**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 №1**

**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 1).**

**Мета: розробити аллокатор загального призначення.**

**Опис розробленого алгоритму:**

1. Виділення пам’яті через створення масиву: Заголовок займає 17 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).

5.

Оцінка часу пошуку вільного блоку пам’яті:

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

Оцінка часу звільнення занятого блоку:

O(1).

Оцінка витрати пам'яті для зберігання службовох інформації:

17 \* 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.

Недоліки аллокатора:

1. Великий час пошуку нового блоку( лінійна складність)
2. При виділенні блоку можливе надмірне виділення пам’яті у розмірі, меншому за розмір заголовку.

**Лістинг програми (мова С):**

## ****allocator.h****

|  |
| --- |
| #include <stdlib.h> |
|  | #include <stdbool.h> |
|  |  |
|  | #define DEFAULT\_POOL\_SIZE 100 \* 1024 \*1024 |
|  |  |
|  | typedef struct block\_header { |
|  | bool allocated; |
|  | size\_t size; |
|  | struct block\_header\* prev; |
|  | struct block\_header\* next; |
|  | } block\_header\_t; |
|  |  |
|  | bool mem\_pool\_alloc(size\_t size); |
|  |  |
|  | void\* mem\_alloc(size\_t size); |
|  |  |
|  | void\* mem\_realloc(void\* addr, size\_t size); |
|  |  |
|  | void mem\_free(void\* addr); |
|  |  |
|  | void mem\_pool\_free(void); |
|  |  |
|  | typedef struct { |
|  | block\_header\_t\* head; |
|  | size\_t size; |
|  | size\_t length; |
|  | } blocks\_status\_t; |
|  |  |
|  | blocks\_status\_t get\_blocks\_status(void); |

