Національний Технічний Університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет Інформатики та Обчислювальної Техніки

Кафедра АСОІУ

**Лабораторна робота №1**

Виконав:

студент 3 курсу ФІОТ

групи ІС-72

Корнієнко В.

Перевірив:

Дифучин А.

Київ 2020

***Тема:*** аллокатор пам’яті загального призначення.

***Мета:*** розробити аллокатор загального призначення.

***Опис розроблених алгоритмів:***

Аллокатор пам'яті загального призначення повинен виконувати, принаймні, три завдання: виділення блоку пам'яті заданого розміру, звільнення виділеного блоку пам'яті і зміна розміру виділеного блоку пам'яті. Ці функції складають інтерфейс аллокатора пам'яті загального призначення:

* void \*mem\_alloc(size\_t size) – шукає блок вільної пам’яті заданого розміру. Якщо такий блок знайдений в пам’яті – створюється блок пам’яті, якщо ні то до виділеної пам’яті додається пам’ять і створюється блок пам’яті.
* void \*mem\_realloc(void \*address, size\_t size) – якщо address = NULL mem\_realloc працює за принципом mem\_alloc. Якщо ні, створюється блок заданого розміру, дані в нього копіюються зі старого блоку.
* void mem\_free(void \*address) – звільнює блок пам’яті та об’єднує вільні фрагменти пам’яті, що розташовані поруч. Допоміжна функція union\_mem об’єднує всі вільні фрагменти пам’яті у global\_memory.
* void mem\_dump() – виводить статистику пам’яті ( адресу, розмір та дані, що знаходяться в блоках).

Переваги:

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

Недоліки:

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

**Оцінка часу**

O(N) - Оцінка часу пошуку вільної комірки пам’яті і видалення комірки.

N – кількість комірок пам’яті.

**Лістинг програми**

**lab1.cpp**

#include "allocator.h"

memory\_list \*create\_block(size\_t size, size\_t is\_fill)

{

memory\_list \*memory\_current\_block;

memory\_current\_block = (memory\_list \*)malloc(sizeof(memory\_list));

if (memory\_current\_block == NULL || size == 0){ return NULL;}

memory\_current\_block->size = size;

memory\_current\_block->address = is\_fill ? malloc(size \* sizeof(void\*)) : NULL;

memory\_current\_block->is\_busy = !is\_fill;

memory\_current\_block->prev = NULL;

memory\_current\_block->next = NULL;

return memory\_current\_block;

}

memory\_list \*add\_memory(size\_t size)

{

memory\_list \*begin;

memory\_list \*mem\_block;

begin = starting\_memory;

mem\_block = create\_block(size, 1);

while (begin->next != NULL)

{

begin = begin->next;

}

begin->next = mem\_block;

mem\_block->prev = begin;

return mem\_block;

}

memory\_list \*find\_address\_block(void \*address)

{

memory\_list \*current\_block;

current\_block = starting\_memory;

while (current\_block != NULL)

{

if (current\_block->address == address)

{

return current\_block;

}

current\_block = current\_block->next;

}

return NULL;

}

memory\_list \*find\_first\_free\_block(size\_t size)

{

memory\_list \*current\_block;

current\_block = starting\_memory;

while (current\_block != NULL)

{

if (current\_block->is\_busy == 1 && size <= current\_block->size)

{

return current\_block;

}

current\_block = current\_block->next;

}

return NULL;

}

//delete block of memory, used in mem\_free function

void delete\_block(memory\_list\* mem\_block) {

if (mem\_block->next != NULL) {

mem\_block->next->prev = mem\_block->prev;

}

if (mem\_block->prev != NULL) {

mem\_block->prev->next = mem\_block->next;

}

else {

starting\_memory = mem\_block->next;

}

// free all block and data

free(mem\_block->address);

free(mem\_block);

}

//reallocate memory block if it valid. If not, it will allocate memory with the specified size

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

if (address == NULL) {

return mem\_alloc(size);

}

memory\_list\* old\_block;

memory\_list\* new\_block;

old\_block = find\_address\_block(address);

if (old\_block != NULL && size != 0) {

if (find\_first\_free\_block(size) != NULL) {

find\_first\_free\_block(size)->size += old\_block->size < size ? 0 - (size - old\_block->size) : old\_block->size - size;

}

new\_block = create\_block(size, 1);

new\_block->address = memcpy(new\_block->address, old\_block->address, old\_block->size > size ? size : old\_block->size);

new\_block->next = old\_block->next;

new\_block->prev = old\_block->prev;

if (old\_block->prev != NULL) {

old\_block->prev->next = new\_block;

