**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

**Факультет інформатики та обчислювальної техніки**

**Кафедра автоматизації систем обробки інформації та управління**

**“Сучасні операційні системи”**

**Лабораторна робота №2**

**Перевірив: Дифучин А. Ю.**

**Виконав: Коноплянка Д.С.**

**Студент гр. ІС-73, ФІОТ,**

**3 курс**

Київ

НТУУ «КПІ ім. І. Сікорського»

2020

**Тема**: Аллокатор пам’яті загального призначення (частина 2).

**Мета**: розробити аллокатор загального призначення.

**Опис розробленого алгоритму**:

1. Виділення пам’яті через створення масиву:

Заголовок займає 22 байт.

Під час ініціалізації виділяється 1 вільний блок, який займає всю доступну пам’ять масиву.

1. Виділення пам’яті заданого розміру mem\_alloc(size\_t size):

перший знайдений підходящий блок, тобто такий,  розмір якого не менший за size. Якщо розмір більший необхідного, блок розбивається на зайнятий і вільний блоки, і користувачу повертається вказівник на початок зайнятого блоку. Якщо потрібний блок не знайдений, то повертається NULL.

1. Перевиділення пам’яті mem\_realloc(void \*addr, size\_t size).

Якщо addr = NULL, то виконується виклик mem\_alloc(size). Інакше, відбувається перевірка сусідніх блоків. Якщо хоча б один з них вільний, відбувається об’єднання блоків. Далі перевіряється, чи цей новий блок достатнього розміру для перевизначення. Якщо так, то в ньому створюється 2 блоки: зайнятий і вільний. У зайнятий копіюються дані з колишнього блоку користувача. Якщо ж ні – йде пошук нового вільного блоку і, якщо він знайдений, виділяється блок для користувача і копіюються данні. Інакше – повертається NULL, а данні користувача не змінюютсья.

1. Звільнення пам’яті mem\_free(void \*addr). Помічаємо блок як вільний і об’єднуємj даний блок с сусідніми вільними блоками (максимум – 2).

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

 O(log(n)), де n – кількість блоків у пам’яті.

**Оцінка часу звільнення занятого блоку**:

O(1).

**Оцінка витрати пам'яті для зберігання службовох інформації:**

22 \* N байт, де N – кількість блоків у пам’яті.

**Переваги аллокатора:**

1. Невеликий об’єм службової інформації.
2. Малий час виконання звільнення блоку.
3. Об’єднання сусідніх вільних блоків в один.
4. Швидке виділення блоку

**Недоліки аллокатора:**

1. Білший розмір заголовків блоків, ніж в лаб1
2. Більш складна реалізація ніж в лаб 1

**Лістинг програми (мова С):**

**allocator.h**

|  |
| --- |
| #include <iostream> |
|  | #include <cstring> |
|  |  |
|  | #define memory 1340 |
|  | #define page\_size 256 |
|  | #define info 12 |
|  |  |
|  |  |
|  | typedef struct s\_page { |
|  | void\* address; |
|  | char state; |
|  | size\_t size; |
|  | } mem\_page; |
|  |  |
|  | void initialise\_memory(size\_t size); |
|  |  |
|  | void\* mem\_alloc(size\_t size); |
|  |  |
|  | void\* mem\_realloc(void\* address, size\_t size); |
|  |  |
|  | void mem\_free(void\* address); |
|  |  |
|  | void mem\_dump(); |
|  |  |
|  | extern mem\_page\* global\_memory; |
|  |  |
|  | extern size\_t page\_count; |