**allocator.cpp**

|  |
| --- |
| #include <string.h> |
|  | #include "allocator.h" |
|  |  |
|  | struct mem\_pool\_t { |
|  | void\* data; |
|  | size\_t size; |
|  | } mem\_pool; |
|  |  |
|  | static block\_header\_t\* get\_head(struct mem\_pool\_t pool); |
|  | static size\_t block\_size(block\_header\_t\* block); |
|  | static block\_header\_t\* split\_block(block\_header\_t\* block, size\_t size); |
|  | static block\_header\_t\* get\_block\_header(void\* addr); |
|  | static block\_header\_t\* merge\_blocks(block\_header\_t\* base, block\_header\_t\* appendix); |
|  | static void\* get\_block\_payload(block\_header\_t\* block); |
|  | static size\_t align(size\_t size); |
|  |  |
|  | static block\_header\_t\* get\_block\_header(void\* addr) { |
|  | return (block\_header\_t\*)((size\_t)addr - sizeof(block\_header\_t)); |
|  | }; |
|  |  |
|  | static void\* get\_block\_payload(block\_header\_t\* block) { |
|  | return (void\*)((size\_t)block + sizeof(block\_header\_t)); |
|  | }; |
|  |  |
|  | static size\_t align(size\_t size) { |
|  | size\_t offset = size % 4; |
|  | return offset ? size + (4 - offset) : size; |
|  | }; |
|  |  |
|  | static block\_header\_t\* get\_head(struct mem\_pool\_t pool) { |
|  | return (block\_header\_t\*)pool.data; |
|  | }; |
|  |  |
|  | static size\_t block\_size(block\_header\_t\* block) { |
|  | if (block->next == NULL) { |
|  | return 0; |
|  | } |
|  |  |
|  | size\_t begin = (size\_t)block + sizeof(block\_header\_t); |
|  | size\_t end = (size\_t)(block->next); |
|  | return end - begin; |
|  | }; |
|  |  |
|  | blocks\_status\_t get\_blocks\_status(void) { |
|  | block\_header\_t\* block = get\_head(mem\_pool); |
|  |  |
|  | size\_t count = 1; |
|  | while (block->next != NULL) { |
|  | count++; |
|  | block = block->next; |
|  | } |
|  |  |
|  | blocks\_status\_t info = { |
|  | get\_head(mem\_pool), |
|  | mem\_pool.size, |
|  | count |
|  | }; |
|  |  |
|  | return info; |
|  | }; |
|  |  |
|  | bool mem\_pool\_alloc(size\_t size) { |
|  | if (mem\_pool.data != NULL) { |
|  | return false; |
|  | } |
|  | if (size < 2 \* sizeof(block\_header\_t)) { |
|  | size = DEFAULT\_POOL\_SIZE; |
|  | } |
|  | mem\_pool.size = size; |
|  |  |
|  | mem\_pool.data = &mem\_pool.size; |
|  | if (mem\_pool.data == NULL) { |
|  | return false; |
|  | } |
|  |  |
|  | block\_header\_t\* head = get\_head(mem\_pool); |
|  | block\_header\_t\* tail = (block\_header\_t\*)( |
|  | (size\_t)mem\_pool.data + mem\_pool.size - sizeof(block\_header\_t) |
|  | ); |
|  |  |
|  | head->allocated = false; |
|  | head->prev = NULL; |
|  | head->next = tail; |
|  | head->size = block\_size(head); |
|  |  |
|  | tail->allocated = true; |
|  | tail->size = 0; |
|  | tail->prev = tail; |
|  | tail->next = NULL; |
|  |  |
|  | return true; |
|  | } |
|  |  |
|  | void\* mem\_alloc(size\_t size) { |
|  | size = align(size); |
|  | block\_header\_t\* block = get\_head(mem\_pool); |
|  | while (block->allocated || block->size < size) { |
|  | if (block->next == NULL) { |
|  | return NULL; |
|  | } |
|  | block = block->next; |
|  | } |
|  |  |
|  | block = split\_block(block, size); |
|  | block->allocated = true; |
|  |  |
|  | return get\_block\_payload(block); |
|  | }; |
|  |  |
|  | void\* mem\_realloc(void\* addr, size\_t size) { |
|  | if (addr == NULL) { |
|  | return mem\_alloc(size); |
|  | } |
|  | size = align(size); |
|  |  |
|  | block\_header\_t\* block = get\_block\_header(addr); |
|  | if (block->size >= size) { |
|  | split\_block(block, size); |
|  | return addr; |
|  | } |
|  |  |
|  | const size\_t block\_header\_size = sizeof(block\_header\_t); |
|  | block\_header\_t\* next\_block = block->next; |
|  | if ( |
|  | next\_block != NULL && |
|  | !next\_block->allocated && |
|  | block->size + next\_block->size + block\_header\_size >= size |
|  | ) { |
|  | merge\_blocks(block, next\_block); |
|  | split\_block(block, size); |
|  |  |
|  | return addr; |
|  | } |
|  |  |
|  | block\_header\_t\* prev\_block = block->prev; |
|  | if ( |
|  | prev\_block != NULL && |
|  | !prev\_block->allocated && |
|  | block->size + prev\_block->size + block\_header\_size >= size |
|  | ) { |
|  | size\_t data\_size = block->size; |
|  |  |
|  | block->allocated = false; |
|  | block = merge\_blocks(prev\_block, block); |
|  |  |
|  | block->allocated = true; |
|  | void\* block\_data = get\_block\_payload(block); |
|  | memcpy(block\_data, addr, data\_size); |
|  | split\_block(block, size); |
|  |  |
|  | return block\_data; |
|  | } |
|  |  |
|  | void\* temp = mem\_alloc(size); |
|  | if (temp != NULL) { |
|  | memcpy(temp, addr, block->size); |
|  | mem\_free(addr); |
|  | } |
|  |  |
|  | return temp; |
|  | }; |
|  |  |
|  | void mem\_pool\_free(void) { |
|  | if (mem\_pool.data != NULL) { |
|  | mem\_pool.data = NULL; |
|  | mem\_pool.size = 0; |
|  | } |
|  | }; |
|  |  |
|  | void mem\_free(void\* addr) { |
|  | block\_header\_t\* block = get\_block\_header(addr); |
|  | block->allocated = false; |
|  |  |
|  | if (!block->next->allocated) { |
|  | merge\_blocks(block, block->next); |
|  | } |
|  |  |
|  | if (block->prev != NULL && !block->prev->allocated) { |
|  | merge\_blocks(block->prev, block); |
|  | } |
|  | }; |
|  |  |
|  | static block\_header\_t\* split\_block(block\_header\_t\* block, size\_t size) { |
|  | if (block->size < size) { |
|  | return NULL; |
|  | } |
|  |  |
|  | size\_t remaining = block->size - size; |
|  | if (remaining >= sizeof(block\_header\_t)) { |
|  | block\_header\_t\* new\_block = (block\_header\_t\*)( |
|  | (size\_t)get\_block\_payload(block) + size |
|  | ); |
|  |  |
|  | new\_block->allocated = false; |
|  |  |
|  | new\_block->prev = block; |
|  | new\_block->next = block->next; |
|  | block->next->prev = new\_block; |
|  | block->next = new\_block; |
|  |  |
|  | new\_block->size = block\_size(new\_block); |
|  | block->size = size; |
|  | } |
|  |  |
|  | return block; |
|  | }; |
|  |  |
|  | static block\_header\_t\* merge\_blocks(block\_header\_t\* base, block\_header\_t\* appendix) { |
|  | if (appendix->allocated || base->next != appendix) { |
|  | return NULL; |
|  | } |
|  |  |
|  | block\_header\_t\* next = appendix->next; |
|  | base->next = next; |
|  | next->prev = base; |
|  |  |
|  | base->size = block\_size(base); |
|  | return base; |
|  | }; |