}

else

{

starting\_memory = new\_block;

}

if (old\_block->next != NULL) {

old\_block->next->prev = new\_block;

}

free(old\_block->address);

free(old\_block);

return new\_block->address;

}

else {

return NULL;

}

}

//Clears the memory and removes the block.

void mem\_free(void\* address) {

memory\_list\* mem\_block;

mem\_block = find\_address\_block(address);

if (mem\_block != NULL) {

mem\_block->is\_busy = 1;

free(mem\_block->address);

mem\_block->address = NULL;

if (mem\_block->prev != NULL && mem\_block->prev->is\_busy) {

mem\_block->prev->size += mem\_block->size;

}

else if (mem\_block->next != NULL && mem\_block->next->is\_busy) {

mem\_block->next->size += mem\_block->size;

}

else {

return;

}

delete\_block(mem\_block);

}

else {

return;

}

}

// if size != 0 allocate memory. Create block

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

if (size == 0) {

return NULL;

}

memory\_list\* make\_newblock;

memory\_list\* block\_extract;

make\_newblock = find\_first\_free\_block(size);

if (make\_newblock != NULL) {

make\_newblock->address = malloc(size \* sizeof(void\*));

make\_newblock->is\_busy = 0;

int z = (make\_newblock->size - size);

if (abs(z) > 0) {

block\_extract = create\_block(make\_newblock->size - size, 0);

if (make\_newblock->next != NULL) {

make\_newblock->next->prev = block\_extract;

}

block\_extract->next = make\_newblock->next;

block\_extract->prev = make\_newblock;

make\_newblock->next = block\_extract;

}

make\_newblock->size = size;

return (make\_newblock->address);

}

else {

return (add\_memory(size)->address);

}

}

//displays the memory stats

void mem\_dump()

{

memory\_list\* begin;

begin = starting\_memory;

printf("STATISTICS:\n");

while (begin)

{

printf("Address: %15p| Size: %15lu\n"/\* | Data: %15s\n"\*/, begin->address, begin->size/\*, (char \*)begin->address\*/);

begin = begin->next;

}

}

void union\_mem() {

memory\_list\* begin = starting\_memory;

while (begin != NULL) {

if (begin->is\_busy == 1) {

if (begin->prev != NULL && begin->prev->is\_busy) {

begin->prev->size += begin->size;

delete\_block(begin);

}

else if (begin->next != NULL && begin->next->is\_busy) {

begin->next->size += begin->size;

delete\_block(begin);

}

}

begin = begin->next;

}

}

**main.cpp**

#include "allocator.h"

#include <string.h>

int main()

{

int choice;

void \*address;

size\_t size;

//char data[2048];

starting\_memory = create\_block(2048, 0);

printf("Your initial statistics:");

/\*void \* var = mem\_alloc(5);

void \* var1 = mem\_alloc(1);

mem\_dump();

mem\_realloc(var, 8);

mem\_dump();

mem\_realloc(var1, 10);

mem\_dump();

mem\_free(var);

union\_mem();

mem\_dump();\*/

while (1)

{

mem\_dump();

printf("\n1) Add memmory Block\n2) Realloc memmory Block\n3) Delete memomry Block\n4) Exit the Allocator\n------------------\n");

printf("Input your number please: ");

scanf\_s("%i", &choice);

if(choice == 1){

printf("Input size: ");

scanf\_s("%lu", &size);

mem\_alloc(size);

}

else if(choice == 2){

printf("Input address: ");

scanf\_s("%p", &address);

printf("Input new size: ");

scanf\_s("%lu", &size);

mem\_realloc(address, size);

}

else if(choice == 3){

printf("Input address: ");

scanf\_s("%p", &address);

mem\_free(address);

union\_mem();

}

else if (choice == 4)

{

exit(0);

}

}

return (0);

}

**allocator.h**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Struct of double linked list - memory

typedef struct memory\_list {

void \*address;

size\_t size;

size\_t is\_busy;

struct memory\_list \*prev;

struct memory\_list \*next;

} memory\_list;

//global memory - initialize in main function

memory\_list \*starting\_memory;

//create memory block

memory\_list \*create\_block(size\_t size, size\_t is\_fill);

//adds memory when global memory runs out

memory\_list \*add\_memory(size\_t size);

// find node by address

memory\_list \*find\_address\_block(void \*address);

//find first free\_node->size < size

memory\_list \*find\_first\_free\_block(size\_t size);

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

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

void mem\_free(void \*address);

void union\_mem();

void mem\_dump();

void delete\_block(memory\_list \*mem\_block);