## ****allocator.cpp****

|  |
| --- |
| #include "allocator.h" |
|  |  |
|  | mem\_page\* global\_memory; |
|  | size\_t page\_count; |
|  |  |
|  | void initialise\_memory(size\_t size) |
|  | { |
|  | global\_memory = (mem\_page\*)calloc(memory, 1); |
|  | page\_count = size / (page\_size + info); |
|  |  |
|  | for (size\_t i = 0; i < page\_count; i++) |
|  | { |
|  | global\_memory[i].address = (void\*)((unsigned long int)global\_memory + page\_count \* info + i \* page\_size); |
|  | global\_memory[i].size = 0; |
|  | global\_memory[i].state = 0; |
|  | } |
|  |  |
|  | } |
|  |  |
|  | void\* mem\_alloc(size\_t size) |
|  | { |
|  | void\* address; |
|  | size\_t counter = 0; |
|  | size\_t firstpage = 0; |
|  | size\_t pages\_alloc = (double)size / page\_size + 0.5; |
|  | size\_t i; |
|  | if (size <= page\_size) |
|  | { |
|  | for (i = 0; i < page\_count; i++) |
|  | { |
|  | if (global\_memory[i].state == 0 || (global\_memory[i].state == 1 && size <= page\_size / 2)) |
|  | { |
|  | address = (void\*)((size\_t)(global\_memory[i].address) + (global\_memory[i].state == 1 ? page\_size / 2 : 0)); |
|  | if (size > page\_size / 2) |
|  | { |
|  | global\_memory[i].state = 2; |
|  | global\_memory[i].size = 1; |
|  | } |
|  | else |
|  | { |
|  | global\_memory[i].state = global\_memory[i].state == 1 ? 2 : 1; |
|  | global\_memory[i].size = 0; |
|  | } |
|  | return address; |
|  | } |
|  | } |
|  | } |
|  | else |
|  | { |
|  | for (i = 0; i < page\_count; i++) |
|  | { |
|  | if (global\_memory[i].state == 0) |
|  | { |
|  | counter += 1; |
|  | firstpage = i - counter + 1; |
|  | if (counter == pages\_alloc) |
|  | { |
|  | for (size\_t j = firstpage; j < firstpage + counter; j++) |
|  | { |
|  | global\_memory[j].state = 3; |
|  | } |
|  | global\_memory[firstpage].size = pages\_alloc \* page\_size; |
|  | address = global\_memory[firstpage].address; |
|  | return address; |
|  | } |
|  | } |
|  | else |
|  | { |
|  | counter = 0; |
|  | firstpage = 0; |
|  | } |
|  | } |
|  | } |
|  | return NULL; |
|  | } |
|  |  |
|  | void\* mem\_realloc(void\* address, size\_t size) |
|  | { |
|  | void\* new\_address; |
|  | size\_t size\_of\_block = (size\_t)address - (size\_t)global\_memory - info \* page\_count; |
|  | size\_t started\_block\_page = size\_of\_block / page\_size; |
|  | if (global\_memory[started\_block\_page].state == 3) |
|  | size\_of\_block = global\_memory[started\_block\_page].size; |
|  | else if (global\_memory[started\_block\_page].size != 0) |
|  | size\_of\_block = global\_memory[started\_block\_page].size \* page\_size; |
|  | else |
|  | size\_of\_block = page\_size / 2; |
|  |  |
|  | new\_address = mem\_alloc(size); |
|  | if (new\_address) |
|  | { |
|  | memmove(new\_address, address, size\_of\_block > size ? size : size\_of\_block); |
|  | mem\_free(address); |
|  | return new\_address; |
|  | } |
|  | return NULL; |
|  | } |
|  |  |
|  | void mem\_free(void\* address) |
|  | { |
|  | size\_t size\_of\_block = (size\_t)address - (size\_t)global\_memory - info \* page\_count; |
|  | size\_t started\_block\_page = size\_of\_block / page\_size; |
|  | size\_t page\_state = global\_memory[started\_block\_page].state; |
|  | if (page\_state == 1) |
|  | global\_memory[started\_block\_page].state = 0; |
|  | else if (page\_state == 2) |
|  | { |
|  | if (global\_memory[started\_block\_page].size == 0) |
|  | global\_memory[started\_block\_page].state = 1; |
|  | else |
|  | global\_memory[started\_block\_page].state = 0; |
|  | } |
|  |  |
|  | else if (page\_state == 3) |
|  | for (size\_t i = started\_block\_page; i < global\_memory[started\_block\_page].size / page\_size; i++) |
|  | global\_memory[i].state = 0; |
|  | std::memset(address, 0, global\_memory[started\_block\_page].size == 0 ? page\_size / 2 : global\_memory[started\_block\_page].size); |
|  | global\_memory[started\_block\_page].size = 0; |
|  | address = NULL; |
|  |  |
|  |  |
|  | } |

## ****print.cpp****

|  |
| --- |
| #include <stdio.h> |
|  | #include "allocator.h" |
|  |  |
|  |  |
|  | void mem\_dump() |
|  | { |
|  | printf("\nMemory Dump\n"); |
|  | printf("Global address: %p\n", global\_memory); |
|  | for (size\_t i = 0; i < page\_count; i++) |
|  | { |
|  | printf("-----------------------------\n"); |
|  | printf("Page: %i \nState: %3i \nAddress: %p\n", i, global\_memory[i].state, global\_memory[i].address); |
|  | printf("Data: %s\n", (char\*)global\_memory[i].address); |
|  | printf("-----------------------------\n"); |
|  | } |
|  | } |

## ****Lab2.cpp****

|  |
| --- |
| #include "allocator.h" |
|  |  |
|  | int main() |
|  | { |
|  | initialise\_memory(memory); |
|  |  |
|  | void\* data = mem\_alloc(300); |
|  | memmove(data, "Block #1", info); |
|  |  |
|  | void\* data1 = mem\_alloc(200); |
|  | memmove(data1, "Block #2", info); |
|  |  |
|  | void\* data2 = mem\_alloc(200); |
|  | memmove(data2, "Block #3", info); |
|  |  |
|  | void\* data3 = mem\_alloc(600); |
|  | memmove(data3, "Block #4", info); |
|  |  |
|  | mem\_dump(); |
|  | mem\_realloc(data2, 100); |
|  | mem\_free(data); |
|  | mem\_dump(); |
|  | return 0; |
|  | } |

**Висновок:**

В результаті виконання лабораторної роботи було розроблено програму алокатора пам'яті загального призначення на основі бінарного дерева. Було реалізовано функції mem\_alloc(), mem\_realloc(), mem\_free(), що складають інтерфейс алокатора пам'яті загального призначення.