size = size\_to\_alloc;

void \*new\_block = newBlock(size\_to\_alloc, alloc->mem\_end);

alloc->mem\_end->next\_block = new\_block;

alloc->mem\_end = new\_block;

// recursive function call after memory allocation

return allocGetBlockData(alloc, size);

}

}

void allocFreeBlock(void \*data)

{

// get the pointer on mem\_block struct

mem\_block \*block = dataPtrToMemBlockPtr(data);

block->is\_free = true;

// concat all possible blocks

allocConcatAllPossibleBlocks(block);

}

mem\_block \*allocCopyBlockData(mem\_block\* from\_block, mem\_block \*dest\_block)

{

if (dest\_block->data\_size > from\_block->data\_size) {

return NULL;

}

else {

byte \*from\_data = (byte \*) memBlockPtrToDataPtr(from\_block);

byte \*dest\_data = (byte \*) memBlockPtrToDataPtr(dest\_block);

for (unsigned int i = 0; i < from\_block->data\_size; i++) {

dest\_data[i] = from\_data[i];

}

return dest\_block;

}

}

mem\_block \*allocReduceBlock(allocator\* alloc, mem\_block \*block, size\_t size)

{

if (size > block->data\_size) {

return NULL;

}

mem\_block \*res\_divided = allocDivideBlock(block, size);

if (res\_divided != NULL) {

return res\_divided;

}

else {

mem\_block \*new\_block = dataPtrToMemBlockPtr(allocGetBlockData(alloc, size));

allocCopyBlockData(block, new\_block);

allocFreeBlock(memBlockPtrToDataPtr(block));

return new\_block;

}

}

mem\_block \*allocEnlargeBlock(allocator\* alloc, mem\_block \*block, size\_t size)

{

if (size < block->data\_size) {

return NULL;

}

if (allocIsCanConcateBlocks(block))

{

if (block->data\_size + sizeof(mem\_block) + block->next\_block->data\_size == size) {

return allocTryConcatBlocks(block);

}

else if (block->data\_size + block->next\_block->data\_size > size) {

mem\_block \*concated\_block = allocTryConcatBlocks(block);

return allocDivideBlock(concated\_block, size);

}

}

mem\_block \*new\_block = dataPtrToMemBlockPtr(allocGetBlockData(alloc, size));

allocCopyBlockData(block, new\_block);

allocFreeBlock(memBlockPtrToDataPtr(block));

return new\_block;

}

void \*allocRealloc(allocator \*alloc, void \*data, size\_t size)

{

if (data == NULL) {

return allocGetBlockData(ALLOC, size);

}

else {

mem\_block \*block = dataPtrToMemBlockPtr(data);

if (size < block->data\_size) {

return memBlockPtrToDataPtr(allocReduceBlock(alloc, block, size));

}

else if (size > block->data\_size) {

return memBlockPtrToDataPtr(allocEnlargeBlock(alloc, block, size));

}

else {

return data;

}

}

}

// retruns pointer on allocated block

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

{

return allocGetBlockData(ALLOC, size);

}

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

{

return allocRealloc(ALLOC, addr, size);

}

void mem\_free(void \*addr)

{

allocFreeBlock(addr);

}

void mem\_dump()

{

printf("==========MEM DUMP==========\n");

mem\_block \*block = ALLOC->mem\_start;

while (block != NULL) {

printf("----\n");

printf("Byte: %ld\n", (byte \*) block - (byte \*)ALLOC);

printf("Is free: %d\n", block->is\_free);

printf("Data size: %d\n", block->data\_size);

printf("----\n");

block = block->next\_block;

}

}

int main(int argc, char const \*argv[])

{

// set the global variable

ALLOC = allocatorInit();

mem\_dump();

int \*x = mem\_alloc(4);

\*x = 3;

mem\_dump();

printf("x = %d\n", \*x);

mem\_free(x);

mem\_dump();

//

//

// byte \*new\_x = (byte \*) mem\_realloc(x, 1);

// mem\_dump();

// printf("x = %d\n", \*x);

//

// mem\_free(new\_x);

// mem\_dump();

// printf("%ld", sizeof(mem\_block));

// int \*x = mem\_alloc(4);

// \*x = 2;

// mem\_dump();

// byte \*new\_x = (byte \*) mem\_realloc(x, 1);

// mem\_dump();

//

// mem\_free(x);

// mem\_dump();

// printf("%p %p\n", (byte\*)dataPtrToMemBlockPtr(x), (byte\*)dataPtrToMemBlockPtr(y));

//

// mem\_free(x);

// mem\_dump();

// mem\_free(y);

// mem\_dump();

return 0;

}

**Приклад роботи аллокатора:**