## ****output.cpp****

|  |
| --- |
| #include <stdio.h> |
|  | #include "allocator.h" |
|  |  |
|  | void mem\_dump(void) { |
|  | blocks\_status\_t info = get\_blocks\_status(); |
|  |  |
|  | printf("\n========================================\n"); |
|  | printf("Memory pool size: %ld\n", info.size); |
|  | printf("Number of blocks: %ld\n", info.length); |
|  |  |
|  | block\_header\_t\* block = info.head; |
|  | for (size\_t id = 1; id <= info.length; id++) { |
|  | printf("----------------------------------------\n"); |
|  | printf( |
|  | "Block #%ld\n" |
|  | "Address: %p\n" |
|  | "Previous: %p\n" |
|  | "Next: %p\n" |
|  | "Size: %ld\n" |
|  | "Status: %s\n", |
|  | id, |
|  | block, block->prev, block->next, |
|  | block->size, |
|  | block->allocated ? "allocated" : "free" |
|  | ); |
|  | block = block->next; |
|  | } |
|  | printf("========================================\n\n"); |
|  | }; |

## ****Lab1.cpp****

|  |
| --- |
| #include <stdio.h> |
|  | #include "allocator.h" |
|  | #include "output.h" |
|  |  |
|  | int main(int argc, char\* argv[]) { |
|  | printf("Allocate memory pool.\n"); |
|  | printf("Result: %d\n", mem\_pool\_alloc(1536)); |
|  | printf("Result: %d\n", mem\_pool\_alloc(0)); |
|  | mem\_dump(); |
|  |  |
|  | // Allocate blocks. |
|  | printf("Allocate memory block #1.\n"); |
|  | void\* arr1 = mem\_alloc(100 \* sizeof(int)); |
|  |  |
|  | mem\_dump(); |
|  |  |
|  | printf("Allocate memory block #2.\n"); |
|  | void\* arr2 = mem\_alloc(100 \* sizeof(int)); |
|  |  |
|  | mem\_dump(); |
|  |  |
|  | printf("Allocate memory block #3.\n"); |
|  | void\* arr3 = mem\_alloc(100 \* sizeof(int)); |
|  |  |
|  | mem\_dump(); |
|  |  |
|  | printf("Allocate memory block #4.\n"); |
|  | void\* arr4 = mem\_alloc(100 \* sizeof(int)); |
|  | mem\_dump(); |
|  |  |
|  | printf("Realloc memory block #1\n"); |
|  | void\* arr1\_1 = mem\_realloc(arr1, 92 \* sizeof(int)); |
|  | printf("Realloc memory block #2\n"); |
|  | void\* arr2\_1 = mem\_realloc(arr2, 101 \* sizeof(int)); |
|  | printf("Realloc memory block #3\n"); |
|  | void\* arr3\_1 = mem\_realloc(arr3, 101 \* sizeof(int)); |
|  |  |
|  | if (arr1\_1 == NULL || arr2\_1 == NULL || arr3\_1 == NULL) { |
|  | perror("Relloc failed."); |
|  | } |
|  |  |
|  | if (arr1\_1 != NULL) { |
|  | printf("Free memory block #3.\n"); |
|  | mem\_free(arr1\_1); |
|  | mem\_dump(); |
|  | } |
|  |  |
|  | if (arr3\_1 != NULL) { |
|  | printf("Free memory block #1.\n"); |
|  | mem\_free(arr3\_1); |
|  | mem\_dump(); |
|  | } |
|  |  |
|  | if (arr2\_1 != NULL) { |
|  | printf("Free memory block #2.\n"); |
|  | mem\_free(arr2\_1); |
|  | mem\_dump(); |
|  | } |
|  |  |
|  |  |
|  | printf("Free memory pool.\n"); |
|  | mem\_pool\_free(); |
|  | mem\_pool\_free(); |
|  |  |
|  | return 0; |
|  | } |

**Висновок:**

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